Writing Interpreters in Rust: a Guide

Welcome!

In this book we will walk through the basics of interpreted language implementation in Rust with a focus on the challenges that are specific to using Rust.

At a glance, these are:

- A custom allocator for use in an interpreter
- A safe-Rust wrapper over allocation
- A compiler and VM that interact with the above two layers

The goal of this book is not to cover a full featured language but rather to provide a solid foundation on which you can build further features. Along the way we'll implement as much as possible in terms of our own memory management abstractions rather than using Rust std collections.

Level of difficulty

Bob Nystrom's Crafting Interpreters is recommended *introductory* reading to this book for beginners to the topic. Bob has produced a high quality, accessible work and while there is considerable overlap, in some ways this book builds on Bob's work with some additional complexity, optimizations and discussions of Rust's safe vs unsafe.

We hope you find this book to be informative!

Further reading and other projects to study:

All the links below are acknowledged as inspiration or prior art.

Interpreters

- Bob Nystrom's Crafting Interpreters
- The Inko programming language
- kyren - luster and gc-arena

Memory management

- Richard Jones, Anthony Hosking, Elliot Moss - The Garbage Collection Handbook
- Stephen M. Blackburn & Kathryn S. McKinley - Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance
- Felix S Klock II - GC and Rust Part 0: Garbage Collection Background
- Felix S Klock II - GC and Rust Part 1: Specifying the Problem
- Felix S Klock II - GC and Rust Part 2: The Roots of the Problem

Allocators

This section gives an overview and implementation detail of allocating blocks of memory.

What this is not: a custom allocator to replace the global Rust allocator

Alignment

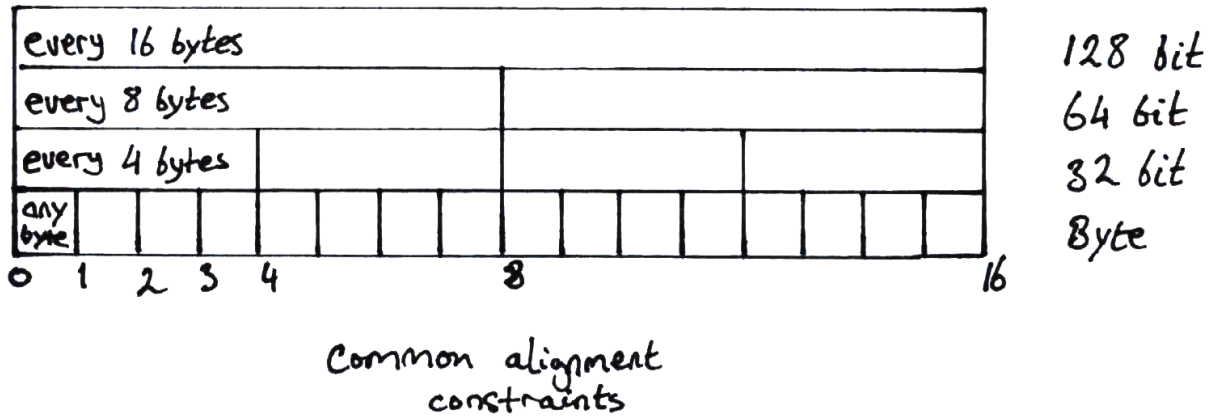
There are subtleties in memory access alignment:

- Some hardware architectures and implementations may fault on unaligned memory access.
- Atomic operations require word-aligned access.
- SIMD operations typically require double-word-aligned access.
- In practice on 64 bit architectures, allocators align objects to 8 byte boundaries for 64 bit objects and smaller and 16 byte boundaries for larger objects for performance optimization and the above reasons.

Intel 32 and 64 bit x86 architectures allow general access to be unaligned but will probably incur an access penalty. The story on 32bit ARM and aarch64 is sufficiently similar but there is a higher chance that an ARM core is configured to raise a bus error on a misaligned access.

Another very important factor is atomic memory operations. Atomic access works on a whole word basis - any unaligned access by nature cannot be guaranteed to be atomic as it will probably involve more than one access. To support atomic operations, alignment must be minmally on word boundaries.

SIMD operations, tending to be 128 bits wide or higher, should be aligned to 16 byte boundaries for optimal code generation and performance. Unaligned loads and stores may be allowed but normally these incur performance penalties.



While Intel allows unaligned access (that is, alignment on any byte boundary), the recommended (see section 3.6.4) alignment for objects larger than 64 bits is to 16 byte boundaries.

Apparently system `malloc()` implementations tend to comply with the 16 byte boundary.

To verify the above, a rough test of both the system allocator and jemalloc on x86_64 by using `Box::new()` on a set of types (`u8`, `u16`, `u32`, `u64`, `String` and a larger `struct`) confirms a minimum of 8 byte alignment for anything word size or smaller and 16 byte alignment for everything bigger. Sample pointer printouts below are for jemalloc but Linux libc malloc produced the same pattern:

```
p=0x7fb78b421028 u8
p=0x7fb78b421030 u16
p=0x7fb78b421038 u32
p=0x7fb78b421050 u64
p=0x7fb78b420060 "spam"
p=0x7fb78b4220f0 Hoge { y: 2, z: "ほげ", x: 1 }
```

Compare with `std::mem::align_of<T>()` which, on x86_64 for example, returns alignment values:

- `u8` : 1 byte
- `u16` : 2 bytes
- `u32` : 4 bytes
- `u64` : 8 bytes
- any bigger struct: 8

Thus despite the value of `std::mem::align_of::<T>()`, mature allocators will do what is most pragmatic and follow recommended practice in support of optimal performance.

With all that in mind, to keep things simple, we'll align everything to a double-word boundaries. When we add in prepending an object header, the minimum memory

required for an object will be two words anyway.

Thus, the allocated size of an object will be calculated[1] by

```
let alignment = size_of::<usize>() * 2;
// mask out the least significant bits that correspond to the alignment - 1
// then add the full alignment
let size = (size_of::<T>() & !(alignment - 1)) + alignment;
```

---

[1] For a more detailed explanation of alignment adjustment calculations, see phil-opp's kernel heap allocator.

Obtaining Blocks of Memory

When requesting blocks of memory at a time, one of the questions is *what is the desired block alignment?*

- In deciding, one factor is that using an alignment that is a multiple of the page size can make it easier to return memory to the operating system.
- Another factor is that if the block is aligned to it's size, it is fast to do bitwise arithmetic on a pointer to an object in a block to compute the block boundary and therefore the location of any block metadata.

With both these in mind we'll look at how to allocate blocks that are aligned to the size of the block.

A basic crate interface

A block of memory is defined as a base address and a size, so we need a struct that contains these elements.

To wrap the base address pointer, we'll use the recommended type for building collections, `std::ptr::NonNull<T>`, which is available on stable.

```
pub struct Block {
    ptr: BlockPtr,
    size: BlockSize,
}
```

Where `BlockPtr` and `BlockSize` are defined as:

```
pub type BlockPtr = NonNull<u8>;
pub type BlockSize = usize;
```

To obtain a `Block`, we'll create a `Block::new()` function which, along with `Block::drop()`, is implemented internally by wrapping the stabilized Rust alloc routines:

```
    pub fn new(size: BlockSize) -> Result<Block, BlockError> {
        if !size.is_power_of_two() {
            return Err(BlockError::BadRequest);
        }

        Ok(Block {
            ptr: internal::alloc_block(size)?,
            size,
        })
    }
```

Where parameter `size` must be a power of two, which is validated on the first line of the function. Requiring the block size to be a power of two means simple bit arithmetic can be used to find the beginning and end of a block in memory, if the block size is always the same.

Errors take one of two forms, an invalid block-size or out-of-memory, both of which may be returned by `Block::new()`.

```
#[derive(Debug, PartialEq)]
pub enum BlockError {
    /// Usually means requested block size, and therefore alignment, wasn't a
    /// power of two
    BadRequest,
    /// Insufficient memory, couldn't allocate a block
    OOM,
}
```

Now on to the platform-specific implementations.

Custom aligned allocation on stable Rust

On the stable rustc channel we have access to some features of the Alloc API.

This is the ideal option since it abstracts platform specifics for us, we do not need to write different code for Unix and Windows ourselves.

Fortunately there is enough stable functionality to fully implement what we need.

With an appropriate underlying implementation this code should compile and execute for any target. The allocation function, implemented in the `internal` mod, reads:

```
pub fn alloc_block(size: BlockSize) -> Result<BlockPtr, BlockError> {
    unsafe {
        let layout = Layout::from_size_align_unchecked(size, size);

        let ptr = alloc(layout);
        if ptr.is_null() {
            Err(BlockError::OOM)
        } else {
            Ok(NonNull::new_unchecked(ptr))
        }
    }
}
```

Once a block has been allocated, there is no safe abstraction at this level to access the memory. The `Block` will provide a bare pointer to the beginning of the memory and it is up to the user to avoid invalid pointer arithmetic and reading or writing outside of the block boundary.

```
pub fn as_ptr(&self) -> *const u8 {
    self.ptr.as_ptr()
}
```

Deallocation

Again, using the stable Alloc functions:

```
pub fn dealloc_block(ptr: BlockPtr, size: BlockSize) {
    unsafe {
        let layout = Layout::from_size_align_unchecked(size, size);

        dealloc(ptr.as_ptr(), layout);
    }
}
```

The implementation of `Block::drop()` calls the deallocation function for us so we can create and drop `Block` instances without leaking memory.

Testing

We want to be sure that the system level allocation APIs do indeed return block-size-aligned blocks. Checking for this is straightforward.

A correctly aligned block should have it's low bits set to `0` for a number of bits that represents the range of the block size - that is, the block size minus one. A bitwise XOR

will highlight any bits that shouldn't be set:

```
        // the block address bitwise AND the alignment bits (size - 1) should
        // be a mutually exclusive set of bits
        let mask = size - 1;
        assert!((block.ptr.as_ptr() as usize & mask) ^ mask == mask);
```

The type of allocation

Before we start writing objects into `Block`s, we need to know the nature of the interface in Rust terms.

If we consider the global allocator in Rust, implicitly available via `Box::new()`, `Vec::new()` and so on, we'll notice that since the global allocator is available on every thread and allows the creation of new objects on the heap (that is, mutation of the heap) from any code location without needing to follow the rules of borrowing and mutable aliasing, it is essentially a container that implements `Sync` and the interior mutability pattern.

We need to follow suit, but we'll leave `Sync` for advanced chapters.

An interface that satisfies the interior mutability property, by borrowing the allocator instance immutably, might look like:

```
trait AllocRaw {
    fn alloc<T>(&self, object: T) -> *const T;
}
```

naming it `AllocRaw` because when layering on top of `Block` we'll work with raw pointers and not concern ourselves with the lifetime of allocated objects.

It will become a little more complex than this but for now, this captures the essence of the interface.

An allocator: Sticky Immix

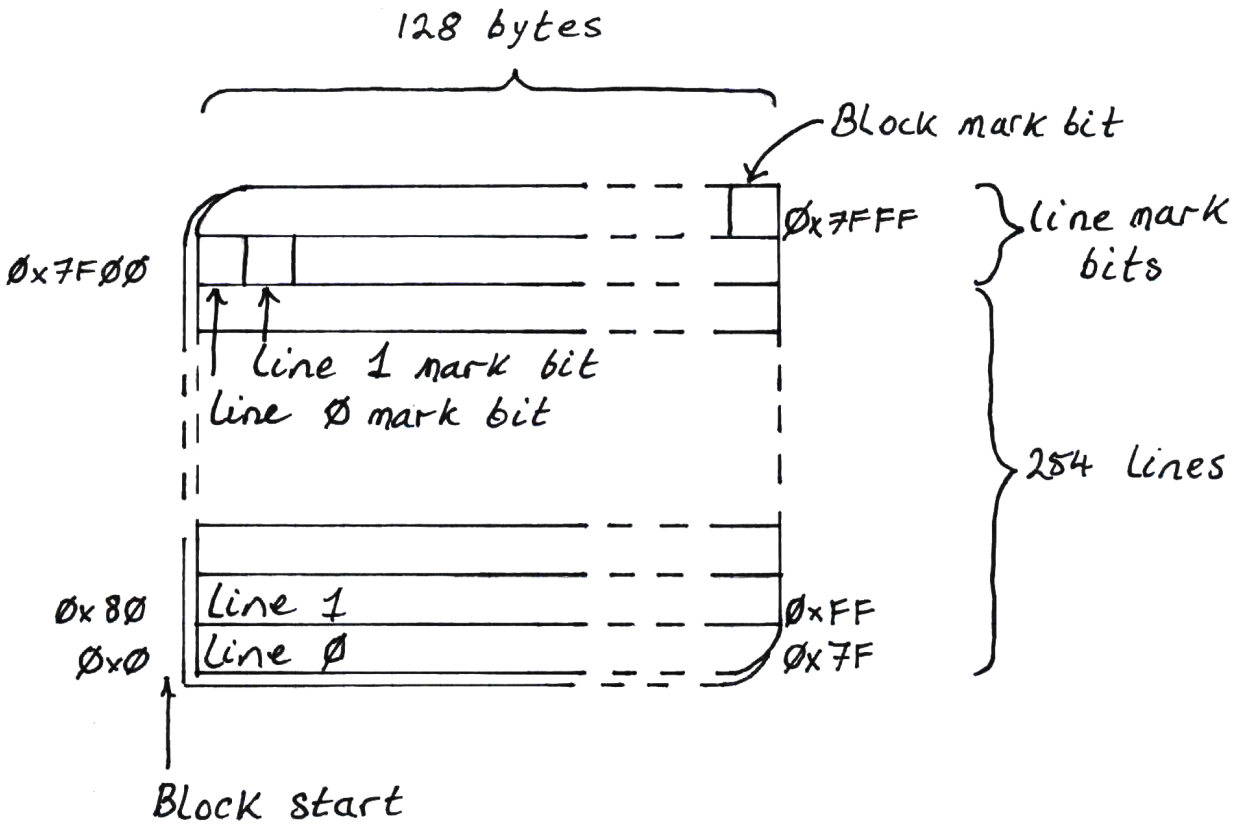
Quickly, some terminology:

- Mutator: the thread of execution that writes and modifies objects on the heap.
- Live objects: the graph of objects that the mutator can reach, either directly from it's stack or indirectly through other reachable objects.
- Dead objects: any object that is disconnected from the mutator's graph of live objects.
- Collector: the thread of execution that identifies objects that are no longer reachable by the mutator and marks them as free space that can be reused

- Fragmentation: as objects have many different sizes, after allocating and freeing many objects, gaps of unused memory appear between objects that are too small for most objects but that add up to a measurable percentage of wasted space.
- Evacuation: when the collector *moves* live objects to another block of memory so that the originating block can be *de_fragmented*

About Immix

Immix is a memory management scheme that considers blocks of fixed size at a time. Each block is divided into lines. In the original paper, blocks are sized at 32k and lines at 128 bytes. Objects are allocated into blocks using bump allocation and objects can cross line boundaries.



During tracing to discover live objects, objects are marked as live, but the line, or lines, that each object occupies are also marked as live. This can mean, of course, that a line may contain a dead object and a live object but the whole line is marked as live.

To mark lines as live, a portion of the block is set aside for line mark bits, usually one byte per mark bit. If *any* line is marked as live, the whole block is also marked as live. There must also, therefore, be a bit that indicates block liveness.

Conservative marking

The Immix authors found that marking *every* line that contains a live object could be expensive. For example, many small objects might cross line boundaries, requiring two lines to be marked as live. This would require looking up the object size and calculating whether the object crosses the boundary into the next line. To save CPU cycles, they simplified the algorithm by saying that any object that fits in a line *might* cross into the next line so we will conservatively *consider* the next line marked just in case. This sped up marking at little fragmentation expense.

Collection

During collection, only lines not marked as live are considered available for re-use. Inevitably then, there is acceptance of some amount of fragmentation at this point.

Full Immix implements evacuating objects out of the most fragmented blocks into fresh, empty blocks, for defragmentation.

For simplicity of implementation, we'll leave out this evacuation operation in this guide. This is called *Sticky* Immix.

We'll also stick to a single thread for the mutator and collector to avoid the complexity overhead of a multi-threaded implementation for now.

Recommended reading: Stephen M. Blackburn & Kathryn S. McKinley - Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance

About this part of the book

This section will describe a Rust crate that implements a Sticky Immix heap. As part of this implementation we will dive into the crate API details to understand how we can define an interface between the heap and the language VM that will come later.

What this is not: custom memory management to replace the global Rust allocator! The APIs we arrive at will be substantially incompatible with the global Rust allocator.

Bump allocation

Now that we can get blocks of raw memory, we need to write objects into it. The simplest way to do this is to write objects into a block one after the other in consecutive order. This is bump allocation - we have a pointer, the bump pointer, which points at the space in the block after the last object that was written. When the next object is written, the bump

pointer is incremented to point to the space after *that* object.

In a twist of mathematical convenience, though, it is more efficient to bump allocate from a high memory location *downwards*. We will do that.

We will used a fixed power-of-two block size. The benefit of this is that given a pointer to an object, by zeroing the bits of the pointer that represent the block size, the result points to the beginning of the block. This will be useful later when implementing garbage collection.

Our block size will be 32k, a reasonably optimal size arrived at in the original Immix paper. This size can be any power of two though and different use cases may show different optimal sizes.

```
pub const BLOCK_SIZE_BITS: usize = 15;
pub const BLOCK_SIZE: usize = 1 << BLOCK_SIZE_BITS;
```

Now we'll define a struct that wraps the block with a bump pointer and garbage collection metadata:

```
pub struct BumpBlock {
    cursor: *const u8,
    limit: *const u8,
    block: Block,
    meta: BlockMeta,
}
```

Bump allocation basics

In this struct definition, there are two members that we are interested in to begin with. The other two, `limit` and `meta`, will be discussed in the next section.

- `cursor`: this is the bump pointer. In our implementation it is the index into the block where the last object was written.
- `block`: this is the `Block` itself in which objects will be written.

Below is a start to a bump allocation function:

```
impl BumpBlock {
    pub fn inner_alloc(&mut self, alloc_size: usize) -> Option<*const u8> {
        let block_start_ptr = self.block.as_ptr() as usize;
        let cursor_ptr = self.cursor as usize;

        // align to word boundary
        let align_mask: usize = !(size_of::<usize>() - 1);

        let next_ptr = cursor_ptr.checked_sub(alloc_size)? & align_mask;

        if next_ptr < block_start_ptr {
            // allocation would start lower than block beginning, which means
            // there isn't space in the block for this allocation
            None
        } else {
            self.cursor = next_ptr as *const u8;
            Some(next_ptr as *const u8)
        }
    }
}
```

In our function, the `alloc_size` parameter should be a number of bytes of memory requested.

The value of `alloc_size` may produce an unaligned pointer at which to write the object. Fortunately, by bump allocating downward we can apply a simple mask to the pointer to align it down to the nearest word:

```
        let align_mask: usize = !(size_of::<usize>() - 1);
```

In initial implementation, allocation will simply return `None` if the block does not have enough capacity for the requested `alloc_size`. If there *is* space, it will be returned as a `Some(*const u8)` pointer.

Note that this function does not *write* the object to memory, it merely returns a pointer to an available space. Writing the object will require invoking the `std::ptr::write` function. We will do that in a separate module but for completeness of this chapter, this might look something like:

```
use std::ptr::write;

unsafe fn write<T>(dest: *const u8, object: T) {
    write(dest as *mut T, object);
}
```

Some time passes...

After allocating and freeing objects, we will have gaps between objects in a block that can be reused. The above bump allocation algorithm is unaware of these gaps so we'll have to modify it before it can allocate into fragmented blocks.

To recap, in Immix, a block is divided into lines and only whole lines are considered for reuse. When objects are marked as live, so are the lines that an object occupies. Therefore, only lines that are *not* marked as live are usable for allocation into. Even if a line is only partially allocated into, it is not a candidate for further allocation.

In our implementation we will use the high bytes of the `Block` to represent these line mark bits, where each line is represented by a single byte.

We'll need a data structure to represent this. we'll call it `BlockMeta`, but first some constants that we need in order to know

- how big a line is
- how many lines are in a block
- how many bytes remain in the `Block` for allocating into

```
pub const LINE_SIZE_BITS: usize = 7;
pub const LINE_SIZE: usize = 1 << LINE_SIZE_BITS;

// How many total lines are in a block
pub const LINE_COUNT: usize = BLOCK_SIZE / LINE_SIZE;

// We need LINE_COUNT number of bytes for marking lines, so the capacity of a
block
// is reduced by that number of bytes.
pub const BLOCK_CAPACITY: usize = BLOCK_SIZE - LINE_COUNT;
```

For clarity, let's put some numbers to the definitions we've made so far:

- A block size is 32Kbytes
- A line is 128 bytes long
- The number of lines within a 32Kbyte `Block` is 256

Therefore the top 256 bytes of a `Block` are used for line mark bits. Since these line mark bits do not need to be marked themselves, the last *two bytes* of the `Block` are not needed to mark lines.

This leaves one last thing to mark: the entire `Block`. If *any* line in the `Block` is marked, then the `Block` is considered to be live and must be marked as such.

