The intermediate representation consists of a few python classes. Data definitions are stored as the names of the constructors along with their arities. Functions are converted to a high level variant of continuation passing style, with an abstraction for case statements. Variable references are converted to either the position of the corresponding argument if it references a local variable, or marked as a global variable which should be resolved once all the symbols are parsed. References to undefined functions will be caught during type checking. The intermediate representation stores no type information, other than the names, as all type information can be discarded after type checking. The type system ensures that all statements and function calls will be well formed. This may change if we ever decide to implement typeclasses, but those are hard and are for much further down the line.

The intermediate representation is then compiled to C. This translation is pretty direct, as every continuation gets compiled into its own function. Continuations used to retrieve the value of a variable or literal are inlined. All other continuations are just compiled directly to an equivalent function in C. The C functions take two arguments: the current continuation, and a list of arguments. The list of arguments represents the current local scope. The current continuation contains a pointer to the next scope, and is also used for recalling the current function after evaluating an object since the program uses lazy evaluation, and for passing the current continuation to the garbage collector, so it knows what to save. Arguments are stored in reverse. The last argument is stored as the first item in a linked list of function arguments. This is because it is faster to prepend.

All runtime objects are THCObjects, which contain a union that can store either a function, an integer, and algebraic object, or a linked list node, which is used by algebraic datatypes or function argument passing. First, the data definitions are compiled. Enums are defined to enumerate the constructors of each datatype, this is needed for case statements to work. Then a constructor function is created for each constructor, except for any 0 arity constructors, which are defined directly as an algebraic THCObject. All the continuation functions are also wrapped with THCObjects at the beginning of runtime. A pointer for the entry point of each particular function is set as well.

Since everything is stateless and immutable, there is no IO at the moment, so the program will simply evaluate the value of main and print that. Unsafe IO may have been added after the time of writing.

Case statements do not support structural pattern matching.

There is currently no type checking at the moment, but the compiler will produce correct output assuming the input is correctly typed.

The example programs are in test/

test.thc contains a basic overview of all the language features, which include data definitions, function definitions (including infix ones), a fairly complicated nested function call, case statements. It should output 5.

functional.thc demonstrates a partial function application, and a recursive map function, and a catamorphism (fold). Using infix operators as partial functions doesn't work at the moment so I needed to define the add function in order to use + as a partial function. It should output 9.

lazytest.thc was created to test the garbage collector and lazy evaluation. Lazy evaluation may or may not be implemented at the time of writing. Without the garbage collector, the program will stack overflow, and terminate in a segmentation fault. With the garbage collector, but without lazy evaluation, it should start eating up memory until it is manually terminated. With lazy evaluation, it should terminate and output 0.

Nestedcase.thc os used to test if nested case statements worked. It should output 100. You can also try changing the arguments to "Just Nothing" or "Nothing"

copy.thc creates a list with two copies of a value, and requires the evaluation of both of them. This was to test whether there were some issues with setting the value of a THCObject on evaluation, as well as having two pointers to one THCObject. It should output 10.

gc.thc is meant to test the garbage collector. The program should run indefinitely and not have a segmentation fault.

./run_tests.sh will run all of these tests. It will terminate gc.thc after 5 seconds.

Garbage Collection. This section is supplemental, and you don't have to read it, but I've been eyeing this gc system for a long time and I wanted to write about it.

THC uses a garbage collection known as Cheney on the MTA. This garbage collection method is also used by the CHICKEN scheme compiler.

Compiling a program CPS in a stack which grows forever, so this must be addressed somehow. Most instances use a trampoline. Continuations normally never return, but with a trampoline, continuations return the next continuation to evaluate along with its arguments, and a loop at the bottom repeatedly calls the next continuation. However, this approach fails to make use of the stack, which is now wasted space.

THC allocates all data onto the stack, and has continuations which do not return. When the stack gets too large (currently a hardcoded limit), a minor garbage collection is performed. The top stack frame is copied to the heap and saved. It then uses cheney's algorithm to copy all live objects to the heap before performing a longjmp to the base loop, which resets the stack. The base loop then continues execution from the saved stack frame.

Cheney's algorithm works as follows. All objects are allocated sequentially on the heap. When all the top stack frame is copied, a pointer to it is created. All objects between this pointer and the end of the heap have references which are possibly on the stack. All references within the live object are then copied to the heap, with the copies on the stack being marked as copied with a pointer to the new object, so any other references can be updated appropriately, rather than copying a second time. The references in the live object are updated to the copies on the heap, and the pointer is incremented. The process ends when the pointer reaches the end of the used heap.

The heap is divided into two halves, one half is always unused. If the used half of the heap runs out of space, a major garbage collection is performed, where cheney's algorithm is used to copy all objects from the used half to the unused half. The unused half is now the used half, and the used half is now the unused half. If the heap runs out of space in this step, the heap is reallocated to double the size.

UPDATE:
Minor collection assumes all only heap objects point at heap objects.
This is no longer the case since lazy evaluation may update heap objects to point at newly evaluated stack objects. Therefore, at the moment, every garbage collection step is a major step. This significantly reduces GC efficiency and I am looking for a way to fix this.