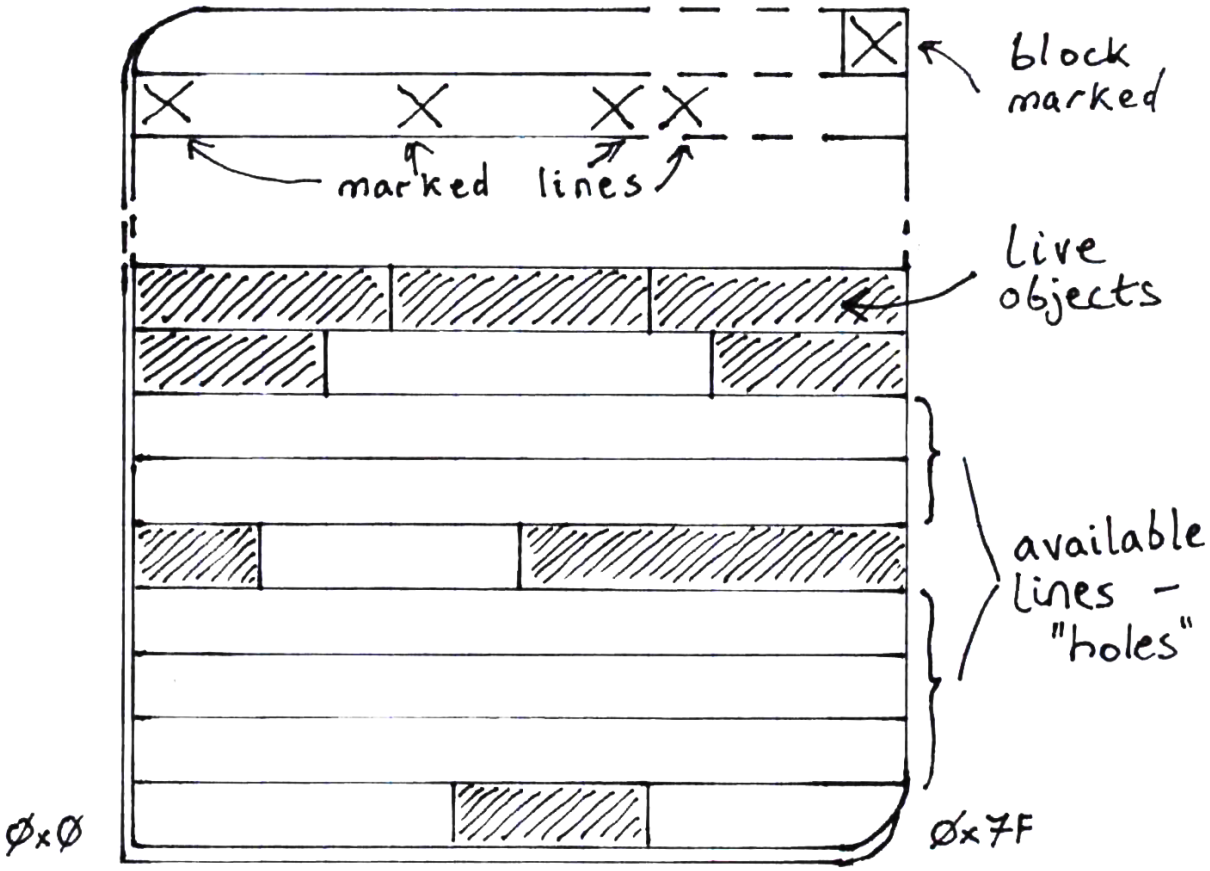
We use the final byte of the `Block` to store the `Block` mark bit.

The definition of `BumpBlock` contains member `meta` which is of type `BlockMeta`. We can now introduce the definition of `BlockMeta` which we simply need to represent a pointer to the line mark section at the end of the `Block`:

```
pub struct BlockMeta {
    lines: *mut u8,
}
```

This pointer could be easily calculated, of course, so this is just a handy shortcut.

Allocating into a fragmented Block



A fragmented block

The struct `BlockMeta` contains one function we will study:

```
    /// When it comes to finding allocatable holes, we bump-allocate
downward.
    pub fn find_next_available_hole(
        &self,
        starting_at: usize,
        alloc_size: usize,
    ) -> Option<(usize, usize)> {
        // The count of consecutive avaliable holes. Must take into account a
conservatively marked
        // hole at the beginning of the sequence.
        let mut count = 0;
        let starting_line = starting_at / constants::LINE_SIZE;
        let lines_required = (alloc_size + constants::LINE_SIZE - 1) /
constants::LINE_SIZE;
        // Counting down from the given search start index
        let mut end = starting_line;

        for index in (0..starting_line).rev() {
            let marked = unsafe { *self.lines.add(index) };

            if marked == 0 {
                // count unmarked lines
                count += 1;

                if index == 0 && count >= lines_required {
                    let limit = index * constants::LINE_SIZE;
                    let cursor = end * constants::LINE_SIZE;
                    return Some((cursor, limit));
                }
            } else {
                // This block is marked
                if count > lines_required {
                    // But at least 2 previous blocks were not marked. Return
the hole, considering the
                    // immediately preceding block as conservatively marked
                    let limit = (index + 2) * constants::LINE_SIZE;
                    let cursor = end * constants::LINE_SIZE;
                    return Some((cursor, limit));
                }

                // If this line is marked and we didn't return a new cursor/
limit pair by now,
                // reset the hole search state
                count = 0;
                end = index;
            }
        }

        None
    }
```

The purpose of this function is to locate a gap of unmarked lines of sufficient size to allocate an object of size `alloc_size` into.

The input to this function, `starting_at`, is the offset into the block to start looking for a

hole.

If no suitable hole is found, the return value is `None`.

If there are unmarked lines lower in memory than the `starting_at` point (bump allocating downwards), the return value will be a pair of numbers: `(cursor, limit)` where:

- `cursor` will be the new bump pointer value
- `limit` will be the lower bound of the available hole.

A deeper dive

Our first variable is a counter of consecutive available lines. This count will always assume that the first line in the sequence is conservatively marked and won't count toward the total, unless it is line 0.

```
        let mut count = 0;
```

Next, the `starting_at` and `alloc_size` arguments have units of bytes but we want to use line count math, so conversion must be done.

```
         let starting_line = starting_at / constants::LINE_SIZE;
         let lines_required = (alloc_size + constants::LINE_SIZE - 1) /
constants::LINE_SIZE;
```

Our final variable will be the end line that, together with `starting_line`, will mark the boundary of the hole we hope to find.

```
        let mut end = starting_line;
```

Now for the loop that identifies holes and ends the function if either:

- a large enough hole is found
- no suitable hole is found

We iterate over lines in decreasing order from `starting_line` down to line zero and fetch the mark bit into variable `marked`.

```
        for index in (0..starting_line).rev() {
            let marked = unsafe { *self.lines.add(index) };
```

If the line is unmarked, we increment our consecutive-unmarked-lines counter.

Then we reach the first termination condition: we reached line zero and we have a large enough hole for our object. The hole extents can be returned, converting back to byte offsets.

```
            if marked == 0 {
                count += 1;

                if index == 0 && count >= lines_required {
                    let limit = index * constants::LINE_SIZE;
                    let cursor = end * constants::LINE_SIZE;
                    return Some((cursor, limit));
                }
            } else {
```

Otherwise if the line is marked, we've reached the end of the current hole (if we were even over one.)

Here, we have the second possible termination condition: we have a large enough hole for our object. The hole extents can be returned, taking the last line as conservatively marked.

This is seen in adding 2 to `index`:

- 1 for walking back from the current marked line
- plus 1 for walking back from the previous conservatively marked line

If this condition isn't met, our search is reset - `count` is back to zero and we keep iterating.

```
            } else {
                if count > lines_required {
                    let limit = (index + 2) * constants::LINE_SIZE;
                    let cursor = end * constants::LINE_SIZE;
                    return Some((cursor, limit));
                }

                count = 0;
                end = index;
            }
```

Finally, if iterating over lines reached line zero without finding a hole, we return `None` to indicate failure.

```
        }

        None
    }
```

Making use of the hole finder

We'll return to the `BumpBlock::inner_alloc()` function now to make use of `BlockMeta` and its hole finding operation.

The `BumpBlock` struct contains two more members: `limit` and `meta`. These should now be obvious - `limit` is the known byte offset limit into which we can allocate, and `meta` is the `BlockMeta` instance associated with the block.

We need to update `inner_alloc()` with a new condition:

- the size being requested must fit between `self.cursor` and `self.limit`

(Note that for a fresh, new block, `self.limit` is set to the block size.)

If the above condition is not met, we will call `BlockMeta::find_next_available_hole()` to get a new `cursor` and `limit` to try, and repeat that until we've either *found* a big enough hole or reached the end of the block, exhausting our options.

The new definition of `BumpBlock::inner_alloc()` reads as follows:

```
    pub fn inner_alloc(&mut self, alloc_size: usize) -> Option<*const u8> {
        let ptr = self.cursor as usize;
        let limit = self.limit as usize;

        let next_ptr = ptr.checked_sub(alloc_size)? &
constants::ALLOC_ALIGN_MASK;

        if next_ptr < limit {
            let block_relative_limit =
                unsafe { self.limit.sub(self.block.as_ptr() as usize) } as
usize;

            if block_relative_limit > 0 {
                if let Some((cursor, limit)) = self
                    .meta
                    .find_next_available_hole(block_relative_limit,
alloc_size)
                {
                    self.cursor = unsafe { self.block.as_ptr().add(cursor) };
                    self.limit = unsafe { self.block.as_ptr().add(limit) };
                    return self.inner_alloc(alloc_size);
                }
            }

            None
        } else {
            self.cursor = next_ptr as *const u8;
            Some(self.cursor)
        }
    }
```

and as you can see, this implementation is recursive.

Wrapping this up

At the beginning of this chapter we stated that given a pointer to an object, by zeroing the bits of the pointer that represent the block size, the result points to the beginning of the block.

We'll make use of that now.

During the mark phase of garbage collection, we will need to know which line or lines to mark, in addition to marking the object itself. We will make a copy of the `BlockMeta` instance pointer in the 0th word of the memory block so that given any object pointer, we can obtain the `BlockMeta` instance.

In the next chapter we'll handle multiple `BumpBlock` s so that we can keep allocating objects after one block is full.

Allocating into Multiple Blocks

Let's now zoom out of the fractal code soup one level and begin arranging multiple blocks so we can allocate - in theory - indefinitely.

Lists of blocks

We'll need a new struct for wrapping multiple blocks:

```
struct BlockList {
    head: Option<BumpBlock>,
    overflow: Option<BumpBlock>,
    rest: Vec<BumpBlock>,
}
```

Immix maintains several lists of blocks. We won't include them all in the first iteration but in short they are:

- `free` : a list of blocks that contain no objects. These blocks are held at the ready to allocate into on demand
- `recycle` : a list of blocks that contain some objects but also at least one line that can be allocated into
- `large` : not a list of blocks, necessarily, but a list of objects larger than the block size, or some other method of accounting for large objects
- `rest` : the rest of the blocks that have been allocated into but are not suitable for recycling

In our first iteration we'll only keep the `rest` list of blocks and two blocks to immediately allocate into. Why two? To understand why, we need to understand how Immix thinks

about object sizes.

Immix and object sizes

We've seen that there are two numbers that define granularity in Immix: the block size and the line size. These numbers give us the ability to categorize object sizes:

- small: those that (with object header and alignment overhead) fit inside a line
- medium: those that (again with object header and alignment overhead) are larger than one line but smaller than a block
- large: those that are larger than a block

In the previous chapter we described the basic allocation algorithm: when an object is being allocated, the current block is scanned for a hole between marked lines large enough to allocate into. This does seem like it could be inefficient. We could spend a lot of CPU cycles looking for a big enough hole, especially for a medium sized object.

To avoid this, Immix maintains a second block, an overflow block, to allocate medium objects into that don't fit the first available hole in the main block being allocated into.

Thus two blocks to immediately allocate into:

- `head` : the current block being allocated into
- `overflow` : a block kept handy for writing medium objects into that don't fit the `head` block's current hole

We'll be ignoring large objects for now and attending only to allocating small and medium objects into blocks.

Instead of recycling blocks with holes, or maintaining a list of pre-allocated free blocks, we'll allocate a new block on demand whenever we need more space. We'll get to identifying holes and recyclable blocks in a later chapter.

Managing the overflow block

Generally in our code for this book, we will try to default to not allocating memory unless it is needed. For example, when an array is instantiated, the backing storage will remain unallocated until a value is pushed on to it.

Thus in the definition of `BlockList`, `head` and `overflow` are `Option` types and won't be instantiated except on demand.

For allocating into the overflow block we'll define a function in the `BlockList` impl:

```
impl BlockList {
    fn overflow_alloc(&mut self, alloc_size: usize) -> Result<*const u8,
AllocError> {
        ...
    }
}
```

The input constraint is that, since overflow is for medium objects, `alloc_size` must be less than the block size.

The logic inside will divide into three branches:

1. We haven't got an overflow block yet - `self.overflow` is `None`. In this case we have to instantiate a new block (since we're not maintaining a list of preinstantiated free blocks yet) and then, since that block is empty and we have a medium sized object, we can expect the allocation to succeed.

```
match self.overflow {
    Some ...,
    None => {
        let mut overflow = BumpBlock::new()?;

        // object size < block size means we can't fail this expect
        let space = overflow
            .inner_alloc(alloc_size)
            .expect("We expected this object to fit!");

        self.overflow = Some(overflow);

        space
    }
}
```

2. We *have* an overflow block and the object fits. Easy.

```
        match self.overflow {
            // We already have an overflow block to try to use...
            Some(ref mut overflow) => {
                // This is a medium object that might fit in the current
    block...
                match overflow.inner_alloc(alloc_size) {
                    // the block has a suitable hole
                    Some(space) => space,
                    ...
                }
            },
            None => ...
        }
```

3. We have an overflow block but the object does not fit. Now we simply instantiate a *new* overflow block, adding the old one to the `rest` list (in future it will make a good candidate for recycing!). Again, since we're writing a medium object into a block, we can expect allocation to succeed.

```
        match self.overflow {
            // We already have an overflow block to try to use...
            Some(ref mut overflow) => {
                // This is a medium object that might fit in the current
    block...
                match overflow.inner_alloc(alloc_size) {
                    Some ...,
                    // the block does not have a suitable hole
                    None => {
                        let previous = replace(overflow, BumpBlock::new()?);

                        self.rest.push(previous);

                        overflow.inner_alloc(alloc_size).expect("Unexpected
    error!")
                    }
                }
            },
            None => ...
        }
```

In this logic, the only error can come from failing to create a new block.

On success, at this level of interface we continue to return a `*const u8` pointer to the available space as we're not yet handling the type of the object being allocated.

You may have noticed that the function signature for `overflow_alloc` takes a `&mut self`. This isn't compatible with the interior mutability model of allocation. We'll have to wrap the `BlockList` struct in another struct that handles this change of API model.

The heap struct

This outer struct will provide the external crate interface and some further implementation of block management.

The crate interface will require us to consider object headers and so in the struct definition below there is reference to a generic type `H` that the *user* of the heap will define as the object header.

```
pub struct StickyImmixHeap<H> {
    blocks: UnsafeCell<BlockList>,

    _header_type: PhantomData<*const H>,
}
```

Since object headers are not owned directly by the heap struct, we need a `PhantomData` instance to associate with `H`. We'll discuss object headers in a later chapter.

Now let's focus on the use of the `BlockList`.

The instance of `BlockList` in the `StickyImmixHeap` struct is wrapped in an `UnsafeCell` because we need interior mutability. We need to be able to borrow the `BlockList` mutably while presenting an immutable interface to the outside world. Since we won't be borrowing the `BlockList` in multiple places in the same call tree, we don't need `RefCell` and we can avoid it's runtime borrow checking.

Allocating into the head block

We've already taken care of the overflow block, now we'll handle allocation into the `head` block. We'll define a new function:

```
impl StickyImmixHeap {
    fn find_space(
        &self,
        alloc_size: usize,
        size_class: SizeClass,
    ) -> Result<*const u8, AllocError> {
        let blocks = unsafe { &mut *self.blocks.get() };
        ...
    }
}
```

This function is going to look almost identical to the `alloc_overflow()` function defined earlier. It has more or less the same cases to walk through:

1. `head` block is `None`, i.e. we haven't allocated a head block yet. Allocate one and write the object into it.
2. We have `Some(ref mut head)` in `head`. At this point we divert from the `alloc_overflow()` function and query the size of the object - if this is is a medium object and the current hole between marked lines in the `head` block is too small, call into `alloc_overflow()` and return.

   ```
   if size_class == SizeClass::Medium && alloc_size > head.current_hole_size() {
       return blocks.overflow_alloc(alloc_size);
   }
   ```

   Otherwise, continue to allocate into `head` and return.
3. We have `Some(ref mut head)` in `head` but this block is unable to accommodate the object, whether medium or small. We must append the current head to the `rest` list and create a new `BumpBlock` to allocate into.

There is one more thing to mention. What about large objects? We'll cover those in a later chapter. Right now we'll make it an error to try to allocate a large object by putting this at the beginning of the `StickyImmixHeap::inner_alloc()` function:

```
if size_class == SizeClass::Large {
    return Err(AllocError::BadRequest);
}
```

Where to next?

We have a scheme for finding space in blocks for small and medium objects and so, in the next chapter we will define the external interface to the crate.

Defining the allocation API

Let's look back at the allocator prototype API we defined in the introductory chapter.

```
trait AllocRaw {
    fn alloc<T>(&self, object: T) -> *const T;
}
```

This will quickly prove to be inadequate and non-idiomatic. For starters, there is no way to report that allocation failed except for perhaps returning a null pointer. That is certainly a workable solution but is not going to feel idiomatic or ergonomic for how we want to use the API. Let's make a couple changes:

```
trait AllocRaw {
    fn alloc<T>(&self, object: T) -> Result<RawPtr<T>, AllocError>;
}
```

Now we're returning a `Result`, the failure side of which is an error type where we can distinguish between different allocation failure modes. This is often not that useful but working with `Result` is far more idiomatic Rust than checking a pointer for being null. We'll allow for distinguishing between Out Of Memory and an allocation request that for whatever reason is invalid.

```
#[derive(Copy, Clone, Debug, PartialEq)]
pub enum AllocError {
    /// Some attribute of the allocation, most likely the size requested,
    /// could not be fulfilled
    BadRequest,
    /// Out of memory - allocating the space failed
    OOM,
}
```

The second change is that instead of a `*const T` value in the success discriminant we'll wrap a pointer in a new struct: `RawPtr<T>`. This wrapper will amount to little more than containing a `std::ptr::NonNull` instance and some functions to access the pointer.

```
pub struct RawPtr<T: Sized> {
    ptr: NonNull<T>,
}
```

This'll be better to work with on the user-of-the-crate side.

It'll also make it easier to modify internals or even swap out entire implementations. This is a motivating factor for the design of this interface as we'll see as we continue to amend it to account for object headers now.

Object headers

The purpose of an object header is to provide the allocator, the language runtime and the garbage collector with information about the object that is needed at runtime. Typical data points that are stored might include:

- object size
- some kind of type identifier
- garbage collection information such as a mark flag

We want to create a flexible interface to a language while also ensuring that the interpreter will provide the information that the allocator and garbage collector in *this* crate need.

We'll define a trait for the user to implement.

```
pub trait AllocHeader: Sized {
    /// Associated type that identifies the allocated object type
    type TypeId: AllocTypeId;

    /// Create a new header for object type O
    fn new<O: AllocObject<Self::TypeId>>(size: u32, size_class: SizeClass,
mark: Mark) -> Self;

    /// Create a new header for an array type
    fn new_array(size: ArraySize, size_class: SizeClass, mark: Mark) -> Self;

    /// Set the Mark value to "marked"
    fn mark(&mut self);

    /// Get the current Mark value
    fn is_marked(&self) -> bool;

    /// Get the size class of the object
    fn size_class(&self) -> SizeClass;

    /// Get the size of the object in bytes
    fn size(&self) -> u32;

    /// Get the type of the object
    fn type_id(&self) -> Self::TypeId;
}
```

Now we have a bunch more questions to answer! Some of these trait methods are straightforward - `fn size(&self) -> u32` returns the object size; `mark()` and `is_marked()` must be GC related. Some are less obvious, such as `new_array()` which we'll cover at the end of this chapter.

But this struct references some more types that must be defined and explained.

Type identification

What follows is a set of design trade-offs made for the purposes of this book; there are many ways this could be implemented.

The types described next are all about sharing compile-time and runtime object type information between the allocator, the GC and the interpreter.

We ideally want to make it difficult for the user to make mistakes with this and leak undefined behavior. We would also prefer this to be a safe-Rust interface, while at the same time being flexible enough for the user to make interpreter-appropriate decisions about the header design.

First up, an object header implementation must define an associated type

```
pub trait AllocHeader: Sized {
    type TypeId: AllocTypeId;
}
```

where `AllocTypeId` is define simply as:

```
pub trait AllocTypeId: Copy + Clone {}
```

This means the interpreter is free to implement a type identifier type however it pleases, the only constraint is that it implements this trait.

Next, the definition of the header constructor,

```
pub trait AllocHeader: Sized {
    ...

    fn new<O: AllocObject<Self::TypeId>>(
        size: u32,
        size_class: SizeClass,
        mark: Mark
    ) -> Self;

    ...
}
```

refers to a type `O` that must implement `AllocObject` which in turn must refer to the common `AllocTypeId`. The generic type `O` is the object for which the header is being instantiated for.

And what is `AllocObject`? Simply:

```
pub trait AllocObject<T: AllocTypeId> {
    const TYPE_ID: T;
}
```

In summary, we have:

- `AllocHeader` : a trait that the header type must implement
- `AllocTypeId` : a trait that a type identifier must implement
- `AllocObject` : a trait that objects that can be allocated must implement

An example

Let's implement a couple of traits to make it more concrete.

The simplest form of type identifier is an enum. Each discriminant describes a type that the interpreter will use at runtime.

```
#[derive(PartialEq, Copy, Clone)]
enum MyTypeId {
    Number,
    String,
    Array,
}

impl AllocTypeId for MyTypeId {}
```

A hypothetical numeric type for our interpreter with the type identifier as associated constant:

```
struct BigNumber {
    value: i64
}

impl AllocObject<MyTypeId> for BigNumber {
    const TYPE_ID: MyTypeId = MyTypeId::Number;
}
```

And finally, here is a possible object header struct and the implementation of `AllocHeader::new()` :

```
struct MyHeader {
    size: u32,
    size_class: SizeClass,
    mark: Mark,
    type_id: MyTypeId,
}

impl AllocHeader for MyHeader {
    type TypeId = MyTypeId;

    fn new<O: AllocObject<Self::TypeId>>(
        size: u32,
        size_class: SizeClass,
        mark: Mark
    ) -> Self {
        MyHeader {
            size,
            size_class,
            mark,
            type_id: O::TYPE_ID,
        }
    }

    ...
}
```

These would all be defined and implemented in the interpreter and are not provided by the Sticky Immix crate, while all the functions in the trait `AllocHeader` are intended to be called internally by the allocator itself, not on the interpreter side.

The types `SizeClass` and `Mark` *are* provided by this crate and are enums.

The one drawback to this scheme is that it's possible to associate an incorrect type id constant with an object. This would result in objects being misidentified at runtime and accessed incorrectly, likely leading to panics.

Fortunately, this kind of trait implementation boilerplate is ideal for derive macros. Since the language side will be implementing these structs and traits, we'll defer until the relevant interpreter chapter to go over that.

Back to AllocRaw

Now that we have some object and header definitions and constraints, we need to apply them to the `AllocRaw` API. We can't allocate an object unless it implements `AllocObject` and has an associated constant that implements `AllocTypeId`. We also need to expand the interface with functions that the interpreter can use to reliably get the header for an object and the object for a header.

We will add an associated type to tie the allocator API to the header type and indirectly to the type identification that will be used.

```
pub trait AllocRaw {

    type Header: AllocHeader;

    ...
}
```

Then we can update the `alloc()` function definition to constrain the types that can be allocated to only those that implement the appropriate traits.

```
pub trait AllocRaw {
    ...

    fn alloc<T>(&self, object: T) -> Result<RawPtr<T>, AllocError>
    where
        T: AllocObject<<Self::Header as AllocHeader>::TypeId>;

    ...
}
```

We need the user and the garbage collector to be able to access the header, so we need a function that will return the header given an object pointer.

The garbage collector does not know about concrete types, it will need to be able to get the header without knowing the object type. It's likely that the interpreter will, at times, also not know the type at runtime.

Indeed, one of the functions of an object header is to, at runtime, given an object pointer, derive the type of the object.

The function signature therefore cannot refer to the type. That is, we can't write

```
pub trait AllocRaw {
    ...

    // looks good but won't work in all cases
    fn get_header<T>(object: RawPtr<T>) -> NonNull<Self::Header>
    where
        T: AllocObject<<Self::Header as AllocHeader>::TypeId>;

    ...
}
```

even though it seems this would be good and right. Instead this function will have to be much simpler:

```
pub trait AllocRaw {
    ...

    fn get_header(object: NonNull<()>) -> NonNull<Self::Header>;

    ...
}
```

We also need a function to get the object *from* the header:

```
pub trait AllocRaw {
    ...

    fn get_object(header: NonNull<Self::Header>) -> NonNull<()>;

    ...
}
```

These functions are not unsafe but they do return `NonNull` which implies that dereferencing the result should be considered unsafe - there is no protection against passing in garbage and getting garbage out.

Now we have an object allocation function, traits that constrain what can be allocated, allocation header definitions and functions for switching between an object and it's header.

There's one missing piece: we can allocate objects of type `T`, but such objects always have compile-time defined size. `T` is constrained to `Sized` types in the `RawPtr` definition. So how do we allocate dynamically sized objects, such as arrays?

Dynamically sized types

Since we can allocate objects of type `T`, and each `T` must derive `AllocObject` and have an associated const of type `AllocTypeId`, dynamically sized allocations must fit into this type identification scheme.

Allocating dynamically sized types, or in short, arrays, means there's some ambiguity about the type at compile time as far as the allocator is concerned:

- Are we allocating one object or an array of objects? If we're allocating an array of objects, we'll have to initialize them all. Perhaps we don't want to impose that overhead up front?
- If the allocator knows how many objects compose an array, do we want to bake fat pointers into the interface to carry that number around?

In the same way, then, that the underlying implementation of `std::vec::Vec` is backed

by an array of `u8`, we'll do the same. We shall define the return type of an array allocation to be of type `RawPtr<u8>` and the size requested to be in bytes. We'll leave it to the interpreter to build layers on top of this to handle the above questions.

As the definition of `AllocTypeId` is up to the interpreter, this crate can't know the type id of an array. Instead, we will require the interpreter to implement a function on the `AllocHeader` trait:

```
pub trait AllocHeader: Sized {
    ...

    fn new_array(size: ArraySize, size_class: SizeClass, mark: Mark) -> Self;

    ...
}
```

This function should return a new object header for an array of u8 with the appropriate type identifier.

We will also add a function to the `AllocRaw` trait for allocating arrays that returns the `RawPtr<u8>` type.

```
pub trait AllocRaw {
    ...

    fn alloc_array(&self, size_bytes: ArraySize) -> Result<RawPtr<u8>,
AllocError>;

    ...
}
```

Our complete `AllocRaw` trait definition now looks like this:

```
pub trait AllocRaw {
    /// An implementation of an object header type
    type Header: AllocHeader;

    /// Allocate a single object of type T.
    fn alloc<T>(&self, object: T) -> Result<RawPtr<T>, AllocError>
    where
        T: AllocObject<<Self::Header as AllocHeader>::TypeId>;

    /// Allocating an array allows the client to put anything in the
resulting data
    /// block but the type of the memory block will simply be 'Array'. No
other
    /// type information will be stored in the object header.
    /// This is just a special case of alloc<T>() for T=u8 but a count > 1 of
u8
    /// instances.  The caller is responsible for the content of the array.
    fn alloc_array(&self, size_bytes: ArraySize) -> Result<RawPtr<u8>,
AllocError>;

    /// Given a bare pointer to an object, return the expected header address
    fn get_header(object: NonNull<()>) -> NonNull<Self::Header>;

    /// Given a bare pointer to an object's header, return the expected
object address
    fn get_object(header: NonNull<Self::Header>) -> NonNull<()>;
}
```

In the next chapter we'll build out the `AllocRaw` trait implementation.

Implementing the Allocation API

In this final chapter of the allocation part of the book, we'll cover the `AllocRaw` trait implementation.

This trait is implemented on the `StickyImmixHeap` struct:

```
impl<H: AllocHeader> AllocRaw for StickyImmixHeap<H> {
    type Header = H;

    ...
}
```

Here the associated header type is provided as the generic type `H`, leaving it up to the interpreter to define.

Allocating objects

The first function to implement is `AllocRaw::alloc<T>()`. This function must:

- calculate how much space in bytes is required by the object and header
- allocate that space
- instantiate an object header and write it to the first bytes of the space
- copy the object itself to the remaining bytes of the space
- return a pointer to where the object lives in this space

Let's look at the implementation.

```
impl<H: AllocHeader> AllocRaw for StickyImmixHeap<H> {
    fn alloc<T>(&self, object: T) -> Result<RawPtr<T>, AllocError>
    where
        T: AllocObject<<Self::Header as AllocHeader>::TypeId>,
    {
        // calculate the total size of the object and it's header
        let header_size = size_of::<Self::Header>();
        let object_size = size_of::<T>();
        let total_size = header_size + object_size;

        // round the size to the next word boundary to keep objects aligned
and get the size class
        // TODO BUG? should this be done separately for header and object?
        //  If the base allocation address is where the header gets placed,
perhaps
        //  this breaks the double-word alignment object alignment desire?
        let alloc_size = alloc_size_of(total_size);
        let size_class = SizeClass::get_for_size(alloc_size)?;

        // attempt to allocate enough space for the header and the object
        let space = self.find_space(alloc_size, size_class)?;

        // instantiate an object header for type T, setting the mark bit to
"allocated"
        let header = Self::Header::new::<T>(object_size as ArraySize,
size_class, Mark::Allocated);

        // write the header into the front of the allocated space
        unsafe {
            write(space as *mut Self::Header, header);
        }

        // write the object into the allocated space after the header
        let object_space = unsafe { space.offset(header_size as isize) };
        unsafe {
            write(object_space as *mut T, object);
        }

        // return a pointer to the object in the allocated space
        Ok(RawPtr::new(object_space as *const T))
    }
}
```

This, hopefully, is easy enough to follow after the previous chapters -

- `self.find_space()` is the function described in the chapter Allocating into multiple blocks
- `Self::Header::new()` will be implemented by the interpreter
- `write(space as *mut Self::Header, header)` calls the std function `std::ptr::write`

Allocating arrays

We need a similar (but awkwardly different enough) implementation for array allocation. The key differences are that the type is fixed to a `u8` pointer and the array is initialized to zero bytes. It is up to the interpreter to write into the array itself.

```
impl<H: AllocHeader> AllocRaw for StickyImmixHeap<H> {
    fn alloc_array(&self, size_bytes: ArraySize) -> Result<RawPtr<u8>,
AllocError> {
        // calculate the total size of the array and it's header
        let header_size = size_of::<Self::Header>();
        let total_size = header_size + size_bytes as usize;

        // round the size to the next word boundary to keep objects aligned
and get the size class
        let alloc_size = alloc_size_of(total_size);
        let size_class = SizeClass::get_for_size(alloc_size)?;

        // attempt to allocate enough space for the header and the array
        let space = self.find_space(alloc_size, size_class)?;

        // instantiate an object header for an array, setting the mark bit to
"allocated"
        let header = Self::Header::new_array(size_bytes, size_class,
Mark::Allocated);

        // write the header into the front of the allocated space
        unsafe {
            write(space as *mut Self::Header, header);
        }

        // calculate where the array will begin after the header
        let array_space = unsafe { space.offset(header_size as isize) };

        // Initialize object_space to zero here.
        // If using the system allocator for any objects (SizeClass::Large,
for example),
        // the memory may already be zeroed.
        let array = unsafe { from_raw_parts_mut(array_space as *mut u8,
size_bytes as usize) };
        // The compiler should recognize this as optimizable
        for byte in array {
            *byte = 0;
        }

        // return a pointer to the array in the allocated space
        Ok(RawPtr::new(array_space as *const u8))
    }
}
```

Switching between header and object

As stated in the previous chapter, these functions are essentially pointer operations that do not dereference the pointers. Thus they are not unsafe to call, but the types they operate *on* should have a suitably unsafe API.

`NonNull` is the chosen parameter and return type and the pointer arithmetic for

obtaining the header from an object pointer of unknown type is shown below.

For our Immix implementation, since headers are placed immediately ahead of an object, we simply subtract the header size from the object pointer.

```
impl<H: AllocHeader> AllocRaw for StickyImmixHeap<H> {
    fn get_header(object: NonNull<()>) -> NonNull<Self::Header> {
        unsafe {
NonNull::new_unchecked(object.cast::<Self::Header>().as_ptr().offset(-1)) }
    }
}
```

Getting the object from a header is the reverse - adding the header size to the header pointer results in the object pointer:

```
impl<H: AllocHeader> AllocRaw for StickyImmixHeap<H> {
    fn get_object(header: NonNull<Self::Header>) -> NonNull<()> {
        unsafe {
NonNull::new_unchecked(header.as_ptr().offset(1).cast::<()>()) }
    }
}
```

Conclusion

Thus ends the first part of our Immix implementation. In the next part of the book we will jump over the fence to the interpreter and begin using the interfaces we've defined in this part.

An interpreter: Eval-rs

In this part of the book we'll dive into creating:

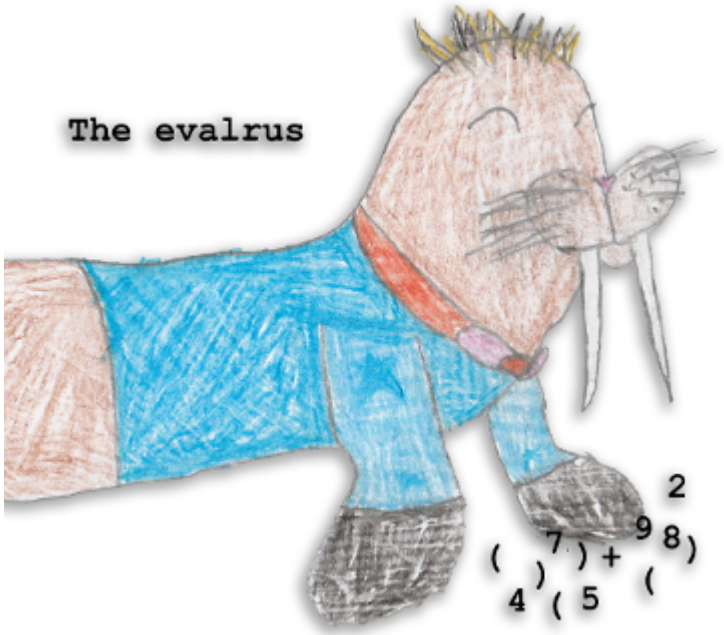
- a safe Rust layer on top of the Sticky Immix API of the previous part
- a compiler for a primitive s-expression syntax language
- a bytecode based virtual machine

So what kind of interpreter will we implement? This book is a guide to help you along your own journey and not not intended to provide an exhaustive language ecosystem. The direction we'll take is to support John McCarthy's classic s-expression based meta-circular evaluator[1].

Along the way we'll need to implement fundamental data types and structures from scratch upon our safe layer - symbols, pairs, arrays and dicts - with each chapter building upon the previous ones.

While this will not result in an exhaustive language implementation, you'll see that we *will* end up with all the building blocks for you to take it the rest of the way!

We shall name our interpreter "Eval-rs", for which we have an appropriate illustration generously provided by the author's then 10 year old daughter.



We'll begin by defining the safe abstration over the Sticky Immix interface. Then we'll put that to use in parsing s-expressions into a very simple data structure.

Once we've covered those basics, we'll build arrays and dicts and then use those in the compiler and virtual machine.

[1] These days this is cliché but that is substantially to our benefit. We're not trying to create yet another Lisp, rather the fact that there is a preexisting design of some elegance and historical interest is a convenience. For a practical, accessible introduction to the topic, do see Paul Graham's The Roots of Lisp

Allocating objects and dereferencing safely

In this chapter we'll build some safe Rust abstractions over the allocation API defined in the Sticky Immix crate.

Let's first recall this interface:

```
pub trait AllocRaw {
    /// An implementation of an object header type
    type Header: AllocHeader;

    /// Allocate a single object of type T.
    fn alloc<T>(&self, object: T) -> Result<RawPtr<T>, AllocError>
    where
        T: AllocObject<<Self::Header as AllocHeader>::TypeId>;

    /// Allocating an array allows the client to put anything in the
resulting data
    /// block but the type of the memory block will simply be 'Array'. No
other
    /// type information will be stored in the object header.
    /// This is just a special case of alloc<T>() for T=u8 but a count > 1 of
u8
    /// instances.  The caller is responsible for the content of the array.
    fn alloc_array(&self, size_bytes: ArraySize) -> Result<RawPtr<u8>,
AllocError>;

    /// Given a bare pointer to an object, return the expected header address
    fn get_header(object: NonNull<()>) -> NonNull<Self::Header>;

    /// Given a bare pointer to an object's header, return the expected
object address
    fn get_object(header: NonNull<Self::Header>) -> NonNull<()>;
}
```

These are the functions we'll be calling. When we allocate an object, we'll get back a `RawPtr<T>` which has no safe way to dereference it. This is impractical, we very much do not want to wrap every dereferencing in `unsafe { ... }`. We'll need a layer over `RawPtr<T>` where we can guarantee safe dereferencing.

Pointers

In safe Rust, mutable (`&mut`) and immutable (`&`) references are passed around to access objects. These reference types are compile-time constrained pointers where the constraints are

1. the mutability of the access
2. the lifetime of the access

For our layer over `RawPtr<T>` we'll have to consider both these constraints.

Mutability

This constraint is concerned with shared access to an object. In other words, it cares

about how many pointers there are to an object at any time and whether they allow mutable or immutable access.

The short of it is:

- Either only one `&mut` reference may be held in a scope
- Or many `&` immutable references may be held in a scope

The compiler must be able to determine that a `&mut` reference is the only live reference in it's scope that points at an object in order for mutable access to that object to be safe of data races.

In a runtime memory managed language such as the interpreter we are building, we will not have compile time knowledge of shared access to objects. We won't know at compile time how many pointers to an object we may have at any time. This is the normal state of things in languages such as Python, Ruby or Javascript.

This means that we can't allow `&mut` references in our safe layer at all!

If we're restricted to `&` immutable references everywhere, that then means we must apply the interior mutability pattern everywhere in our design in order to comply with the laws of safe Rust.

Lifetime

The second aspect to references is their lifetime. This concerns the duration of the reference, from inception until it goes out of scope.

The key concept to think about now is "scope."

In an interpreted language there are two major operations on the objects in memory:

```
fn run_mutator() {
    parse_source_code();
    compile();
    execute_bytecode();
}
```

and

```
fn run_garbage_collection() {
   trace_objects();
   free_dead_objects();
}
```

A few paragraphs earlier we determined that we can't have `&mut` references to objects in our interpreter.

By extension, we can't safely hold a mutable reference to the entire heap as a data structure.

Except, that is exactly what garbage collection requires. The nature of garbage collection is that it views the entire heap as a single data structure in it's own right that it needs to traverse and modify. It wants the heap to be `&mut`.

Consider, especially, that some garbage collectors *move* objects, so that pointers to moved objects, wherever they may be, must be modified by the garbage collector without breaking the mutator! The garbage collector must be able to reliably discover *every single pointer to moved objects* to avoid leaving invalid pointers scattered around[1].

Thus we have two mutually exclusive interface requirements, one that must only hold `&` object references and applies *interior* mutability to the heap and the other that wants the whole heap to be `&mut`.

For this part of the book, we'll focus on the use of the allocator and save garbage collection for a later part.

This mutual exclusivity constraint on the allocator results in the statements:

- When garbage collection is running, it is not safe to run the mutator[2]
- When garbage collection is not running, it is safe to run the mutator

Thus our abstraction must encapsulate a concept of a time when "it is safe to run the mutator" and since we're working with safe Rust, this must be a compile time concept.

Scopes and lifetimes are perfect for this abstraction. What we'll need is some way to define a lifetime (that is, a scope) within which access to the heap by the mutator is safe.

Some pointer types

First, let's define a simple pointer type that can wrap an allocated type `T` in a lifetime:

```
pub struct ScopedPtr<'guard, T: Sized> {
    value: &'guard T,
}
```

This type will implement `Clone`, `Copy` and `Deref` - it can be passed around freely within the scope and safely dereferenced.

As you can see we have a lifetime `'guard` that we'll use to restrict the scope in which this pointer can be accessed. We need a mechanism to restrict this scope.

The guard pattern is what we'll use, if the hint wasn't strong enough.

We'll construct some types that ensure that safe pointers such as `ScopedPtr<T>`, and access to the heap at in any way, are mediated by an instance of a guard type that can provide access.

We will end up passing a reference to the guard instance around everywhere. In most cases we won't care about the instance type itself so much as the lifetime that it carries with it. As such, we'll define a trait for this type to implement that so that we can refer to the guard instance by this trait rather than having to know the concrete type. This'll also allow other types to proxy the main scope-guarding instance.

```
pub trait MutatorScope {}
```

You may have noticed that we've jumped from `RawPtr<T>` to `ScopedPtr<T>` with seemingly nothing to bridge the gap. How do we *get* a `ScopedPtr<T>`?

We'll create a wrapper around `RawPtr<T>` that will complete the picture. This wrapper type is what will hold pointers at rest inside any data structures.

```
#[derive(Clone)]
pub struct CellPtr<T: Sized> {
    inner: Cell<RawPtr<T>>,
}
```

This is straightforwardly a `RawPtr<T>` in a `Cell` to allow for modifying the pointer. We won't allow dereferencing from this type either though.

Remember that dereferencing a heap object pointer is only safe when we are in the right scope? We need to create a `ScopedPtr<T>` *from* a `CellPtr<T>` to be able to use it.

First we'll add a helper function to `RawPtr<T>` in our interpreter crate so we can safely dereference a `RawPtr<T>`. This code says that, given an instance of a `MutatorScope`-implementing type, give me back a reference type with the same lifetime as the guard that I can safely use. Since the `_guard` parameter is never used except to define a lifetime, it should be optimized out by the compiler!

```
pub trait ScopedRef<T> {
    fn scoped_ref<'scope>(&self, guard: &'scope dyn MutatorScope) -> &'scope
T;
}

impl<T> ScopedRef<T> for RawPtr<T> {
    fn scoped_ref<'scope>(&self, _guard: &'scope dyn MutatorScope) -> &'scope
T {
        unsafe { &*self.as_ptr() }
    }
}
```

We'll use this in our `CellPtr<T>` to obtain a `ScopedPtr<T>`:

```
impl<T: Sized> CellPtr<T> {
    pub fn get<'guard>(&self, guard: &'guard dyn MutatorScope) ->
ScopedPtr<'guard, T> {
        ScopedPtr::new(guard, self.inner.get().scoped_ref(guard))
    }
}
```

Thus, anywhere (structs, enums) that needs to store a pointer to something on the heap will use `CellPtr<T>` and any code that accesses these pointers during the scope-guarded mutator code will obtain `ScopedPtr<T>` instances that can be safely dereferenced.

The heap and the mutator

The next question is: where do we get an instance of `MutatorScope` from?

The lifetime of an instance of a `MutatorScope` will define the lifetime of any safe object accesses. By following the guard pattern, we will find we have:

- a heap struct that contains an instance of the Sticky Immix heap
- a guard struct that proxies the heap struct for the duration of a scope
- a mechanism to enforce the scope limit

A heap struct

Let's make a type alias for the Sticky Immix heap so we aren't referring to it as such throughout the interpreter:

```
pub type HeapStorage = StickyImmixHeap<ObjectHeader>;
```

The let's put that into a heap struct, along with any other interpreter-global storage:

```
struct Heap {
    heap: HeapStorage,
    syms: SymbolMap,
}
```

We'll discuss the `SymbolMap` type in the next chapter.

Now, since we've wrapped the Sticky Immix heap in our own `Heap` struct, we'll need to `impl` an `alloc()` method to proxy the Sticky Immix allocation function.

```
impl Heap {
    fn alloc<T>(&self, object: T) -> Result<RawPtr<T>, RuntimeError>
    where
        T: AllocObject<TypeList>,
    {
        Ok(self.heap.alloc(object)?)
    }
}
```

A couple things to note about this function:

- It returns `RuntimeError` in the error case, this type converts `From` the Sticky Immix crate's error type.
- The `where` constraint is similar to that of `AllocRaw::alloc()` but in now we have a concrete `TypeList` type to bind to. We'll look at `TypeList` in the next chapter along with `SymbolMap`.

A guard struct

This next struct will be used as a scope-limited proxy for the `Heap` struct with one major difference: function return types will no longer be `RawPtr<T>` but `ScopedPtr<T>`.

```
pub struct MutatorView<'memory> {
    heap: &'memory Heap,
}
```

Here in this struct definition, it becomes clear that all we are doing is borrowing the `Heap` instance for a limited lifetime. Thus, the lifetime of the `MutatorView` instance *will be* the lifetime that all safe object access is constrained to.

A look at the `alloc()` function now:

```
impl<'memory> MutatorView<'memory> {
    pub fn alloc<T>(&self, object: T) -> Result<ScopedPtr<'_, T>,
RuntimeError>
    where
        T: AllocObject<TypeList>,
    {
        Ok(ScopedPtr::new(
            self,
            self.heap.alloc(object)?.scoped_ref(self),
        ))
    }
}
```

Very similar to `Heap::alloc()` but the return type is now a `ScopedPtr<T>` whose lifetime is the same as the `MutatorView` instance.

Enforcing a scope limit

We now have a `Heap` and a guard, `MutatorView`, but we want one more thing: to prevent an instance of `MutatorView` from being returned from anywhere - that is, enforcing a scope within which an instance of `MutatorView` will live and die. This will make it easier to separate mutator operations and garbage collection operations.

First we'll apply a constraint on how a mutator *gains* heap access: through a trait.

```
pub trait Mutator: Sized {
    type Input;
    type Output;

    fn run(&self, mem: &MutatorView, input: Self::Input) ->
Result<Self::Output, RuntimeError>;

    // TODO
    // function to return iterator that iterates over roots
}
```

If a piece of code wants to access the heap, it *must* implement this trait!

Secondly, we'll apply another wrapper struct, this time to the `Heap` type. This is so that we can borrow the `heap` member instance.

```
pub struct Memory {
    heap: Heap,
}
```

This `Memory` struct and the `Mutator` trait are now tied together with a function:

```
impl Memory {
    pub fn mutate<M: Mutator>(&self, m: &M, input: M::Input) ->
Result<M::Output, RuntimeError> {
        let mut guard = MutatorView::new(self);
        m.run(&mut guard, input)
    }

}
```

The key to the scope limitation mechanism is that this `mutate` function is the only way to gain access to the heap. It creates an instance of `MutatorView` that goes out of scope at the end of the function and thus can't leak outside of the call stack.

An example

Let's construct a simple example to demonstrate these many parts. This will omit defining

a `TypeId` and any other types that we didn't discuss above.

```
struct Stack {}

impl Stack {
    fn say_hello(&self) {
        println!("I'm the stack!");
    }
}

struct Roots {
    stack: CellPtr<Stack>
}

impl Roots {
    fn new(stack: ScopedPtr<'_, Stack>) -> Roots {
        Roots {
            stack: CellPtr::new_with(stack)
        }
    }
}

struct Interpreter {}

impl Mutator for Interpreter {
    type Input: ();
    type Output: Roots;

    fn run(&self, mem: &MutatorView, input: Self::Input) ->
Result<Self::Output, RuntimeError> {
        let stack = mem.alloc(Stack {})?;   // returns a ScopedPtr<'_, Stack>
        stack.say_hello();

        let roots = Roots::new(stack);

        let stack_ptr = roots.stack.get(mem);  // returns a ScopedPtr<'_,
Stack>
        stack_ptr.say_hello();

        Ok(roots)
    }
}

fn main() {
    ...
    let interp = Interpreter {};

    let result = memory.mutate(&interp, ());

    let roots = result.unwrap();

    // no way to do this - compile error
    let stack = roots.stack.get();
    ...
}
```

In this simple, contrived example, we instantiated a `Stack` on the heap. An instance of `Roots` is created on the native stack and given a pointer to the `Stack` instance. The mutator returns the `Roots` object, which continues to hold a pointer to a heap object. However, outside of the `run()` function, the `stack` member can't be safely accesed.

Up next: using this framework to implement parsing!

---

[1] This is the topic of discussion in Felix Klock's series GC and Rust which is recommended reading.

[2] while this distinction exists at the interface level, in reality there are multiple phases in garbage collection and not all of them require exclusive access to the heap. This is an advanced topic that we won't bring into consideration yet.

Tagged pointers and object headers

Since our virtual machine will support a dynamic language where the compiler does no type checking, all the type information will be managed at runtime.

In the previous chapter, we introduced a pointer type `ScopedPtr<T>`. This pointer type has compile time knowledge of the type it is pointing at.

We need an alternative to `ScopedPtr<T>` that can represent all the runtime-visible types so they can be resolved *at* runtime.

As we'll see, carrying around type information or looking it up in the header on every access will be inefficient space and performance-wise.

We'll implement a common optimization: tagged pointers.

Runtime type identification

The object header can always give us the type id for an object, given a pointer to the object. However, it requires us to do some arithmetic on the pointer to get the location of the type identifier, then dereference the pointer to get the type id value. This dereference can be expensive if the object being pointed at is not in the CPU cache. Since getting an object type is a very common operation in a dynamic language, these lookups become expensive, time-wise.

Rust itself doesn't have runtime type *identification* but does have runtime dispatch through trait objects. In this scheme a pointer consists of two words: the pointer to the object itself and a second pointer to the vtable where the concrete object type's methods

can be looked up. The generic name for this form of pointer is a *fat* pointer.

We could easily use a fat pointer type for runtime type identification in our interpreter. Each pointer could carry with it an additional word with the type id in it, or we could even just use trait objects!

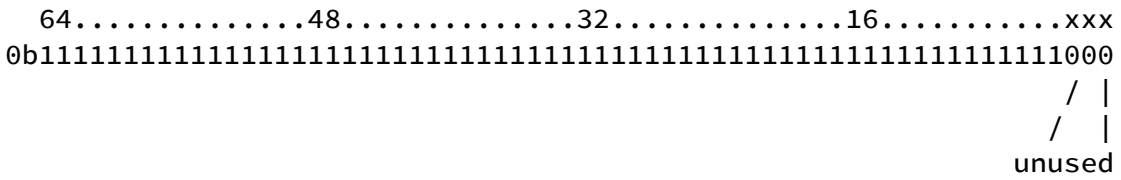
A dynamically typed language will manage many pointers that must be type identified at runtime. Carrying around an extra word per pointer is expensive, space-wise, however.

Tagged pointers

Many runtimes implement tagged pointers to avoid the space overhead, while partially improving the time overhead of the header type-id lookup.

In a pointer to any object on the heap, the least most significant bits turn out to always be zero due to word or double-word alignment.

On a 64 bit platform, a pointer is a 64 bit word. Since objects are at least word-aligned, a pointer is always be a multiple of 8 and the 3 least significant bits are always 0. On 32 bit platforms, the 2 least significant bits are always 0.



When dereferencing a pointer, these bits must always be zero. But we *can* use them in pointers at rest to store a limited type identifier! We'll limit ourselves to 2 bits of type identifier so as to not complicate our code in distinguishing between 32 and 64 bit platforms[1].

Given we'll only have 4 possible types we can id directly from a pointer, we'll still need to fall back on the object header for types that don't fit into this range.

Encoding this in Rust

Flipping bits on a pointer directly definitely constitutes a big Unsafe. We'll need to make a tagged pointer type that will fundamentally be `unsafe` because it won't be safe to dereference it. Then we'll need a safe abstraction over that type to make it safe to dereference.

But first we need to understand the object header and how we get an object's type from

it.

The object header

We introduced the object header traits in the earlier chapter Defining the allocation API. The chapter explained how the object header is the responsibility of the interpreter to implement.

Now that we need to implement type identification, we need the object header.

The allocator API requires that the type identifier implement the `AllocTypeId` trait. We'll use an `enum` to identify for all our runtime types:

```
#[repr(u16)]
#[derive(Debug, Copy, Clone, PartialEq)]
pub enum TypeList {
    ArrayBackingBytes,
    ArrayOpcode,
    ArrayU8,
    ArrayU16,
    ArrayU32,
    ByteCode,
    CallFrameList,
    Dict,
    Function,
    InstructionStream,
    List,
    NumberObject,
    Pair,
    Partial,
    Symbol,
    Text,
    Thread,
    Upvalue,
}

// Mark this as a Stickyimmix type-identifier type
impl AllocTypeId for TypeList {}
```

Given that the allocator API requires every object that can be allocated to have an associated type id `const`, this `enum` represents every type that can be allocated and that we will go on to describe in this book.

It is a member of the `ObjectHeader` struct along with a few other members that our Immix implementation requires:

```
pub struct ObjectHeader {
    mark: Mark,
    size_class: SizeClass,
    type_id: TypeList,
    size_bytes: u32,
}
```

The rest of the header members will be the topic of the later garbage collection part of the book.

A safe pointer abstraction

A type that can represent one of multiple types at runtime is obviously the `enum`. We can wrap possible `ScopedPtr<T>` types like so:

```
#[derive(Copy, Clone)]
pub enum Value<'guard> {
    ArrayU8(ScopedPtr<'guard, ArrayU8>),
    ArrayU16(ScopedPtr<'guard, ArrayU16>),
    ArrayU32(ScopedPtr<'guard, ArrayU32>),
    Dict(ScopedPtr<'guard, Dict>),
    Function(ScopedPtr<'guard, Function>),
    List(ScopedPtr<'guard, List>),
    Nil,
    Number(isize),
    NumberObject(ScopedPtr<'guard, NumberObject>),
    Pair(ScopedPtr<'guard, Pair>),
    Partial(ScopedPtr<'guard, Partial>),
    Symbol(ScopedPtr<'guard, Symbol>),
    Text(ScopedPtr<'guard, Text>),
    Upvalue(ScopedPtr<'guard, Upvalue>),
}
```

Note that this definition does *not* include all the same types that were listed above in `TypeList`. Only the types that can be passed dynamically at runtime need to be represented here. The types not included here are always referenced directly by `ScopedPtr<T>` and are therefore known types at compile and run time.

You probably also noticed that `Value` *is* the fat pointer we discussed earlier. It is composed of a set of `ScopedPtr<T>`s, each of which should only require a single word, and an `enum` discriminant integer, which will also, due to alignment, require a word.

This `enum`, since it wraps `ScopedPtr<T>` and has the same requirement for an explicit lifetime, is Safe To Dereference.

As this type occupies the same space as a fat pointer, it isn't the type we want for storing pointers at rest, though. For that type, let's look at the compact tagged pointer type now.

What lies beneath

Below we have a `union` type, making this an unsafe representation of a pointer. The `tag` value will be constrained to the values 0, 1, 2 or 3, which will determine which of the next four possible members should be accessed. Members will have to be bit-masked to access their correct values.

```
#[derive(Copy, Clone)]
pub union TaggedPtr {
    tag: usize,
    number: isize,
    symbol: NonNull<Symbol>,
    pair: NonNull<Pair>,
    object: NonNull<()>,
}
```

As you can see, we've allocated a tag for a `Symbol` type, a `Pair` type and one for a numeric type. The fourth member indicates an object whose type must be determined from the type id in the object header.

> ***Note:*** Making space for an inline integer is a common use of a tag. It means any integer arithmetic that fits within the available bits will not require memory lookups into the heap to retrieve operands. In our case we've defined the numeric type as an `isize`. Since the 2 least significant bits are used for the tag, we will have to right-shift the value by 2 to extract the correct integer value. We'll go into this implementation in more depth in a later chapter.

The tags and masks are defined as:

```
const TAG_MASK: usize = 0x3;
pub const TAG_SYMBOL: usize = 0x0;
pub const TAG_PAIR: usize = 0x1;
pub const TAG_OBJECT: usize = 0x2;
pub const TAG_NUMBER: usize = 0x3;
const PTR_MASK: usize = !0x3;
```

Thus you can see from the choice of embedded tag values, we've optimized for fast identification of `Pair`s and `Symbol`s and integer math. If we decide to, it will be easy to switch to other types to represent in the 2 tag bits.

Connecting into the allocation API

Translating between `Value` and `TaggedPtr` will be made easier by creating an intermediate type that represents all types as an `enum` but doesn't require a valid lifetime. This type will be useful because it is most closely ergonomic with the allocator

API and the object header type information.

```
#[derive(Copy, Clone)]
pub enum FatPtr {
    ArrayU8(RawPtr<ArrayU8>),
    ArrayU16(RawPtr<ArrayU16>),
    ArrayU32(RawPtr<ArrayU32>),
    Dict(RawPtr<Dict>),
    Function(RawPtr<Function>),
    List(RawPtr<List>),
    Nil,
    Number(isize),
    NumberObject(RawPtr<NumberObject>),
    Pair(RawPtr<Pair>),
    Partial(RawPtr<Partial>),
    Symbol(RawPtr<Symbol>),
    Text(RawPtr<Text>),
    Upvalue(RawPtr<Upvalue>),
}
```

We'll extend `Heap` (see previous chapter) with a method to return a tagged pointer on request:

```
impl Heap {
    fn alloc_tagged<T>(&self, object: T) -> Result<TaggedPtr, RuntimeError>
    where
        FatPtr: From<RawPtr<T>>,
        T: AllocObject<TypeList>,
    {
        Ok(TaggedPtr::from(FatPtr::from(self.heap.alloc(object)?)))
    }
}
```

In this method it's clear that we implemented `From<T>` to convert between pointer types. Next we'll look at how these conversions are implemented.

Type conversions

We have three pointer types: `Value`, `FatPtr` and `TaggedPtr`, each which has a distinct flavor. We need to be able to convert from one to the other:

```
TaggedPtr <-> FatPtr -> Value
```

FatPtr to Value

We can implement `From<FatPtr>` for `TaggedPtr` and `Value` to convert to the final two possible pointer representations. Well, not exactly - the function signature

```
impl From<FatPtr> for Value<'guard> {
    fn from(ptr: FatPtr) -> Value<'guard> { ... }
}
```

is not able to define the `'guard` lifetime, so we have to implement a similar method that can:

```
impl FatPtr {
    pub fn as_value<'guard>(&self, guard: &'guard dyn MutatorScope) ->
Value<'guard> {
        match self {
            FatPtr::ArrayU8(raw_ptr) => {
                Value::ArrayU8(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::ArrayU16(raw_ptr) => {
                Value::ArrayU16(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::ArrayU32(raw_ptr) => {
                Value::ArrayU32(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::Dict(raw_ptr) => Value::Dict(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard))),
            FatPtr::Function(raw_ptr) => {
                Value::Function(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::List(raw_ptr) => Value::List(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard))),
            FatPtr::Nil => Value::Nil,
            FatPtr::Number(num) => Value::Number(*num),
            FatPtr::NumberObject(raw_ptr) => {
                Value::NumberObject(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::Pair(raw_ptr) => Value::Pair(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard))),
            FatPtr::Partial(raw_ptr) => {
                Value::Partial(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::Symbol(raw_ptr) => {
                Value::Symbol(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
            FatPtr::Text(raw_ptr) => Value::Text(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard))),
            FatPtr::Upvalue(raw_ptr) => {
                Value::Upvalue(ScopedPtr::new(guard,
raw_ptr.scoped_ref(guard)))
            }
        }
    }
}
```

FatPtr to TaggedPtr

For converting down to a single-word `TaggedPtr` type we will introduce a helper trait and

methods to work with tag values and `RawPtr<T>` types from the allocator:

```
pub trait Tagged<T> {
    fn tag(self, tag: usize) -> NonNull<T>;
    fn untag(from: NonNull<T>) -> RawPtr<T>;
}

impl<T> Tagged<T> for RawPtr<T> {
    fn tag(self, tag: usize) -> NonNull<T> {
        unsafe { NonNull::new_unchecked((self.as_word() | tag) as *mut T) }
    }

    fn untag(from: NonNull<T>) -> RawPtr<T> {
        RawPtr::new((from.as_ptr() as usize & PTR_MASK) as *const T)
    }
}
```

This will help convert from `RawPtr<T>` values in `FatPtr` to the `NonNull<T>` based `TaggedPtr` discriminants.

Because `TaggedPtr` is a `union` type and because it has to apply the appropriate tag value inside the pointer itself, we can't work with it as ergnomically as an `enum`. We'll create some more helper functions for instantiating `TaggedPtr`s appropriately.

Remember that for storing an integer in the pointer we have to left-shift it 2 bits to allow for the tag. We'll apply proper range checking in a later chapter.

```
impl TaggedPtr {
    pub fn nil() -> TaggedPtr {
        TaggedPtr { tag: 0 }
    }

    pub fn number(value: isize) -> TaggedPtr {
        TaggedPtr {
            number: (((value as usize) << 2) | TAG_NUMBER) as isize,
        }
    }

    pub fn symbol(ptr: RawPtr<Symbol>) -> TaggedPtr {
        TaggedPtr {
            symbol: ptr.tag(TAG_SYMBOL),
        }
    }

    fn pair(ptr: RawPtr<Pair>) -> TaggedPtr {
        TaggedPtr {
            pair: ptr.tag(TAG_PAIR),
        }
    }
}
```

Finally, we can use the above methods to implement `From<FatPtr` for `TaggedPtr`:

```
impl From<FatPtr> for TaggedPtr {
    fn from(ptr: FatPtr) -> TaggedPtr {
        match ptr {
            FatPtr::ArrayU8(raw) => TaggedPtr::object(raw),
            FatPtr::ArrayU16(raw) => TaggedPtr::object(raw),
            FatPtr::ArrayU32(raw) => TaggedPtr::object(raw),
            FatPtr::Dict(raw) => TaggedPtr::object(raw),
            FatPtr::Function(raw) => TaggedPtr::object(raw),
            FatPtr::List(raw) => TaggedPtr::object(raw),
            FatPtr::Nil => TaggedPtr::nil(),
            FatPtr::Number(value) => TaggedPtr::number(value),
            FatPtr::NumberObject(raw) => TaggedPtr::object(raw),
            FatPtr::Pair(raw) => TaggedPtr::pair(raw),
            FatPtr::Partial(raw) => TaggedPtr::object(raw),
            FatPtr::Text(raw) => TaggedPtr::object(raw),
            FatPtr::Symbol(raw) => TaggedPtr::symbol(raw),
            FatPtr::Upvalue(raw) => TaggedPtr::object(raw),
        }
    }
}
```

TaggedPtr to FatPtr

To convert from a `TaggedPtr` to the intermediate type is implemented in two parts: identifying object types from the tag; identifying object types from the header where the tag is insufficient.

Part the first, which requires `unsafe` due to accessing a `union` type and dereferencing the object header for the `TAG_OBJECT` discriminant:

```
impl From<TaggedPtr> for FatPtr {
    fn from(ptr: TaggedPtr) -> FatPtr {
        ptr.into_fat_ptr()
    }
}

impl TaggedPtr {
    fn into_fat_ptr(&self) -> FatPtr {
        unsafe {
            if self.tag == 0 {
                FatPtr::Nil
            } else {
                match get_tag(self.tag) {
                    TAG_NUMBER => FatPtr::Number(self.number >> 2),
                    TAG_SYMBOL => FatPtr::Symbol(RawPtr::untag(self.symbol)),
                    TAG_PAIR => FatPtr::Pair(RawPtr::untag(self.pair)),

                    TAG_OBJECT => {
                        let untyped_object_ptr =
RawPtr::untag(self.object).as_untyped();
                        let header_ptr =
HeapStorage::get_header(untyped_object_ptr);

                        header_ptr.as_ref().get_object_fatptr()
                    }

                    _ => panic!("Invalid TaggedPtr type tag!"),
                }
            }
        }
    }
}
```

And part two, the object header method `get_object_fatptr()` as seen in the code above:

```
impl ObjectHeader {
    pub unsafe fn get_object_fatptr(&self) -> FatPtr {
        let ptr_to_self = self.non_null_ptr();
        let object_addr = HeapStorage::get_object(ptr_to_self);

        match self.type_id {
            TypeList::ArrayU8 =>
FatPtr::ArrayU8(RawPtr::untag(object_addr.cast::<ArrayU8>())),
            TypeList::ArrayU16 =>
FatPtr::ArrayU16(RawPtr::untag(object_addr.cast::<ArrayU16>())),
            TypeList::ArrayU32 =>
FatPtr::ArrayU32(RawPtr::untag(object_addr.cast::<ArrayU32>())),
            TypeList::Dict =>
FatPtr::Dict(RawPtr::untag(object_addr.cast::<Dict>())),
            TypeList::Function =>
FatPtr::Function(RawPtr::untag(object_addr.cast::<Function>())),
            TypeList::List =>
FatPtr::List(RawPtr::untag(object_addr.cast::<List>())),
            TypeList::NumberObject => {

FatPtr::NumberObject(RawPtr::untag(object_addr.cast::<NumberObject>()))
            }
            TypeList::Pair =>
FatPtr::Pair(RawPtr::untag(object_addr.cast::<Pair>())),
            TypeList::Partial =>
FatPtr::Partial(RawPtr::untag(object_addr.cast::<Partial>())),
            TypeList::Symbol =>
FatPtr::Symbol(RawPtr::untag(object_addr.cast::<Symbol>())),
            TypeList::Text =>
FatPtr::Text(RawPtr::untag(object_addr.cast::<Text>())),
            TypeList::Upvalue =>
FatPtr::Upvalue(RawPtr::untag(object_addr.cast::<Upvalue>())),

            // Other types not represented by FatPtr are an error to id here
            _ => panic!("Invalid ObjectHeader type tag {:?}!", self.type_id),
        }
    }
}
```

This method contains no unsafe code and yet we've declared it unsafe!

Manipulating pointer types is not unsafe in of itself, only dereferencing them is unsafe and we are not dereferencing them here.

While we have the safety rails of the `enum` types to prevent *invalid* types from being returned, we could easily mismatch a `TypeList` value with an incorrect `FatPtr` value and return an *incorrect* type. Additionally we could forget to untag a pointer, leaving it as an invalid pointer value.

These possible mistakes could cause undefined behavior and quite likely crash the interpreter.

The compiler will not catch these cases and so this is an area for critical scrutiny of

correctness! Hence the method is marked unsafe to draw attention.

Using tagged pointers in data structures

Finally, we need to see how to use these types in data structures that we'll create.

In the previous chapter, we defined a `CellPtr` type that wrapped a `RawPtr<T>` in a `Cell<T>` so that data structures can contain mutable pointers to other objects. Similarly, we'll want something to wrap tagged pointers:

```
#[derive(Clone)]
pub struct TaggedCellPtr {
    inner: Cell<TaggedPtr>,
}
```

We'll also wrap `Value` in a type `TaggedScopedPtr` that we'll use similarly to `ScopedPtr<T>`.

```
#[derive(Copy, Clone)]
pub struct TaggedScopedPtr<'guard> {
    ptr: TaggedPtr,
    value: Value<'guard>,
}
```

This `TaggedScopedPtr` carries an instance of `TaggedPtr` *and* a `Value`. This tradeoff means that while this type has three words to heft around, the `TaggedPtr` member can be quickly accessed for copying into a `TaggedCellPtr` without needing to down-convert from `Value`.

The type is only suitable for handling pointers that actively need to be dereferenced due to it's size.

> ***Note:*** Redundancy: TaggedScopedPtr and Value are almost identical in requirement and functionality. TODO: consider merging into one type. See issue https://github.com/rust-hosted-langs/book/issues/30

A `TaggedScopedPtr` can be obtained by:

- calling `TaggedCellPtr::get()`
- or the `MutatorView::alloc_tagged()` method

The `get()` method on `TaggedCellPtr` returns a `TaggedScopedPtr`:

```
impl TaggedCellPtr {
    pub fn get<'guard>(&self, guard: &'guard dyn MutatorScope) ->
TaggedScopedPtr<'guard> {
        TaggedScopedPtr::new(guard, self.inner.get())
    }
}
```

The `MutatorView` method to allocate a new object and get back a tagged pointer (a `TaggedScopedPtr`) looks simply like this:

```
impl MutatorView {
    pub fn alloc_tagged<T>(&self, object: T) -> Result<TaggedScopedPtr<'_>,
RuntimeError>
    where
        FatPtr: From<RawPtr<T>>,
        T: AllocObject<TypeList>,
    {
        Ok(TaggedScopedPtr::new(self, self.heap.alloc_tagged(object)?))
    }
}
```

Quick recap

In summary, what we created here was a set of pointer types:

- types suitable for storing a pointer at rest - `TaggedPtr` and `TaggedCellPtr`
- types suitable for dereferencing a pointer - `Value` and `TaggedScopedPtr`
- a type suitable for intermediating between the two - `FatPtr` - that the heap allocation interface can return

We now have the basic pieces to start defining data structures for our interpreter, so that is what we shall do next!

---

[1] There are other pointer tagging schemes, notably the use of "spare" NaN bit patterns in 64 bit floating point values. Further, *which* types are best represented by the tag bits is highly language dependent. Some languages use them for garbage collection information while others may use them for still other types hidden from the language user. In the interest of clarity, we'll stick to a simple scheme.

Symbols and Pairs

To bootstrap our compiler, we'll parse s-expressions into `Symbol` ad `Pair` types, where a

`Pair` is essentially a Lisp cons cell.

The definition of `Symbol` is just the raw components of a `&str`:

```
#[derive(Copy, Clone)]
pub struct Symbol {
    name_ptr: *const u8,
    name_len: usize,
}
```

Why this is how `Symbol` is defined and how we handle these raw components will be covered in just a bit. First though, we'll delve into the `Pair` type.

Pairs of pointers

The definition of `Pair` is

```
#[derive(Clone)]
pub struct Pair {
    pub first: TaggedCellPtr,
    pub second: TaggedCellPtr,
    // Possible source code positions of the first and second values
    pub first_pos: Cell<Option<SourcePos>>,
    pub second_pos: Cell<Option<SourcePos>>,
}
```

The type of `first` and `second` is `TaggedCellPtr`, as seen in the previous chapter. This pointer type can point at any dynamic type. By the end of this chapter we'll be able to build a nested linked list of `Pair`s and `Symbol`s.

Since this structure will be used for parsing and compiling, the `Pair struct` has a couple of extra members that optionally describe the source code line and character number of the values pointed at by `first` and `second`. These will be useful for reporting error messages. We'll come back to these in the chapter on parsing.

To instantiate a `Pair` function with `first` and `second` set to nil, let's create a `new()` function:

```
impl Pair {
    pub fn new() -> Pair {
        Pair {
            first: TaggedCellPtr::new_nil(),
            second: TaggedCellPtr::new_nil(),
            first_pos: Cell::new(None),
            second_pos: Cell::new(None),
        }
    }
}
```

That function, as it's not being allocated into the heap, doesn't require the lifetime guard. Let's look at a more interesting function: `cons()`, which assigns a value to `first` and `second` and puts the `Pair` on to the heap:

```
pub fn cons<'guard>(
    mem: &'guard MutatorView,
    head: TaggedScopedPtr<'guard>,
    rest: TaggedScopedPtr<'guard>,
) -> Result<TaggedScopedPtr<'guard>, RuntimeError> {
    let pair = Pair::new();
    pair.first.set(head);
    pair.second.set(rest);
    mem.alloc_tagged(pair)
}
```

Here we have the lifetime `'guard` associated with the `MutatorView` instance which grants access to the allocator `alloc_tagged()` method and the getter and setter on `TaggedScopedPtr`.

The other two args, `head` and `rest` are required to share the same `'guard` lifetime as the `MutatorView` instance, or rather, `'guard` must at least be a subtype of their lifetimes. Their values, of type `TaggedScopedPtr<'guard>`, can be written directly to the `first` and `second` members of `Pair` with the setter `TaggedCellPtr::set()`.

We'll also add a couple `impl` methods for appending an object to a `Pair` in linked-list fashion:

```
impl Pair {
    pub fn append<'guard>(
        &self,
        mem: &'guard MutatorView,
        value: TaggedScopedPtr<'guard>,
    ) -> Result<TaggedScopedPtr<'guard>, RuntimeError> {
        let pair = Pair::new();
        pair.first.set(value);

        let pair = mem.alloc_tagged(pair)?;
        self.second.set(pair);

        Ok(pair)
    }
}
```

This method, given a value to append, creates a new `Pair` whose member `first` points at the value, then sets the `second` of the `&self Pair` to that new `Pair` instance. This is in support of s-expression notation `(a b)` which describes a linked-list of `Pair`s arranged, in pseudo-Rust:

```
Pair {
    first: a,
    second: Pair {
        first: b,
        second: nil,
    },
}
```

The second method is for directly setting the value of the `second` for s-expression dot-notation style: `(a . b)` is represented by `first` pointing at `a`, dotted with `b` which is pointed at by `second`. In our pseudo representation:

```
Pair {
    first: a,
    second: b,
}
```

The implementation is simply:

```
impl Pair {
    pub fn dot<'guard>(&self, value: TaggedScopedPtr<'guard>) {
        self.second.set(value);
    }
}
```

The only other piece to add, since `Pair` must be able to be passed into our allocator API, is the `AllocObject` impl for `Pair`:

```
impl AllocObject<TypeList> for Pair {
    const TYPE_ID: TypeList = TypeList::Pair;
}
```

This impl pattern will repeat for every type in `TypeList` so it'll be a great candidate for a macro.

And that's it! We have a cons-cell style `Pair` type and some elementary methods for creating and allocating them.

Now, back to `Symbol`, which seems like it should be even simpler, but as we'll see has some nuance to it.

Symbols and pointers

Let's recap the definition of `Symbol` and that it is the raw members of a `&str`:

```
#[derive(Copy, Clone)]
pub struct Symbol {
    name_ptr: *const u8,
    name_len: usize,
}
```

By this definition, a symbol has a name string, but does not own the string itself. What means this?

Symbols are in fact pointers to interned strings. Since each symbol points to a unique string, we can identify a symbol by it's pointer value rather than needing to look up the string itself.

However, symbols do need to be discovered by their string name, and symbol pointers must dereference to return their string form. i.e. a we need a bidirectional mapping of string to pointer and pointer to string.

In our implementation, we use a `HashMap<String, RawPtr<Symbol>>` to map from name strings to symbol pointers, while the `Symbol` object itself points back to the name string.

This is encapsulated in a `SymbolMap` struct:

```
pub struct SymbolMap {
    map: RefCell<HashMap<String, RawPtr<Symbol>>>,
    arena: Arena,
}
```

where we use `RefCell` to wrap operations in interior mutability, just like all other allocator functionality.

The second struct member `Arena` requires further explanation: since symbols are unique strings that can be identified and compared by their pointer values, these pointer values must remain static throughout the program lifetime. Thus, `Symbol` objects cannot be managed by a heap that might perform object relocation. We need a separate heap type for objects that are never moved or freed unil the program ends, the `Arena` type.

The `Arena` type is simple. It, like `Heap`, wraps `StickyImmixHeap` but unlike `Heap`, it will never run garbage collection.

```
pub struct Arena {
    heap: StickyImmixHeap<ArenaHeader>,
}
```

The `ArenaHeader` is a simple object header type to fulfill the allocator API requirements but whose methods will never be needed.

Allocating a `Symbol` will use the `Arena::alloc()` method which calls through to the `StickyImmixHeap` instance.

We'll add a method for getting a `Symbol` from it's name string to the `SymbolMap` at the allocator API level:

```
impl SymbolMap {
    pub fn lookup(&self, name: &str) -> RawPtr<Symbol> {
        {
            if let Some(ptr) = self.map.borrow().get(name) {
                return *ptr;
            }
        }

        let name = String::from(name);
        let ptr = self.arena.alloc(Symbol::new(&name)).unwrap();
        self.map.borrow_mut().insert(name, ptr);
        ptr
    }
}
```

Then we'll add wrappers to the `Heap` and `MutatorView` impls to scope-restrict access:

```
impl Heap {
    fn lookup_sym(&self, name: &str) -> TaggedPtr {
        TaggedPtr::symbol(self.syms.lookup(name))
    }
}
```

and

```
impl<'memory> MutatorView<'memory> {
    pub fn lookup_sym(&self, name: &str) -> TaggedScopedPtr<'_> {
        TaggedScopedPtr::new(self, self.heap.lookup_sym(name))
    }
}
```

This scope restriction is absolutely necessary, despite these objects never being freed or moved during runtime. This is because `Symbol`, as a standalone struct, remains unsafe to use with it's raw `&str` components. These components can only safely be accessed when there is a guarantee that the backing `Hashmap` is still in existence, which is only when the `MutatorView` is accessible.

Two methods on `Symbol` guard access to the `&str`, one unsafe to reassemble the `&str` from raw components, the other safe when given a `MutatorScope` guard instance.

```
impl Symbol {
    pub unsafe fn unguarded_as_str<'desired_lifetime>(&self) ->
&'desired_lifetime str {
        let slice = slice::from_raw_parts(self.name_ptr, self.name_len);
        str::from_utf8(slice).unwrap()
    }

    pub fn as_str<'guard>(&self, _guard: &'guard dyn MutatorScope) -> &'guard
str {
        unsafe { self.unguarded_as_str() }
    }
}
```

Finally, to make `Symbol`s allocatable in the Sticky Immix heap, we need to implement `AllocObject` for it:

```
impl AllocObject<TypeList> for Symbol {
    const TYPE_ID: TypeList = TypeList::Symbol;
}
```

Moving on swiftly

Now we've got the elemental pieces of s-expressions, lists and symbols, we can move on to parsing s-expression strings.

Since the focus of this book is the underlying mechanisms of memory management in Rust and the details of runtime implementation, parsing will receive less attention. We'll make it quick!

Parsing s-expressions

We'll make this quick. It's not the main focus of this book and the topic is better served by seeking out other resources that can do it justice.

In service of keeping it short, we're parsing s-expressions and we'll start by considering only symbols and parentheses. We could hardly make it simpler.

The interface

The interface we want should take a `&str` and return a `TaggedScopedPtr`. We want the tagged version of the scoped ptr because the return value might point to either a `Pair` or a `Symbol`. Examples of valid input are:

- `a-symbol`: a `Symbol` with name "a-symbol"
- `(this is a list)`: a linked list of `Pair`s, each with the `first` value pointing to a `Symbol`
- `(this (is a nested) list)`: a linked list, as above, containing a nested linked list
- `(this () is a nil symbol)`: the two characters `()` together are equivalent to the special symbol `nil`, also the value `0` in our `TaggedPtr` type
- `(one . pair)`: a single `Pair` instance with `first` pointing at the `Symbol` for "one" and `second` at the `Symbol` for "two"

Our internal implementation is split into tokenizing and then parsing the token stream. Tokenizing takes the `&str` input and returns a `Vec<Token>` on success:

```
fn tokenize(input: &str) -> Result<Vec<Token>, RuntimeError>;
```

The return `Vec<Token>` is an intermediate, throwaway value, and does not interact with our Sticky Immix heap. Parsing takes the `Vec<Token>` and returns a `TaggedScopedPtr` on success:

```
fn parse_tokens<'guard>(
    mem: &'guard MutatorView,
    tokens: Vec<Token>,
) -> Result<TaggedScopedPtr<'guard>, RuntimeError>;
```

Tokens, a short description

The full set of tokens we will consider parsing is:

```
#[derive(Debug, PartialEq)]
pub enum TokenType {
    OpenParen,
    CloseParen,
    Symbol(String),
    Dot,
    Text(String),
    Quote,
}
```

We combine this enum with a source input position indicator to compose the `Token` type. This source position is defined as:

```
#[derive(Copy, Clone, Debug, PartialEq)]
pub struct SourcePos {
    pub line: u32,
    pub column: u32,
}
```

And whenever it is available to return as part of an error, error messages can be printed with the relevant source code line.

The `Token` type;

```
#[derive(Debug, PartialEq)]
pub struct Token {
    pub pos: SourcePos,
    pub token: TokenType,
}
```

Parsing, a short description

The key to quickly writing a parser in Rust is the `std::iter::Peekable` iterator which can be obtained from the `Vec<Token>` instance with `tokens.iter().peekable()`. This iterator has a `peek()` method that allows you to look at the next `Token` instance without advancing the iterator.

Our parser, a hand-written recursive descent parser, uses this iterator type to look ahead to the next token to identify primarily whether the next token is valid in combination with the current token, or to know how to recurse next without consuming the token yet.

For example, an open paren `(` followed by a symbol would start a new `Pair` linked list, recursing into a new parser function call, but if it is immediately followed by a close paren `)`, that is `()`, it is equivalent to the symbol `nil`, while otherwise `)` *terminates* a `Pair` linked list and causes the current parsing function instance to return.

Another case is the `.` operator, which is only valid in the following pattern: `(a b c . d)` where `a`, `b`, `c`, and `d` must be symbols or nested lists. A `.` must be followed by a single expression followed by a `)`.

Tokenizing and parsing are wrapped in a function that takes the input `&str` and gives back the `TaggedScopedPtr`:

```
pub fn parse<'guard>(
    mem: &'guard MutatorView,
    input: &str,
) -> Result<TaggedScopedPtr<'guard>, RuntimeError> {
    parse_tokens(mem, tokenize(input)?)
}
```

Notice that this function and `parse_tokens()` require the `mem: &'guard MutatorView` parameter. Parsing creates `Symbol` and `Pair` instances in our Sticky Immix heap and so requires the scope-restricted `MutatorView` instance.

This is all we'll say on parsing s-expressions. In the next chapter we'll do something altogether more informative with regards to memory management and it'll be necessary by the time we're ready to compile: arrays!

Arrays

Before we get to the basics of compilation, we need another data structure: the humble array. The first use for arrays will be to store the bytecode sequences that the compiler generates.

Rust already provides `Vec` but as we're implementing everything in terms of our memory management abstraction, we cannot directly use `Vec`. Rust does not (yet) expose the ability to specify a custom allocator type as part of `Vec`, nor are we interested in replacing the global allocator.

Our only option is to write our own version of `Vec`! Fortunately we can learn a lot from `Vec` itself and it's underlying implementation. Jump over to the Rustonomicon for a primer on the internals of `Vec`.

The first thing we'll learn is to split the implementation into a `RawArray<T>` type and an `Array<T>` type. `RawArray<T>` will provide an unsafe abstraction while `Array<T>` will make a safe layer over it.

RawArray

If you've just come back from *Implementing Vec* in the Nomicon, you'll recognize what we're doing below with `RawArray<T>`:

```
pub struct RawArray<T: Sized> {
    /// Count of T-sized objects that can fit in the array
    capacity: ArraySize,
    ptr: Option<NonNull<T>>,
}
```

Instead of `Unique<T>` for the pointer, we're using `Option<NonNull<T>>`. One simple reason is that `Unique<T>` is likely to be permanently unstable and only available internally to `std` collections. The other is that we can avoid allocating the backing store if no capacity is requested yet, setting the value of `ptr` to `None`.

For when we *do* know the desired capacity, there is `RawArray<T>::with_capacity()`. This method, because it allocates, requires access to the `MutatorView` instance. If you'll recall from the chapter on the allocation API, the API provides an array allocation method with signature:

```
AllocRaw::alloc_array(&self, size_bytes: ArraySize) -> Result<RawPtr<u8>,
AllocError>;
```

This method is wrapped on the interpreter side by `Heap` and `MutatorView` and in both cases the return value remains, simply, `RawPtr<u8>` in the success case. It's up to `RawArray<T>` to receive the `RawPtr<u8>` value and maintain it safely. Here's `with_capcity()`, now:

```
    pub fn with_capacity<'scope>(
        mem: &'scope MutatorView,
        capacity: u32,
    ) -> Result<RawArray<T>, RuntimeError> {
        // convert to bytes, checking for possible overflow of ArraySize
limit
        let capacity_bytes = capacity
            .checked_mul(size_of::<T>() as ArraySize)
            .ok_or(RuntimeError::new(ErrorKind::BadAllocationRequest))?;

        Ok(RawArray {
            capacity,
            ptr: NonNull::new(mem.alloc_array(capacity_bytes)?.as_ptr() as
*mut T),
        })
    }
```

Resizing

If a `RawArray<T>`'s content will exceed it's capacity, there is `RawArray<T>::resize()`. It

allocates a new backing array using the `MutatorView` method `alloc_array()` and copies the content of the old over to the new, finally swapping in the new backing array for the old.

The code for this is straightforward but a little longer, go check it out in `interpreter/src/rawarray.rs`.

Accessing

Since `RawArray<T>` will be wrapped by `Array<T>`, we need a couple more methods to access the raw memory:

```
impl<T: Sized> RawArray<T> {
    pub fn capacity(&self) -> ArraySize {
        self.capacity
    }

    pub fn as_ptr(&self) -> Option<*const T> {
        match self.ptr {
            Some(ptr) => Some(ptr.as_ptr()),
            None => None,
        }
    }
}
```

And that's it! Now for the safe wrapper.

Array

The definition of the struct wrapping `RawArray<T>` is as follows:

```
#[derive(Clone)]
pub struct Array<T: Sized + Clone> {
    length: Cell<ArraySize>,
    data: Cell<RawArray<T>>,
    borrow: Cell<BorrowFlag>,
}
```

Here we have three members:

- `length` - the length of the array
- `data` - the `RawArray<T>` being wrapped
- `borrow` - a flag serving as a runtime borrow check, allowing `RefCell` runtime semantics, since we're in a world of interior mutability patterns

We have a method to create a new array - `Array::alloc()`

```
impl<T: Sized + Clone> Array<T> {
    pub fn alloc<'guard>(
        mem: &'guard MutatorView,
    ) -> Result<ScopedPtr<'guard, Array<T>>, RuntimeError>
    where
        Array<T>: AllocObject<TypeList>,
    {
        mem.alloc(Array::new())
    }
}
```

In fact we'll extend this pattern of a method named "alloc" to any data structure for convenience sake.

There are many more methods for `Array<T>` and it would be exhausting to be exhaustive. Let's go over the core methods used to read and write elements and then an example use case.

Reading and writing

First of all, we need a function that takes an array index and returns a pointer to a memory location, if the index is within bounds:

```
impl<T: Sized + Clone> Array<T> {
    fn get_offset(&self, index: ArraySize) -> Result<*mut T, RuntimeError> {
        if index >= self.length.get() {
            Err(RuntimeError::new(ErrorKind::BoundsError))
        } else {
            let ptr = self
                .data
                .get()
                .as_ptr()
                .ok_or_else(|| RuntimeError::new(ErrorKind::BoundsError))?;

            let dest_ptr = unsafe { ptr.offset(index as isize) as *mut T };

            Ok(dest_ptr)
        }
    }
}
```

There are two bounds checks here - firstly, the index should be within the (likely non-zero) length values; secondly, the `RawArray<T>` instance should have a backing array allocated. If either of these checks fail, the result is an error. If these checks pass, we can be confident that there is array backing memory and that we can return a valid pointer to somewhere inside that memory block.

For reading a value in an array, we need two methods:

1. one that handles move/copy semantics and returns a value
2. one that handles reference semantics and returns a reference to the original value in it's location in the backing memory

First, then:

```
impl<T: Sized + Clone> Array<T> {
    fn read<'guard>(
        &self,
        _guard: &'guard dyn MutatorScope,
        index: ArraySize,
    ) -> Result<T, RuntimeError> {
        unsafe {
            let dest = self.get_offset(index)?;
            Ok(read(dest))
        }
    }
}
```

and secondly:

```
impl<T: Sized + Clone> Array<T> {
    pub fn read_ref<'guard>(
        &self,
        _guard: &'guard dyn MutatorScope,
        index: ArraySize,
    ) -> Result<&T, RuntimeError> {
        unsafe {
            let dest = self.get_offset(index)?;
            Ok(&*dest as &T)
        }
    }
}
```

Writing, or copying, an object to an array is implemented as simply as follows:

```
impl<T: Sized + Clone> Array<T> {
    pub fn read_ref<'guard>(
        &self,
        _guard: &'guard dyn MutatorScope,
        index: ArraySize,
    ) -> Result<&T, RuntimeError> {
        unsafe {
            let dest = self.get_offset(index)?;
            Ok(&*dest as &T)
        }
    }
}
```

These simple functions should only be used internally by `Array<T>` impl methods. We

have numerous methods that wrap the above in more appropriate semantics for values of `T` in `Array<T>`.

The Array interfaces

To define the interfaces to the Array, and other collection types, we define a number of traits. For example, a collection that behaves as a stack implements `StackContainer`; a numerically indexable type implements `IndexedContainer`, and so on. As we'll see, there is some nuance, though, when it comes to a difference between collections of non-pointer types and collections of pointer types.

For our example, we will describe the stack interfaces of `Array<T>`.

First, the general case trait, with methods for accessing values stored in the array (non-pointer types):

```
pub trait StackContainer<T: Sized + Clone>: Container<T> {
    /// Push can trigger an underlying array resize, hence it requires the
ability to allocate
    fn push<'guard>(&self, mem: &'guard MutatorView, item: T) -> Result<(),
RuntimeError>;

    /// Pop returns a bounds error if the container is empty, otherwise moves
the last item of the
    /// array out to the caller.
    fn pop<'guard>(&self, _guard: &'guard dyn MutatorScope) -> Result<T,
RuntimeError>;

    /// Return the value at the top of the stack without removing it
    fn top<'guard>(&self, _guard: &'guard dyn MutatorScope) -> Result<T,
RuntimeError>;
}
```

These are unremarkable functions, by now we're familiar with the references to `MutatorScope` and `MutatorView` in method parameter lists.

In any instance of `Array<T>`, `T` need only implement `Clone` and cannot be dynamically sized. Thus `T` can be any primitive type or any straightforward struct.

What if we want to store pointers to other objects? For example, if we want a heterogenous array, such as Python's `List` type, what would we provide in place of `T`? The answer is to use the `TaggedCellPtr` type. However, an `Array<TaggedCellPtr`, because we want to interface with pointers and use the memory access abstractions provided, can be made a little more ergonomic. For that reason, we have separate traits for containers of type `Container<TaggedCellPtr`. In the case of the stack interface this looks like:

```
pub trait StackAnyContainer: StackContainer<TaggedCellPtr> {
    /// Push can trigger an underlying array resize, hence it requires the
ability to allocate
    fn push<'guard>(
        &self,
        mem: &'guard MutatorView,
        item: TaggedScopedPtr<'guard>,
    ) -> Result<(), RuntimeError>;

    /// Pop returns a bounds error if the container is empty, otherwise moves
the last item of the
    /// array out to the caller.
    fn pop<'guard>(
        &self,
        _guard: &'guard dyn MutatorScope,
    ) -> Result<TaggedScopedPtr<'guard>, RuntimeError>;

    /// Return the value at the top of the stack without removing it
    fn top<'guard>(
        &self,
        _guard: &'guard dyn MutatorScope,
    ) -> Result<TaggedScopedPtr<'guard>, RuntimeError>;
}
```

As you can see, these methods, while for `T = TaggedCellPtr`, provide an interface based on passing and returning `TaggedScopedPtr`.

Let's look at the implementation of one of these methods - `push()` - for both `StackContainer` and `StackAnyContainer`.

Here's the code for `StackContainer::push()`:

```
impl<T: Sized + Clone> StackContainer<T> for Array<T> {
    fn push<'guard>(&self, mem: &'guard MutatorView, item: T) -> Result<(),
RuntimeError> {
        if self.borrow.get() != INTERIOR_ONLY {
            return Err(RuntimeError::new(ErrorKind::MutableBorrowError));
        }

        let length = self.length.get();
        let mut array = self.data.get(); // Takes a copy

        let capacity = array.capacity();

        if length == capacity {
            if capacity == 0 {
                array.resize(mem, DEFAULT_ARRAY_SIZE)?;
            } else {
                array.resize(mem, default_array_growth(capacity)?)?;
            }
            // Replace the struct's copy with the resized RawArray object
            self.data.set(array);
        }

        self.length.set(length + 1);
        self.write(mem, length, item)?;
        Ok(())
    }
}
```

In summary, the order of operations is:

1. Check that a runtime borrow isn't in progress. If it is, return an error.
2. Since we must implement interior mutability, the member `data` of the `Array<T>` struct is a `Cell`. We have to `get()` the content in order to use it.
3. We then ask whether the array backing store needs to be grown. If so, we resize the `RawArray<T>` and, since it's kept in a `Cell` on `Array<T>`, we have to `set()` value back into `data` to save the change.
4. Now we have an `RawArray<T>` that has enough capacity, the length is incremented and the object to be pushed is written to the next memory location using the internal `Array<T>::write()` method detailed earlier.

Fortunately we can implement `StackAnyContainer::push()` in terms of `StackContainer::push()`:

```
impl StackAnyContainer for Array<TaggedCellPtr> {
    fn push<'guard>(
        &self,
        mem: &'guard MutatorView,
        item: TaggedScopedPtr<'guard>,
    ) -> Result<(), RuntimeError> {
        StackContainer::<TaggedCellPtr>::push(self, mem,
TaggedCellPtr::new_with(item))
    }
}
```

One last thing

To more easily differentiate arrays of type `Array<T>` from arrays of type `Array<TaggedCellPtr>`, we make a type alias `List` where:

```
pub type List = Array<TaggedCellPtr>;
```

In conclusion

We referenced how `Vec` is implemented internally and followed the same pattern of defining a `RawArray<T>` unsafe layer with a safe `Array<T>` wrapper. Then we looked into the stack interface for `Array<T>` and the implementation of `push()`.

There is more to arrays, of course - indexed access the most obvious, and also a few convenience methods. See the source code in `interpreter/src/array.rs` for the full detail.

In the next chapter we'll put `Array<T>` to use in a `Bytecode` type!

Bytecode

In this chapter we will look at a bytecode compilation target. We'll combine this with a section on the virtual machine interface to the bytecode data structure.

We won't go much into detail on each bytecode operation, that will be more usefully covered in the compiler and virtual machine chapters. Here, we'll describe the data structures involved. As such, this will be one of our shorter chapters. Let's go!

Design questions

Now that we're talking bytecode, we're at the point of choosing what type of virtual machine we will be compiling for. The most common type is stack-based where operands are pushed and popped on and off the stack. This requires instructions for pushing and popping, with instructions in-between for operating on values on the stack.

We'll be implementing a register-based VM though. The inspiration for this comes from Lua 5[1] which implements a fixed-width bytecode register VM. While stack based VMs are typically claimed to be simpler, we'll see that the Lua way of allocating registers per function also has an inherent simplicity and has performance gains over a stack VM, at least for an interpreted non jit-compiled VM.

Given register based, fixed-width bytecode, each opcode must reference the register numbers that it operates on. Thus, for an (untyped) addition operation `x = a + b`, each of `x`, `a` and `b` must be associated with a register.

Following Lua, encoding this as a fixed width opcode typically looks like encoding the operator and operands as 8 bit values packed into a 32 bit opcode word. That implies, given 8 bits, that there can be a theoretical maximum of 256 registers for a function call. For the addition above, this encoding might look like this:

```
32.....24......16.......8.......0
 [reg a ][reg b ][reg x ][Add   ]
```

where the first 8 bits contain the operator, in this case "Add", and the other three 8 bit slots in the 32 bit word each contain a register number.

For some operators, we will need to encode values larger than 8 bits. As we will still need space for an operator and a destination register, that leaves a maximum of 16 bits for larger values.

Opcodes

We have options in how we describe opcodes in Rust.

1. Each opcode represented by a u32
    - Pros: encoding flexibility, it's just a set of bits
    - Cons: bit shift and masking operations to encode and decode operator and operands. This isn't necessarily a big deal but it doesn't allow us to leverage the Rust type system to avoid encoding mistakes
2. Each opcode represented by an enum discriminant
    - Pros: operators and operands baked as Rust types at compile time, type safe encoding; no bit operations needed
    - Cons: encoding scheme limited to what an enum can represent

The ability to leverage the compiler to prevent opcode encoding errors is attractive and we won't have any need for complex encodings. We'll use an enum to represent all possible opcodes and their operands.

Since a Rust enum can contain named values within each variant, this is what we use to most tightly define our opcodes.

Opcode size

Since we're using `enum` instead of a directly size-controlled type such as u32 for our opcodes, we have to be more careful about making sure our opcode type doesn't take up more space than is necessary. 32 bits is ideal for reasons stated earlier (8 bits for the operator and 8 bits for three operands each.)

Let's do an experiment.

First, we need to define a register as an 8 bit value. We'll also define an inline literal integer as 16 bits.

```
type Register = u8;
type LiteralInteger = i16;
```

Then we'll create an opcode enum with a few variants that might be typical:

```
#[derive(Copy, Clone)]
enum Opcode {
    Add {
        dest: Register,
        a: Register,
        b: Register
    },
    LoadLiteral {
        dest: Register,
        value: LiteralInteger
    }
}
```

It should be obvious that with an enum like this we can safely pass compiled bytecode from the compiler to the VM. It should also be clear that this, by allowing use of `match` statements, will be very ergonomic to work with.

Theoretically, if we never have more than 256 variants, our variants never have more than 3 `Register` values (or one `Register` and one `LiteralInteger` sized value), the compiler should be able to pack `Opcode` into 32 bits.

Our test: we hope the output of the following code to be `4` - 4 bytes or 32 bits.

```
use std::mem::size_of;

fn main() {
    println!("Size of Opcode is {}", size_of::<Opcode>());
}
```

And indeed when we run this, we get `Size of Opcode is 4`!

To keep an eye on this situation, we'll put this check into a unit test:

```
    #[test]
    fn test_opcode_is_32_bits() {
        // An Opcode should be 32 bits; anything bigger and we've mis-defined
some
        // variant
        assert!(size_of::<Opcode>() == 4);
    }
```

Now, let's put these `Opcode`s into an array.

An array of Opcode

We can define this array easily, given that `Array<T>` is a generic type:

```
pub type ArrayOpcode = Array<Opcode>;
```

Is this enough to define bytecode? Not quite. We've accommodated 16 bit literal signed integers, but all kinds of other types can be literals. We need some way of referencing any literal type in bytecode. For that we add a `Literals` type, which is just:

```
pub type Literals = List;
```

Any opcode that loads a literal (other than a 16 bit signed integer) will need to reference an object in the `Literals` list. This is easy enough: just as there's a `LiteralInteger`, we have `LiteralId` defined as

```
pub type LiteralId = u16;
```

This id is an index into the `Literals` list. This isn't the most efficient scheme or encoding, but given a preference for fixed 32 bit opcodes, it will also keep things simple.

The `ByteCode` type, finally, is a composition of `ArrayOpcode` and `Literals`:

```
#[derive(Clone)]
pub struct ByteCode {
    code: ArrayOpcode,
    literals: Literals,
}
```

Bytecode compiler support

There are a few methods implemented for `ByteCode`:

1. `fn push<'guard>(&self, mem: &'MutatorView, op: Opcode) -> Result<(), RuntimeError>` This function pushes a new opcode into the `ArrayOpcode` instance.

2. ```
   fn update_jump_offset<'guard>(
       &self,
       mem: &'guard MutatorView,
       instruction: ArraySize,
       offset: JumpOffset,
   ) -> Result<(), RuntimeError>
   ```

   This function, given an instruction index into the `ArrayOpcode` instance, and given that the instruction at that index is a type of jump instruction, sets the relative jump offset of the instruction to the given offset. This is necessary because forward jumps cannot be calculated until all the in-between instructions have been compiled first.

3. ```
   fn push_lit<'guard>(
       &self,
       mem: &'guard MutatorView,
       literal: TaggedScopedPtr
   ) -> Result<LiteralId, RuntimeError>
   ```

   This function pushes a literal on to the `Literals` list and returns the index - the id - of the item.

4. ```
   fn push_loadlit<'guard>(
       &self,
       mem: &'guard MutatorView,
       dest: Register,
       literal_id: LiteralId,
   ) -> Result<(), RuntimeError>
   ```

   After pushing a literal into the `Literals` list, the corresponding load instruction
   ```

should be pushed into the `ArrayOpcode` list.

`ByteCode` and it's functions combined with the `Opcode` enum are enough to build a compiler for.

Bytecode execution support

The previous section described a handful of functions for our compiler to use to build a `ByteCode` structure.

We'll need a different set of functions for our virtual machine to access `ByteCode` from an execution standpoint.

The execution view of bytecode is of a contiguous sequence of instructions and an instruction pointer. We're going to create a separate `ByteCode` instance for each function that gets compiled, so our execution model will have to be able to jump between `ByteCode` instances. We'll need a new struct to represent that:

```
pub struct InstructionStream {
    instructions: CellPtr<ByteCode>,
    ip: Cell<ArraySize>,
}
```

In this definition, the pointer `instructions` can be updated to point at any `ByteCode` instance. This allows us to switch between functions by managing different `ByteCode` pointers as part of a stack of call frames. In support of this we have:

```
impl InstructionStream {
    pub fn switch_frame(&self, code: ScopedPtr<'_, ByteCode>, ip: ArraySize)
{
        self.instructions.set(code);
        self.ip.set(ip);
    }
}
```

Of course, the main function needed during execution is to retrieve the next opcode. Ideally, we can keep a pointer that points directly at the next opcode such that only a single dereference and pointer increment is needed to get the opcode and advance the instruction pointer. Our implementation is less efficient for now, requiring a dereference of 1. the `ByteCode` instance and then 2. the `ArrayOpcode` instance and finally 3. an indexing into the `ArrayOpcode` instance:

```
    pub fn get_next_opcode<'guard>(
        &self,
        guard: &'guard dyn MutatorScope,
    ) -> Result<Opcode, RuntimeError> {
        let instr = self
            .instructions
            .get(guard)
            .code
            .get(guard, self.ip.get())?;
        self.ip.set(self.ip.get() + 1);
        Ok(instr)
    }
```

Conclusion

The full `Opcode` definition can be found in `interpreter/src/bytecode.rs`.

As we work toward implementing a compiler, the next data structure we need is a dictionary or hash map. This will also build on the foundational `RawArray<T>` implementation. Let's go on to that now!

---

[1] Roberto Ierusalimschy et al, The Implementation of Lua 5.0

Dicts

The implementation of dicts, or hash tables, is going to combine a reuse of the RawArray type and closely follow the Crafting Interpreters design:

- open addressing
- linear probing
- FNV hashing

Go read the corresponding chapter in Crafting Interpreters and then come back here. We won't duplicate much of Bob's excellent explanation of the above terms and we'll assume you are familiar with his chapter when reading ours.

Code design

A `Dict` in our interpreter will allow any hashable value as a key and any type as a value.

We'll store pointers to the key and the value together in a struct `DictItem`.

Here, we'll also introduce the single diversion from Crafting Interpreters' implementation in that we'll cache the hash value and use it as part of a tombstone indicator. This adds an extra word per entry but we will also take the stance that if two keys have the same hash value then the keys are equal. This simplifies our implementation as we won't need to implement object equality comparisons just yet.

```
#[derive(Clone)]
pub struct DictItem {
    key: TaggedCellPtr,
    value: TaggedCellPtr,
    hash: u64,
}
```

The `Dict` itself mirrors Crafting Interpreters' implementation of a count of used entries and an array of entries. Since tombstones are counted as used entries, we'll add a separate `length` that excludes tombstones so we can accurately report the number of items in a dict.

```
pub struct Dict {
    /// Number of items stored
    length: Cell<ArraySize>,
    /// Total count of items plus tombstones
    used_entries: Cell<ArraySize>,
    /// Backing array for key/value entries
    data: Cell<RawArray<DictItem>>,
}
```

Hashing

To implement our compiler we will need to be able to hash the `Symbol` type and integers (inline in tagged pointers.)

The Rust standard library defines trait `std::hash::Hash` that must be implemented by types that want to be hashed. This trait requires the type to implement method `fn hash<H>(&self, state: &mut H) where H: Hasher`.

This signature requires a reference to the type `&self` to access it's data. In our world, this is insufficient: we also require a `&MutatorScope` lifetime to access an object. We will have to wrap `std::hash::Hash` in our own trait that extends, essentially the same signature, with this scope guard parameter. This trait is named `Hashable`:

```
/// Similar to Hash but for use in a mutator lifetime-limited scope
pub trait Hashable {
    fn hash<'guard, H: Hasher>(&self, _guard: &'guard dyn MutatorScope,
hasher: &mut H);
}
```

We can implement this trait for `Symbol` - it's a straightforward wrap of calling `Hash::hash()`:

```
impl Hashable for Symbol {
    fn hash<'guard, H: Hasher>(&self, guard: &'guard dyn MutatorScope, h:
&mut H) {
        self.as_str(guard).hash(h)
    }
}
```

Then finally, because this is all for a dynamically typed interpreter, we'll write a function that can take any type - a `TaggedScopedPtr` - and attempt to return a 64 bit hash value from it:

```
fn hash_key<'guard>(
    guard: &'guard dyn MutatorScope,
    key: TaggedScopedPtr<'guard>,
) -> Result<u64, RuntimeError> {
    match *key {
        Value::Symbol(s) => {
            let mut hasher = FnvHasher::default();
            s.hash(guard, &mut hasher);
            Ok(hasher.finish())
        }
        Value::Number(n) => Ok(n as u64),
        _ => Err(RuntimeError::new(ErrorKind::UnhashableError)),
    }
}
```

Now we can take a `Symbol` or a tagged integer and use them as keys in our `Dict`.

Finding an entry

The methods that a dictionary typically provides, lookup, insertion and deletion, all hinge around one internal function, `find_entry()`.

This function scans the internal `RawArray<DictItem>` array for a slot that matches the hash value argument. It may find an exact match for an existing key-value entry; if it does not, it will return the first available slot for the hash value, whether an empty never-before used slot or the tombstone entry of a formerly used slot.

A tombstone, remember, is a slot that previously held a key-value pair but has been

deleted. These slots must be specially marked so that when searching for an entry that generated a hash for an earlier slot but had to be inserted at a later slot, we know to keep looking rather than stop searching at the empty slot of a deleted entry.

Slot	Content
n - 1	empty
n	X: hash % capacity == n
n + 1	tombstone
n + 2	Y: hash % capacity == n
n + 3	empty

For example, in the above table:

- Key `X` 's hash maps to slot `n` .
- At some point another entry was inserted at slot `n + 1` .
- Then `Y` , with hash mapping also to slot `n` , was inserted, but had to be bumped to slot `n + 2` because the previous two slots were occupied.
- Then the entry at slot `n + 1` was deleted and marked as a tombstone.

If slot `n + 1` was simply marked as `empty` after it's occupant was deleted, then when searching for `Y` we wouldn't know to keep searching and find `Y` in slot `n + 2` . Hence, deleted entries are marked differently to empty slots.

Here is the code for the Find Entry function:

```
/// Given a key, generate the hash and search for an entry that either
matches this hash
/// or the next available blank entry.
fn find_entry<'guard>(
    _guard: &'guard dyn MutatorScope,
    data: &RawArray<DictItem>,
    hash: u64,
) -> Result<&'guard mut DictItem, RuntimeError> {
    // get raw pointer to base of array
    let ptr = data
        .as_ptr()
        .ok_or(RuntimeError::new(ErrorKind::BoundsError))?;

    // calculate the starting index into `data` to begin scanning at
    let mut index = (hash % data.capacity() as u64) as ArraySize;

    // the first tombstone we find will be saved here
    let mut tombstone: Option<&mut DictItem> = None;

    loop {
        let entry = unsafe { &mut *(ptr.offset(index as isize) as *mut
DictItem) as &mut DictItem };

        if entry.hash == TOMBSTONE && entry.key.is_nil() {
            // this is a tombstone: save the first tombstone reference we
find
            if tombstone.is_none() {
                tombstone = Some(entry);
            }
        } else if entry.hash == hash {
            // this is an exact match slot
            return Ok(entry);
        } else if entry.key.is_nil() {
            // this is a non-tombstone empty slot
            if let Some(earlier_entry) = tombstone {
                // if we recorded a tombstone, return _that_ slot to be
reused
                return Ok(earlier_entry);
            } else {
                return Ok(entry);
            }
        }

        // increment the index, wrapping back to 0 when we get to the end of
the array
        index = (index + 1) % data.capacity();
    }
}
```

To begin with, it calculates the index in the array from which to start searching. Then it iterates over the internal array, examining each entry's hash and key as it goes.

- The first tombstone that is encountered is saved. This may turn out to be the entry that should be returned if an exact hash match isn't found by the time a never-before used slot is reached. We want to reuse tombstone entries, of course.

- If no tombstone was found and we reach a never-before used slot, return that slot.
- If an exact match is found, return that slot of course.

The external API

Just as we defined some conainer traits for `Array<T>` to define access to arrays based on stack or indexed style access, we'll define a container trait for `Dict`:

```
pub trait HashIndexedAnyContainer {
    /// Return a pointer to to the object associated with the given key.
    /// Absence of an association should return an error.
    fn lookup<'guard>(
        &self,
        guard: &'guard dyn MutatorScope,
        key: TaggedScopedPtr,
    ) -> Result<TaggedScopedPtr<'guard>, RuntimeError>;

    /// Associate a key with a value.
    fn assoc<'guard>(
        &self,
        mem: &'guard MutatorView,
        key: TaggedScopedPtr<'guard>,
        value: TaggedScopedPtr<'guard>,
    ) -> Result<(), RuntimeError>;

    /// Remove an association by its key.
    fn dissoc<'guard>(
        &self,
        guard: &'guard dyn MutatorScope,
        key: TaggedScopedPtr,
    ) -> Result<TaggedScopedPtr<'guard>, RuntimeError>;

    /// Returns true if the key exists in the container.
    fn exists<'guard>(
        &self,
        guard: &'guard dyn MutatorScope,
        key: TaggedScopedPtr,
    ) -> Result<bool, RuntimeError>;
}
```

This trait contains the external API that `Dict` will expose for managing keys and values. The implementation of each of these methods will be in terms of the `find_entry()` function described above. Let's look at a couple of the more complex examples, `assoc()` and `dissoc()`.

assoc

```
impl HashIndexedAnyContainer for Dict {
    fn assoc<'guard>(
        &self,
        mem: &'guard MutatorView,
        key: TaggedScopedPtr<'guard>,
        value: TaggedScopedPtr<'guard>,
    ) -> Result<(), RuntimeError> {
        let hash = hash_key(mem, key)?;

        let mut data = self.data.get();
        // check the load factor (what percentage of the capacity is or has
been used)
        if needs_to_grow(self.used_entries.get() + 1, data.capacity()) {
            // create a new, larger, backing array, and copy all existing
entries over
            self.grow_capacity(mem)?;
            data = self.data.get();
        }

        // find the slot whose entry matches the hash or is the nearest
available entry
        let entry = find_entry(mem, &data, hash)?;

        // update counters if necessary
        if entry.key.is_nil() {
            // if `key` is nil, this entry is unused: increment the length
            self.length.set(self.length.get() + 1);
            if entry.hash == 0 {
                // if `hash` is 0, this entry has _never_ been used:
increment the count
                // of used entries
                self.used_entries.set(self.used_entries.get() + 1);
            }
        }

        // finally, write the key, value and hash to the entry
        entry.key.set(key);
        entry.value.set(value);
        entry.hash = hash;

        Ok(())
    }
}
```

dissoc

```
impl HashIndexedAnyContainer for Dict {
    fn dissoc<'guard>(
        &self,
        guard: &'guard dyn MutatorScope,
        key: TaggedScopedPtr,
    ) -> Result<TaggedScopedPtr<'guard>, RuntimeError> {
        let hash = hash_key(guard, key)?;

        let data = self.data.get();
        let entry = find_entry(guard, &data, hash)?;

        if entry.key.is_nil() {
            // a nil key means the key was not found in the Dict
            return Err(RuntimeError::new(ErrorKind::KeyError));
        }

        // decrement the length but not the `used_entries` count
        self.length.set(self.length.get() - 1);

        // write the "tombstone" markers to the entry
        entry.key.set_to_nil();
        entry.hash = TOMBSTONE;

        // return the value that was associated with the key
        Ok(entry.value.get(guard))
    }
}
```

As you can see, once `find_entry()` is implemented as a separate function, these methods become fairly easy to comprehend.

Conclusion

If you *haven't* read Bob Nystron's chapter on hash tables in Crafting Interpreters we encourage you to do so: it will help make sense of this chapter.

Now, we'll transition to some compiler and virtual machine design before we continue with code implementation.

Virtual Machine: Architecture and Design

In this short chapter we will outline our virtual machine design choices. These are substantially a matter of pragmatic dynamic language implementation points and as such, borrow heavily from uncomplicated prior work such as Lua 5 and Crafting Interpreters.

Bytecode

We already discussed our Lua-inspired bytecode in a previous chapter. To recap, we are using 32 bit fixed-width opcodes with space for 8 bit register identifiers and 16 bit literals.

The stack

Following the example of Crafting Interpreters we'll maintain two separate stack data structures:

- the register stack for storing stack values
- the call frame stack

In our case, these are best separated out because the register stack will be composed entirely of `TaggedCellPtr` s.

To store call frames on the register stack we would have to either:

1. allocate every stack frame on the heap with pointers to them from the register stack
2. or coerce a call frame `struct` type into the register stack type

Neither of these is attractive so we will maintain the call frame stack as an independent data structure.

The register stack

The register stack is a homogeneous array of `TaggedCellPtr` s. Thus, no object is allocated directly on the stack, all objects are heap allocated and the stack only consists of pointers to heap objects. The exception is literal integers that fit within the range allowed by a tagged pointer.

Since this is a register virtual machine, not following stack push and pop semantics, and bytecode operands are limited to 8 bit register indexes, a function is limited to addressing a maximum of 256 contiguous registers.

Due to function call nesting, the register stack may naturally grow much more than a length of 256.

This requires us to implement a sliding window into the register stack which will move as functions are called and return. The call frame stack will contain the stack base pointer for each function call. We can then happily make use a Rust slice to implement the window of 256 contiguous stack slots which a function call is limited to.

The call frame stack

A call frame needs to store three critical data points:

- a pointer to the function being executed
- the return instruction pointer when a nested function is called
- the stack base pointer for the function call

These three items can form a simple struct and we can define an `Array<CallFrame>` type for optimum performance.

Global values

To store global values, we have all we need: the `Dict` type that maps `Symbol`s to another value. The VM will, of course, have an abstraction over the internal `Dict` to enforce `Symbol`s only as keys.

Closures

In the classic upvalues implementation from Lua 5, followed also by Crafting Interpreters, a linked list of upvalues is used to map stack locations to shared variables.

In every respect but one, our implementation will be similar.

In our implementation, we'll use the `Dict` type that we already have available to do this mapping of stack locations to shared variables.

As the language and compiler will implement lexical scoping, the compiler will have static knowledge of the *relative* stack locations of closed-over variables and can generate the appropriate bytecode operands for the virtual machine to calculate the absolute stack locations at runtime. Thus, absolute stack locations can be mapped to `Upvalue` objects and so a `Dict` can be employed to facilitate the mapping. This obviates the need to implement a linked list data structure.

The compiler must issue instructions to tell the VM when to make a closure data structure. It can do so, of course, because simple analysis shows whether a function references nonlocal bindings. A closure data structure as generated by the compiler must reference the function that will be called and the list of relative stack locations that correspond to each nonlocal binding.

The VM, when executing the instruction to make a closure, will calculate the absolute stack locations for each nonlocal binding and create the closure environment - a

`List<Upvalue>`. VM instructions within the function code, as in Lua, indirectly reference nonlocal bindings by indexing into this environment.

Partial functions

Here is one point where we will introduce a less common construct in our virtual machine. Functions will be first class, that is they are objects that can be passed around as values and arguments. On top of that, we'll allow passing insufficient arguments to a function when it is called. The return value of such an operation will, instead of an error, be a `Partial` instance. This value must carry with it the arguments given and a pointer to the function waiting to be called.

This is insufficient for a fully featured currying implementation but is an interesting extension to first class functions, especially as it allows us to not *require* lambdas to be constructed syntactically every time they might be used.

By that we mean the following: if we have a function `(def mul (x y) (* x y))`, to turn that into a function that multiplies a number by 3 we'd normally have to define a second function, or lambda, `(lambda (x) (mul x 3))` and call it instead. However, with a simple partial function implementation we can avoid the lambda definition and call `(mul 3)` directly, which will collect the function pointer for `mul` and argument `3` into a `Partial` and wait for the final argument before calling into the function `mul` with both required arguments.

> ***Note:*** We can use the same struct for both closures and partial functions. A closure is a yet-to-be-called function carrying a list of references to values. or a list of values. A partial is a yet-to-be-called function carrying a list of arguments. They look very similar, and it's possible, of course, to partially apply arguments to a closure.

Instruction dispatch

In dispatch, one optimal outcome is to minimize the machine code overhead between each VM instruction code. This overhead, where the next VM instruction is fetched, decoded and mapped to the entry point of the instruction code, is the dispatch code. The other axis of optimization is code ergonomics.

Prior research into implementing dispatch in Rust concludes that simple switch-style dispatch is the only cross-platform construct we can reasonably make use of. Other mechanisms come with undesirable complexity or are platform dependent. For the most

part, with modern CPU branch prediction, the overhead of switch dispatch is small.

What this looks like: a single `match` expression with a pattern to represent each bytecode discriminant, all wrapped in a loop. To illustrate:

```
loop {
    let opcode = get_next_opcode();
    match opcode {
        Opcode::Add(a, x, y) => { ... },
        Opcode::Call(f, r, p) => { ... },
    }
}
```

That's it!

Next we'll look at the counterpart of VM design - compiler design.

Virtual Machine: Implementation

In this chapter we'll dive into some of the more interesting and important implementation details of our virtual machine.

To begin with, we'll lay out a struct for a single thread of execution. This struct should contain everything needed to execute the output of the compiler.

```
pub struct Thread {
    /// An array of CallFrames
    frames: CellPtr<CallFrameList>,
    /// An array of pointers any object type
    stack: CellPtr<List>,
    /// The current stack base pointer
    stack_base: Cell<ArraySize>,
    /// A dict that should only contain Number keys and Upvalue values. This
is a mapping of
    /// absolute stack indeces to Upvalue objects where stack values are
closed over.
    upvalues: CellPtr<Dict>,
    /// A dict that should only contain Symbol keys but any type as values
    globals: CellPtr<Dict>,
    /// The current instruction location
    instr: CellPtr<InstructionStream>,
}
```

Here we see every data structure needed to represent:

- function call frames
- stack values

- closed-over stack values (Upvalues)
- global values
- bytecode to execute

The VM's primary operation is to iterate through instructions, executing each in sequence. The outermost control struture is, therefore, a loop containing a `match` expression.

Here is a code extract of the opening lines of this match operation. The function shown is a member of the `Thread` struct. It evaluates the next instruction and is called in a loop by an outer function. We'll look at several extracts from this function in this chapter.

```
    /// Execute the next instruction in the current instruction stream
    fn eval_next_instr<'guard>(
        &self,
        mem: &'guard MutatorView,
    ) -> Result<EvalStatus<'guard>, RuntimeError> {
        // TODO not all these locals are required in every opcode - optimize
and get them only
        // where needed
        let frames = self.frames.get(mem);
        let stack = self.stack.get(mem);
        let globals = self.globals.get(mem);
        let instr = self.instr.get(mem);

        // Establish a 256-register window into the stack from the stack base
        stack.access_slice(mem, |full_stack| {
            let stack_base = self.stack_base.get() as usize;
            let window = &mut full_stack[stack_base..stack_base + 256];

            // Fetch the next instruction and identify it
            let opcode = instr.get_next_opcode(mem)?;

            match opcode {
                // Do nothing.
                Opcode::NoOp => return Ok(EvalStatus::Pending),

                ...
```

The function obtains a slice view of the register stack, then narrows that down to a 256 register window for the current function.

Then it fetches the next opcode and using `match`, decodes it.

Let's take a closer look at the stack.

The stack

While some runtimes and compilers, particularly low-level languages, have a single stack

that represents both function call information and local variables, our high-level runtime splits the stack into:

1. a stack of `CallFrame` objects containing function call and return information
2. and a register stack for local variables.

Let's look at each in turn.

The register stack

In our `Thread` struct, the register stack is represented by the two members:

```
pub struct Thread {
    ...
    stack: CellPtr<List>,
    stack_base: Cell<ArraySize>,
    ...
}
```

Remember that the `List` type is defined as `Array<TaggedCellPtr>` and is therefore an array of tagged pointers. Thus, the register stack is a homogenous array of word sized values that are pointers to objects on the heap or values that can be inlined in the tagged pointer word.

We also have a `stack_base` variable to quickly retrieve the offset into `stack` that indicates the beginning of the window of 256 registers that the current function has for it's local variables.

The call frame stack

In our `Thread` struct, the call frame stack is represented by the members:

```
pub struct Thread {
    ...
    frames: CellPtr<CallFrameList>,
    instr: CellPtr<InstructionStream>,
    ...
}
```

A `CallFrame` and an array of them are defined as:

```
#[derive(Clone)]
pub struct CallFrame {
    /// Pointer to the Function being executed
    function: CellPtr<Function>,
    /// Return IP when returning from a nested function call
    ip: Cell<ArraySize>,
    /// Stack base - index into the register stack where register window for
this function begins
    base: ArraySize,
}

pub type CallFrameList = Array<CallFrame>;
```

A `CallFrame` contains all the information needed to resume a function when a nested function call returns:

- a `Function` object, which references the `Bytecode` comprising the function
- the return instruction pointer
- the stack base index for the function's stack register window

On every function call, a `CallFrame` instance is pushed on to the `Thread`'s `frames` stack and on every return from a function, the top `CallFrame` is popped off the stack.

Additionally, we keep a pointer to the current executing function (the function represented by the top `CallFrame`) with the member `instr: CellPtr<InstructionStream>`.

For a review of the definition of `InstructionStream` see the bytecode chapter where we defined it as a pair of values - a `ByteCode` reference and a pointer to the next `Opcode` to fetch.

The VM keeps the `InstructionStream` object pointing at the same `ByteCode` object as is pointed at by the `Function` in the `CallFrame` at the top of the call frame stack. Thus, when a call frame is popped off the stack, the `InstructionStream` is updated with the `ByteCode` and instruction pointer from the `CallFrame` at the new stack top; and similarly when a function is called *into* and a new `CallFrame` is pushed on to the stack.

Functions and function calls

Function objects

Since we've mentioned `Function` objects above, let's now have a look at the definition.

```
#[derive(Clone)]
pub struct Function {
    /// name could be a Symbol, or nil if it is an anonymous fn
    name: TaggedCellPtr,
    /// Number of arguments required to activate the function
    arity: u8,
    /// Instructions comprising the function code
    code: CellPtr<ByteCode>,
    /// Param names are stored for introspection of a function signature
    param_names: CellPtr<List>,
    /// List of (CallFrame-index: u8 | Window-index: u8) relative offsets
from this function's
    /// declaration where nonlocal variables will be found. Needed when
creating a closure. May be
    /// nil
    nonlocal_refs: TaggedCellPtr,
}
```

Instances of `Function` are produced by the compiler, one for each function definition that is compiled, including nested function definitions.

A `Function` object is a simple collection of values, some of which may be `nil`. Any member represented by a `TaggedCellPtr` may, of course, contain a `nil` value.

Thus the function may be anonymous, represented by a `nil` name value.

While the function name is optional, the parameter names are always included. Though they do not need to be known in order to execute the function, they are useful for representing the function in string form if the programmer needs to introspect a function object.

Members that are *required* to execute the function are the arity, the `ByteCode` and any nonlocal references.

Nonlocal references are an optional list of `(relative_stack_frame, register)` tuples, provided by the compiler, that are needed to locate nonlocal variables on the register stack. These are, of course, a key component of implementing closures.

We'll talk about closures shortly, but before we do, we'll extend `Function`s with partial application of arguments.

Partial functions

A partial function application takes a subset of the arguments required to make a function call. These arguments must be stored for later.

Thus, a `Partial` object references the `Function` to be called and a list of arguments to give it when the call is finally executed.

Below is the definition of `Partial`. Note that it also contains a possible closure environment which, again, we'll arrive at momentarily.

```
#[derive(Clone)]
pub struct Partial {
    /// Remaining number of arguments required to activate the function
    arity: u8,
    /// Number of arguments already applied
    used: u8,
    /// List of argument values already applied
    args: CellPtr<List>,
    /// Closure environment - must be either nil or a List of Upvalues
    env: TaggedCellPtr,
    /// Function that will be activated when all arguments are applied
    func: CellPtr<Function>,
}
```

The `arity` and `used` members indicate how many arguments are expected and how many have been given. These are provided directly in this struct rather than requiring dereferencing the `arity` on the `Function` object and the length of the `args` list. This is for convenience and performance.

Each time more arguments are added to a `Partial`, a new `Partial` instance must be allocated and the existing arguments copied over. A `Partial` object, once created, is immutable.

Closures

Closures and partial applications have, at an abstract level, something in common: they both reference values that the function will need when it is finally called.

It's also possible, of course, to have a partially applied closure.

We can extend the `Partial` definition with a closure environment so that we can use the same object type everywhere to represent a function pointer, applied arguments and closure environment as needed.

Compiling a closure

The compiler, because it keeps track of variable names and scopes, knows when a `Function` references nonlocal variables. After such a function is defined, the compiler emits a `MakeClosure` instruction.

Referencing the stack with upvalues

The VM, when it executes `MakeClosure`, creates a new `Partial` object. It then iterates

over the list of nonlocal references and allocates an `Upvalue` object for each, which are added to the `env` member on the `Partial` object.

The below code extract is from the function `Thread::eval_next_instr()` in the `MakeClosure` instruction decode and execution block.

The two operands of the `MakeClosure` operation - `dest` and `function` - are registers. `function` points at the `Function` to be given an environment and made into a closure `Partial` instance; the pointer to this instance will be written to the `dest` register.

```
                // This operation should be generated by the compiler after a
function definition
                // inside another function but only if the nested function
refers to nonlocal
                // variables.
                // The result of this operation is a Partial with a closure
environment
                Opcode::MakeClosure { dest, function } => {
                    // 1. iter over function nonlocals
                    //   - calculate absolute stack offset for each
                    //   - find existing or create new Upvalue for each
                    //   - create closure environment with list of Upvalues
                    // 2. create new Partial with environment
                    // 3. set dest to Partial
                    let function_ptr = window[function as usize].get(mem);
                    if let Value::Function(f) = *function_ptr {
                        let nonlocals = f.nonlocals(mem);
                        // Create an environment array for upvalues
                        let env = List::alloc_with_capacity(mem,
nonlocals.length())?;

                        // Iter over function nonlocals, calculating absolute
stack offset for each
                        nonlocals.access_slice(mem, |nonlocals| -> Result<(),
RuntimeError> {
                            for compound in nonlocals {
                                // extract 8 bit register and call frame
values from 16 bit nonlocal
                                // descriptors
                                let frame_offset = (*compound >> 8) as
ArraySize;
                                let window_offset = (*compound & 0xff) as
ArraySize;

                                // look back frame_offset frames and add the
register number to
                                // calculate the absolute stack position of
the value
                                let frame = frames.get(mem, frames.length() -
frame_offset)?;
                                let location = frame.base + window_offset;

                                // look up, or create, the Upvalue for the
location, and add it to
                                // the environment
                                let (_, upvalue) =
self.upvalue_lookup_or_alloc(mem, location)?;
                                StackAnyContainer::push(&*env, mem,
upvalue.as_tagged(mem))?;
                            }

                            Ok(())
                        })?;

                        // Instantiate a Partial function application from
the closure environment
```

```
                        // and set the destination register
                        let partial = Partial::alloc(mem, f, Some(env),
&[])?;
                        window[dest as usize].set(partial.as_tagged(mem));
                    } else {
                        return Err(err_eval("Cannot make a closure from a
non-Function type"));
                    }
                }
```

The `Upvalue` struct itself is defined as:

```
#[derive(Clone)]
pub struct Upvalue {
    // Upvalue location can't be a pointer because it would be a pointer into
the dynamically
    // alloocated stack List - the pointer would be invalidated if the stack
gets reallocated.
    value: TaggedCellPtr,
    closed: Cell<bool>,
    location: ArraySize,
}
```

An `Upvalue` is an object that references an absolute register stack location (that is the `location` member.)

The initial value of `closed` is `false`. In this state, the location on the stack that contains the variable *must* be a valid location. That is, the stack can not have been unwound yet. If the closure is called, `Upvalue`s in this state are simply an indirection between the function and the variable on the register stack.

The compiler is able to keep track of variables and whether they are closed over. It emits bytecode instructions to close `Upvalue` objects when variables on the stack go out of scope.

This instruction, `CloseUpvalues`, copies the variable from the register stack to the `value` member of the `Upvalue` object and sets `closed` to `true`.

From then on, when the closure reads or writes to this variable, the value on the `Upvalue` object is modified rather than the location on the register stack.

Global values

```
pub struct Thread {
    ...
    globals: CellPtr<Dict>,
    ...
}
```

The outermost scope of a program's values and functions are the global values. We can manage these with an instance of a `Dict`. While a `Dict` can use any hashable value as a key, internally the VM will only allow `Symbol`s to be keys. That is, globals must be named objects.

Next...

Let's dive into the compiler!

Compiler: Design

Drawing from the VM design, the compiler must support the following language constructs:

- function definitions
- anonymous functions
- function calls
- lexical scoping
- closures
- local variables
- global variables
- expressions

This is a minimally critical set of features that any further language constructs can be built on while ensuring that our compiler remains easy to understand for the purposes of this book.

Our parser, recall, reads in s-expression syntax and produces a nested `Pair` and `Symbol` based abstract syntax tree. Adding other types - integers, strings, arrays etc - is mostly a matter of expanding the parser. The compiler as described here, being for a dynamically typed language, will support them without refactoring.

Eval/apply

Our compiler design is based on the *eval/apply* pattern.

In this pattern we recursively descend into the `Pair` AST, calling *eval* on the root node of the expression to be compiled.

Eval is, of course, short for "evaluate" - we want to evaluate the given expression. In the case of a compiler, we don't want the result yet, rather the sequence of instructions that will generate the result.

More concretely, *eval* looks at the node in the AST it is given and if it resolves to fetching a value for a variable, it generates that instruction; otherwise if it is a compound expression, the arguments are evaluated and then the function and arguments are passed to *apply*, which generates appropriate function call instructions.

Designing an Eval function

Eval looks at the given node and attempts to generate an instruction for it that would resolve the node to a value - that is, evaluate it.

Symbols

If the node is a special symbol, such as `nil` or `true`, then it is treated as a literal and an instruction is generated to load that literal symbol into the next available register.

Otherwise if the node is any other symbol, it is assumed to be bound to a value (it must be a variable) and an instruction is generated for fetching the value into a register.

Variables come in three kinds: local, nonlocal or global.

Local: the symbol has been declared earlier in the expression (either it is a function parameter or it was declared using `let`) and the compiler already has a record of it. The symbol is already associated with a local register index and a simple register copy instruction is generated.

Nonlocal: the symbol has been bound in a parent nesting function. Again, the compiler already has a record of the declaration, which register is associated with the symbol and which relative call frame will contain that register. An upvalue lookup instruction is generated.

Global: if the symbol isn't found as a local binding or a nonlocal binding, it is assumed to be a global, and a late-binding global lookup instruction is generated. In the event the programmer has misspelled a variable name, this is possibly the instruction that will be generated and the programmer will see an unknown-variable error at runtime.

Expressions and function calls

When *eval* is passed a `Pair`, this represents the beginning of an expression, a function call. A composition of things.

In s-expression syntax, all expressions and function calls looks like `(function_name arg1 arg2)`. That is parsed into a `Pair` tree, which takes the form:

```
Pair(
  Symbol(function_name),
  Pair(
    Symbol(arg1),
    Pair(
      Symbol(arg2),
      nil
    )
  )
)
```

It is *apply*'s job to handle this case, so *eval* extracts the first and second values from the outermost `Pair` and passes them into apply. In more general terms, *eval* calls *apply* with the function name and the argument list and leaves the rest up to *apply*.

Designing an Apply function

Apply takes a function name and a list of arguments. First it recurses into *eval* for each argument expression, then generates instructions to call the function with the argument results.

Calling functions

Functions are either built into to the language and VM or are library/user-defined functions composed of other functions.

In every case, the simplified pattern for function calls is:

- allocate a register to write the return value into
- *eval* each of the arguments in sequence, allocating their resulting values into consequent registers
- compile the function call opcode, giving it the number of argument registers it should expect

Compiling a call to a builtin function might translate directly to a dedicated bytecode operation. For example, querying whether a value is `nil` with builtin function `nil?` compiles 1:1 to a bytecode operation that directly represents that query.

Compiling a call to a user defined function is a more involved. In it's more general form, supporting first class functions and closures, a function call requires two additional pointers to be placed in registers. The complete function call register allocation looks like this:

Register	Use
0	reserved for return value

Register	Use
1	reserved for closure environment pointer
2	first argument
3	second argument
...	
n	function pointer

If a closure is called, the closure object itself contains a pointer to it's environment and the function to call and those pointers can be copied over to registers. Otherwise, the closure environment pointer will be a `nil` pointer.

The VM, when entering a new function, will represent the return value register always as the zeroth register.

When the function call returns, all registers except the return value are discarded.

Compiling functions

Let's look at a simple function definition:

```
(def is_true (x)
  (is? x true))
```

This function has a name `is_true`, takes one argument `x` and evaluates one expression `(is? x true)`.

The same function may be written without a name:

```
(lambda (x) (is? x true))
```

Compiling a function requires a few inputs:

- an optional reference to a parent nesting function
- an optional function name
- a list of argument names
- a list of expressions that will compute the return value

The desired output is a data structure that combines:

- the optional function name
- the argument names
- the compiled bytecode

First, a scope structure is established. A scope is a lexical block in which variables are bound and unbound. In the compiler, this structure is simply a mapping of variable name

to the register number that contains the value.

The first variables to be bound in the function's scope are the argument names. The compiler, given the list of argument names to the function and the order in which the arguments are given, associates each argument name with the register number that will contain it's value. As we saw above, these are predictably and reliably registers 2 and upward, one for each argument.

A scope may have a parent scope if the function is defined within another function. This is how nonlocal variable references will be looked up. We will go further into that when we discuss closures.

The second step is to *eval* each expression in the function, assigning the result to register 0, the preallocated return value register. The result of compiling each expression via *eval* is bytecode.

Thirdly and finally, a function object is instantiated, given it's name, the argument names and the bytecode.

Compiling closures

During compilation of the expressions within a function, if any of those expressions reference nonlocal variables (that is, variables not declared within the scope of the function) then the function object needs additional data to describe how to access those nonlocal variables at runtime.

In the below example, the anonymous inner function references the parameter `n` to the outer function, `n`. When the inner function is returned, the value of `n` must be carried with it even after the stack scope of the outer function is popped and later overwritten with values for other functions.

```
(def make_adder (n)
  (lambda (x) (+ x n))
)
```

Eval, when presented with a symbol to evaluate that has not been declared in the function scope, searches outer scopes next. If a binding is found in an outer scope, a nonlocal reference is added to the function's *local* scope that points to the outer scope and a `GetUpvalue` instruction is compiled.

This nonlocal reference is a combination of two values: a count of stack frames to skip over to find the outer scope variable and the register offset in that stack frame.

Non-local references are added to the function object that is returned by the function compiler. The VM will use these to identify the absolute location on the stack where a nonlocal variable should be read from and create upvalue objects at runtime when a

variable is closed over.

Compiling let

Let is the declaration of variables and assigning values: the binding of values, or the results of expressions, to symbols. Secondly, it provides space to evaluate expressions that incorporate those variables.

Here we bind the result of `(make_adder 3)` - a function - to the symbol `add_3` and then call `add_3` with argument `4`.

```
(let ((add_3 (make_adder 3)))
  (add_3 4))
```

The result of the entire `let` expression should be `7`.

Compiling `let` simply introduces additional scopes within a function scope. That is, instead of a function containing a single scope for all it's variables, scopes are nested. A stack of scopes is needed, with the parameters occupying the outermost scope.

First a new scope is pushed on to the scope stack and each symbol being bound is added to the new scope.

To generate code, a result register is reserved and a register for each binding is reserved.

Finally, each expression is evaluated and the scope is popped, removing the bindings from view.

Register allocation

A function call may make use of no more than 256 registers. Recall from earlier that the 0th register is reserved for the function return value and subsequent registers are reserved for the function arguments.

Beyond these initial registers the compiler uses a simple strategy in register allocation: if a variable (a parameter or a `let` binding) is declared, it is allocated a register based on a stack discipline. Thus, variables are essentially pushed and popped off the register stack as they come into and out of scope.

This strategy primarily ensures code simplicity - there is no register allocation optimization.

C'est tout!

That covers the VM and compiler design at an overview level. We've glossed over a lot of detail but the next chapters will expose the implementation detail. Get ready!

Compiler: Implementation

Before we get into eval and apply let's consider how we will support variables and lexical scoping.

Variables and Scopes

As seen in the previous chapter, variable accesses come in three types, as far as the compiler and VM are concerned: local, nonlocal and global. Each access uses a different bytecode operation, and so the compiler must be able to determine what operations to emit at compile time.

Given that we have named function parameters and `let`, we have syntax for explicit variable declaration within function definitions. This means that we can easily keep track of whether a variable reference is local, nonlocal or global.

If a variable wasn't declared as a parameter or in a `let` block, it must be global and global variables are accessed dynamically by name.

As far as local and nonlocal variables are concerned, the VM does not care about or consider their names. At the VM level, local and nonlocal variables are numbered registers. That is, each function's local variables are mapped to a register numbered between 2 and 255. The compiler must generate the mapping from variable names to register numbers.

For generating and maintaining mappings, we need data structures for keeping track of:

- function local variables and their mappings to register numbers
- references to nonlocal variables and their relative stack offsets
- nested scopes within functions

Named variables

Our first data structure will define a register based variable:

```
/// A variable is a named register. It has compile time metadata about how it
is used by closures.
struct Variable {
    register: Register,
    closed_over: Cell<bool>,
}
```

For every named, non-global variable (created by defining function parameters and `let` blocks) a `Variable` instance is created in the compiler.

The member `closed_over` defaults to `false`. If the compiler detects that the variable must escape the stack as part of a closure, this flag will be flipped to `true` (it cannot be set back to `false`.)

Scope structure

The data structures that manage nesting of scopes and looking up a `Variable` by name are defined here.

```
/// A Scope contains a set of local variable to register bindings
struct Scope {
    /// symbol -> variable mapping
    bindings: HashMap<String, Variable>,
}

/// A nonlocal reference will turn in to an Upvalue at VM runtime.
/// This struct stores the non-zero frame offset and register values of a
parent function call
/// frame where a binding will be located.
struct Nonlocal {
    upvalue_id: u8,
    frame_offset: u8,
    frame_register: u8,
}

/// A Variables instance represents a set of nested variable binding scopes
for a single function
/// definition.
struct Variables<'parent> {
    /// The parent function's variables.
    parent: Option<&'parent Variables<'parent>>,
    /// Nested scopes, starting with parameters/arguments on the outermost
scope and let scopes on
    /// the inside.
    scopes: Vec<Scope>,
    /// Mapping of referenced nonlocal nonglobal variables and their upvalue
indexes and where to
    /// find them on the stack.
    nonlocals: RefCell<HashMap<String, Nonlocal>>,
    /// The next upvalue index to assign when a new nonlocal is encountered.
    next_upvalue: Cell<u8>,
}
```

For every function defined, the compiler maintains an instance of the type `Variables`.

Each function's `Variables` has a stack of `Scope` instances, each of which has it's own set of name to `Variable` register number mappings. The outermost `Scope` contains the mapping of function parameters to registers.

A nested function's `Variables`, when the function refers to a nesting function's variable, builds a mapping of nonlocal variable name to relative stack position of that variable. This is a `NonLocal` - a relative stack frame offset and the register number within that stack frame of the variable.

In summary, under these definitions:

- A `Nonlocal` instance caches a relative stack location of a nonlocal variable for compiling upvalues
- `Scope` manages the mapping of a variable name to the `Variable` register number within a single scope
- `Variables` maintains all the nested scopes for a function during compilation and

caches all the nonlocal references. It also keeps a reference to a parent nesting function if there is one, in order to handle lexically scoped lookups.

Retrieving named variables

Whenever a variable is referenced in source code, the mapping to it's register must be looked up. The result of a lookup is `Option<Binding>`.

```
/// A binding can be either local or via an upvalue depending on how a
closure refers to it.
#[derive(Copy, Clone, PartialEq)]
enum Binding {
    /// An local variable is local to a function scope
    Local(Register),
    /// An Upvalue is an indirection for pointing at a nonlocal variable on
the stack
    Upvalue(UpvalueId),
}
```

The lookup process checks the local function scopes first.

If the variable is found to be declared there, `Some(Local)` enum variant is returned. In terms of bytecode, this will translate to a direct register reference.

Next, any outer function scopes are searched. If the variable is found in any outer scope, `Some(Upvalue)` variant is returned. The compiler will emit instructions to copy the value refered to by the upvalue into a function-local temporary register.

If the lookup for the variable returns `None`, a global lookup instruction is emitted that will dynamically look up the variable name in the global namespace and copy the result into a function-local temporary register or raise an error if the binding does not exist.

Evaluation

We've just somewhat described what happens in the lower levels of *eval*. Let's finish the job and put *eval* in a code context. Here is the definition of a function compilation data structure:

```
struct Compiler<'parent> {
    bytecode: CellPtr<ByteCode>,
    /// Next available register slot.
    next_reg: Register,
    /// Optional function name
    name: Option<String>,
    /// Function-local nested scopes bindings list (including parameters at
outer level)
    vars: Variables<'parent>,
}
```

The two interesting members are

- `bytecode`, which is an instance of ByteCode
- `vars`, an instance of `Variables` which we've described above. This instance will be the outermost scope of the `let` or function block being compiled.

The main entrypoint to this structure is the function `compile_function()`:

```
    fn compile_function<'guard>(
        mut self,
        mem: &'guard MutatorView,
        name: TaggedScopedPtr<'guard>,
        params: &[TaggedScopedPtr<'guard>],
        exprs: &[TaggedScopedPtr<'guard>],
    ) -> Result<ScopedPtr<'guard, Function>, RuntimeError> {
        ...
    }
```

This function will set up a `Variables` scope with the given parameters and call into function `compile_eval()` for each expression in the function. The full definition of `compile_eval()` is below, and we'll go into the details of `compile_function()` later.

```
    fn compile_eval<'guard>(
        &mut self,
        mem: &'guard MutatorView,
        ast_node: TaggedScopedPtr<'guard>,
    ) -> Result<Register, RuntimeError> {
        match *ast_node {
            Value::Pair(p) => self.compile_apply(mem, p.first.get(mem),
p.second.get(mem)),
            Value::Symbol(s) => {
                match s.as_str(mem) {
                    "nil" => {
                        let dest = self.acquire_reg();
                        self.push(mem, Opcode::LoadNil { dest })?;
                        Ok(dest)
                    }

                    "true" => self.push_load_literal(mem,
mem.lookup_sym("true")),

                    // Search scopes for a binding; if none do a global
lookup
                    _ => {
                        match self.vars.lookup_binding(ast_node)? {
                            Some(Binding::Local(register)) => Ok(register),

                            Some(Binding::Upvalue(upvalue_id)) => {
                                // Retrieve the value via Upvalue indirection
                                let dest = self.acquire_reg();
                                self.push(
                                    mem,
                                    Opcode::GetUpvalue {
                                        dest,
                                        src: upvalue_id,
                                    },
                                )?;
                                Ok(dest)
                            }

                            None => {
                                // Otherwise do a late-binding global lookup
                                let name = self.push_load_literal(mem,
ast_node)?;

                                let dest = name; // reuse the register
                                self.push(mem, Opcode::LoadGlobal { dest,
name })?;

                                Ok(dest)
                            }
                        }
                    }
                }
            }

            _ => self.push_load_literal(mem, ast_node),
        }
    }
```

Note that the return type is `Result<Register, RuntimeError>`. That is, a successful *eval* will return a register where the result will be stored at runtime.

In the function body, the match branches fall into three categories:

- keywords literals (`nil`, `true`)
- all other literals
- named variables represented by `Symbol`s

What's in the evaluation of the `Symbol` AST type? Locals, nonlocals and globals!

We can see the generation of special opcodes for retrieving nonlocal and global values here, whereas a local will resolve directly to an existing register without the need to generate any additional opcodes.

Application

To evaluate a function call, we switch over to *apply*:

```
        match *ast_node {
            ...

            Value::Pair(p) => self.compile_apply(mem, p.first.get(mem),
p.second.get(mem)),

            ...
        }
```

This is the evaluation of the `Pair` AST type. This represents, visually, the syntax `(function_name arg1 arg2 argN)` which is, of course, a function call. *Eval* cannot tell us the value of a function call, the function must be applied to it's arguments first. Into *apply* we recurse.

The first argument to `compile_apply()` is the function name `Symbol`, the second argument is the list of function arguments.

Since we included the full `compile_eval()` function earlier, it wouldn't be fair to leave out the definition of `compile_apply()`. Here it is:

```
    fn compile_apply<'guard>(
        &mut self,
        mem: &'guard MutatorView,
        function: TaggedScopedPtr<'guard>,
        args: TaggedScopedPtr<'guard>,
    ) -> Result<Register, RuntimeError> {
        match *function {
            Value::Symbol(s) => match s.as_str(mem) {
                "quote" => self.push_load_literal(mem, value_from_1_pair(mem,
args)?),
                "atom?" => self.push_op2(mem, args, |dest, test|
Opcode::IsAtom { dest, test }),
                "nil?" => self.push_op2(mem, args, |dest, test| Opcode::IsNil
{ dest, test }),
                "car" => self.push_op2(mem, args, |dest, reg|
Opcode::FirstOfPair { dest, reg }),
                "cdr" => self.push_op2(mem, args, |dest, reg|
Opcode::SecondOfPair { dest, reg }),
                "cons" => self.push_op3(mem, args, |dest, reg1, reg2|
Opcode::MakePair {
                    dest,
                    reg1,
                    reg2,
                }),
                "cond" => self.compile_apply_cond(mem, args),
                "is?" => self.push_op3(mem, args, |dest, test1, test2|
Opcode::IsIdentical {
                    dest,
                    test1,
                    test2,
                }),
                "set" => self.compile_apply_assign(mem, args),
                "def" => self.compile_named_function(mem, args),
                "lambda" => self.compile_anonymous_function(mem, args),
                "\\" => self.compile_anonymous_function(mem, args),
                "let" => self.compile_apply_let(mem, args),
                _ => self.compile_apply_call(mem, function, args),
            },

            // Here we allow the value in the function position to be
evaluated dynamically
            _ => self.compile_apply_call(mem, function, args),
        }
    }
```

The `function` parameter is expected to be a `Symbol`, that is, have a *name* represented by a `Symbol`. Thus, the function is `match`ed on the `Symbol`.

Caling nil?

Let's follow the compilation of a simple function: `nil?`. This is where we'll start seeing some of the deeper details of compilation, such as register allocation and

```
                ...
                "nil?" => self.push_op2(mem, args, |dest, test| Opcode::IsNil
{ dest, test }),
                ...
```

The function `nil?` takes a single argument and returns:

- the `Symbol` for `true` if the value of the argument is `nil`
- `nil` if the argument is *not* `nil`.

In compiling this function call, a single bytecode opcode will be pushed on to the `ByteCode` array. This is done in the `Compiler::push_op2()` function. It is named `push_op2` because the opcode takes two operands: an argument register and a result destination register. This function is used to compile all simple function calls that follow the pattern of one argument, one result value. Here is `push_op2()`:

```
fn push_op2<'guard, F>(
    &mut self,
    mem: &'guard MutatorView,
    params: TaggedScopedPtr<'guard>,
    f: F,
) -> Result<Register, RuntimeError>
where
    F: Fn(Register, Register) -> Opcode,
{
    let result = self.acquire_reg();
    let reg1 = self.compile_eval(mem, value_from_1_pair(mem, params)?)?;
    self.bytecode.get(mem).push(mem, f(result, reg1))?;
    Ok(result)
}
```

Let's break the function body down, line by line:

1. `let result = self.acquire_reg();`
    - `self.acquire_reg()`: is called to get an unused register. In this case, we need a register to store the result value in. This register acquisition follows a stack approach. Registers are acquired (pushed on to the stack window) as new variables are declared within a scope, and popped when the scope is exited.
    - The type of `result` is `Register` which is an alias for `u8` - an unsigned int from 0 to 255.
2. `let reg1 = self.compile_eval(mem, value_from_1_pair(mem, params)?)?;`
    - `value_from_1_pair(mem, params)?`: inspects the argument list and returns the argument if there is a single one, otherwise returns an error.
    - `self.compile_eval(mem, <arg>)?`: recurses into the argument to compile it down to a something that can be applied to the function call.
    - `let reg1 = <value>;`: where `reg1` will be the argument register to the

opcode.

3. `self.bytecode.get(mem).push(mem, f(result, reg1))?;`
    - `f(result, reg1)` : calls function `f` that will return the opcode with operands applied in `ByteCode` format.
    - In the case of calling `nil?` , the argument `f` is:
        - `|dest, test| Opcode::IsNil { dest, test }`
    - `self.bytecode.get(mem).push(mem, <opcode>)?;` : gets the `ByteCode` reference and pushes the opcode on to the end of the bytecode array.
4. `Ok(result)`
    - the result register is returned to the `compile_apply()` function

... and `compile_apply()` itself returns the result register to *it's* caller.

The pattern for compiling function application, more generally, is this:

- acquire a result register
- acquire any temporary intermediate result registers
- recurse into arguments to compile *them* first
- emit bytecode for the function, pushing opcodes on to the bytecode array and putting the final result in the result register
- release any intermediate registers
- return the result register number

Compiling `nil?` was hopefully quite simple. Let's look at something much more involved, now.

Compiling anonymous functions

An anonymous function is defined, syntactically, as:

```
(lambda (param1 param2)
  (expr1)
  (expr2)
  (return-expr))
```

There are 0 or more parameters and 1 or more expresssions in the body of the function. The last expression of the body provides the return value.

Function compilation is initiated by *apply*. This is because a function is a compound expression and cannot be reduced to a value by a single *eval*. Here's the line in `compile_apply()` that calls anonymous function compilation:

```
            ...
            "lambda" => self.compile_anonymous_function(mem, args),
            ...
```

Let's look at the type signature of `compile_anonymous_function()`:

```
fn compile_anonymous_function<'guard>(
    &mut self,
    mem: &'guard MutatorView,
    params: TaggedScopedPtr<'guard>,
) -> Result<Register, RuntimeError> {
```

The `params` parameter will be expected to be a `Pair` list: firstly, a list of parameter names, followed by function body expressions.

The return value from is the same as all the other compilation functions so far: `Result<Register>`. The compiled code will return a pointer to the function object in a register.

Here is the function in full:

```
    fn compile_anonymous_function<'guard>(
        &mut self,
        mem: &'guard MutatorView,
        params: TaggedScopedPtr<'guard>,
    ) -> Result<Register, RuntimeError> {
        let items = vec_from_pairs(mem, params)?;

        if items.len() < 2 {
            return Err(err_eval(
                "An anonymous function definition must have at least (lambda
(params) expr)",
            ));
        }

        // a function consists of (name (params) expr1 .. exprn)
        let fn_params = vec_from_pairs(mem, items[0])?;
        let fn_exprs = &items[1..];

        // compile the function to a Function object
        let fn_object = compile_function(mem, Some(&self.vars), mem.nil(),
&fn_params, fn_exprs)?;

        // load the function object as a literal
        let dest = self.push_load_literal(mem, fn_object)?;

        // if fn_object has nonlocal refs, compile a MakeClosure instruction
in addition, replacing
        // the Function register with a Partial with a closure environment
        match *fn_object {
            Value::Function(f) => {
                if f.is_closure() {
                    self.push(
                        mem,
                        Opcode::MakeClosure {
                            function: dest,
                            dest,
                        },
                    )?;
                }
            }
            // 's gotta be a function
            _ => unreachable!(),
        }

        Ok(dest)
    }
```

After converting `Pair` lists to `Vec`s for convenience (wherein parameter names and function body expressions are separated) the process calls into function `compile_function()`, which brings us full circle to *eval*.

In `compile_function()`, below:

1. a `Scope` is instantiated and the parameters are pushed on to this outermost scope.

2. the function body expressions are iterated over, *eval*-ing each one
3. any upvalues that will be closed over as the compiled-function exits and goes out of scope have upvalue instructions generated
4. a `Function` object is returned with all details necessary to running the function in the VM environment

Here is `compile_function()`:

```
    fn compile_function<'guard>(
        mut self,
        mem: &'guard MutatorView,
        name: TaggedScopedPtr<'guard>,
        params: &[TaggedScopedPtr<'guard>],
        exprs: &[TaggedScopedPtr<'guard>],
    ) -> Result<ScopedPtr<'guard, Function>, RuntimeError> {
        // validate function name
        self.name = match *name {
            Value::Symbol(s) => Some(String::from(s.as_str(mem))),
            Value::Nil => None,
            _ => {
                return Err(err_eval(
                    "A function name may be nil (anonymous) or a symbol
(named)",
                ))
            }
        };
        let fn_name = name;

        // validate arity
        if params.len() > 254 {
            return Err(err_eval("A function cannot have more than 254
parameters"));
        }
        // put params into a list for the Function object
        let fn_params = List::from_slice(mem, params)?;

        // also assign params to the first level function scope and give each
one a register
        let mut param_scope = Scope::new();
        self.next_reg = param_scope.push_bindings(params, self.next_reg)?;
        self.vars.scopes.push(param_scope);

        // validate expression list
        if exprs.len() == 0 {
            return Err(err_eval("A function must have at least one
expression"));
        }

        // compile expressions
        let mut result_reg = 0;
        for expr in exprs.iter() {
            result_reg = self.compile_eval(mem, *expr)?;
        }

        // pop parameter scope
        let closing_instructions = self.vars.pop_scope();
        for opcode in &closing_instructions {
            self.push(mem, *opcode)?;
        }

        // finish with a return
        let fn_bytecode = self.bytecode.get(mem);
        fn_bytecode.push(mem, Opcode::Return { reg: result_reg })?;
```

```
        let fn_nonlocals = self.vars.get_nonlocals(mem)?;

        Ok(Function::alloc(
            mem,
            fn_name,
            fn_params,
            fn_bytecode,
            fn_nonlocals,
        )?)
    }
```

Note that in addition to generating upvalue instructions as the compiled-function goes out of scope, the calling compiler function `compile_anonymous_function()` will issue a `MakeClosure` opcode such that a closure object is put in the return register rather than a direct `Function` object reference.

In our language, a closure object is represented by the `Partial` data structure

- a struct that represents a `Function` object pointer plus closed over values and/or partially applied arguments. This data structure was described in the chapter Virtual Machine: Implementation.

Thus ends our tour of our interpreter.

Concluding remarks

In this section, we've looked at a ground-up compiler and virtual machine implementation within a memory-safe allocation system.

There is, of course, much more to explore in the VM and compiler source code. The reader is encouraged to experiment with running and modifying the source.

404 - this chapter has not yet been written

404 - this chapter has not yet been written

404 - this chapter has not yet been written

404 - this chapter has not yet been written