Closures in Pallene.

This document records the specification for implementing closures in the Pallene programming language.

Pallene calling convention.
Pallene compiles to C, using Lua's C-API for communicating with Lua scripts when required. Every exported function in Pallene is compiled to two C functions, each representing a separate entry point. A C entry point, for when a function is called locally from within a Pallene script, and a Lua entry point to be used by Lua scripts.

While the implementation of C Entry point is flexible, Lua requires all C-API functions to follow a single type signature.

```
typedef int (*lua_CFunction) (lua_State *L);
```

To get around this, Pallene mandates separate calling conventions.

- When called from Lua, the arguments are read from the stack and passed over to the C entry point. The return values thus received are pushed back onto the stack.
- When called from Pallene, the arguments are passed as C data types and the C entry point is used directly.

Pallene closures can be implemented as Lua's `CClosure` type. The C Entry points for closures will then be handed down the upvalues they need as extra parameters in the generated code.

Closures in Pallene.
Keeping the above constraints in mind, one possible implementation of closures can be to represent all Pallene closures as Lua's `CClosure` type.

- A closure would contain a `lua_CFunction` along with a single upvalue containing some userdata used by pallene to represent the actual upvalues as C data types.

- Nested function declarations can be lifted up to the global scope and upvalues captured by a closure can then be forwarded to the C entry point as arguments, completing the lambda lift.

- Upvalues that are mutated inside the body of a closure can be "boxed" inside `Udata` objects.

- Multiple `CClosure`s can share the same C and Lua entry points.

For example, let us consider a function **make_adder** that takes an integer and returns a closure that adds that integer to it's argument.

```
function make_adder(x: integer): integer -> integer
    return function (y: integer): integer
       return x + y
    end
end
```

The **make_adder** can be compiled to two Lua and C entry points as usual, with the latter having a return type of `CClosure`.

The inner closure returned by **make_adder** in the above snippet can have the following two entry points:

```
/* C entry point for lambda returned by make_adder */
static lua_Integer lambda_c(lua_State *L,
  Udata *G,
  StackValue *base,
  lua_Integer x, /* upvalue x */
  lua_Integer y /*  param y */);
```

```
/* Lua entry point for lambda returned by make_adder */
static int lambda_lua(lua_State *L);
```

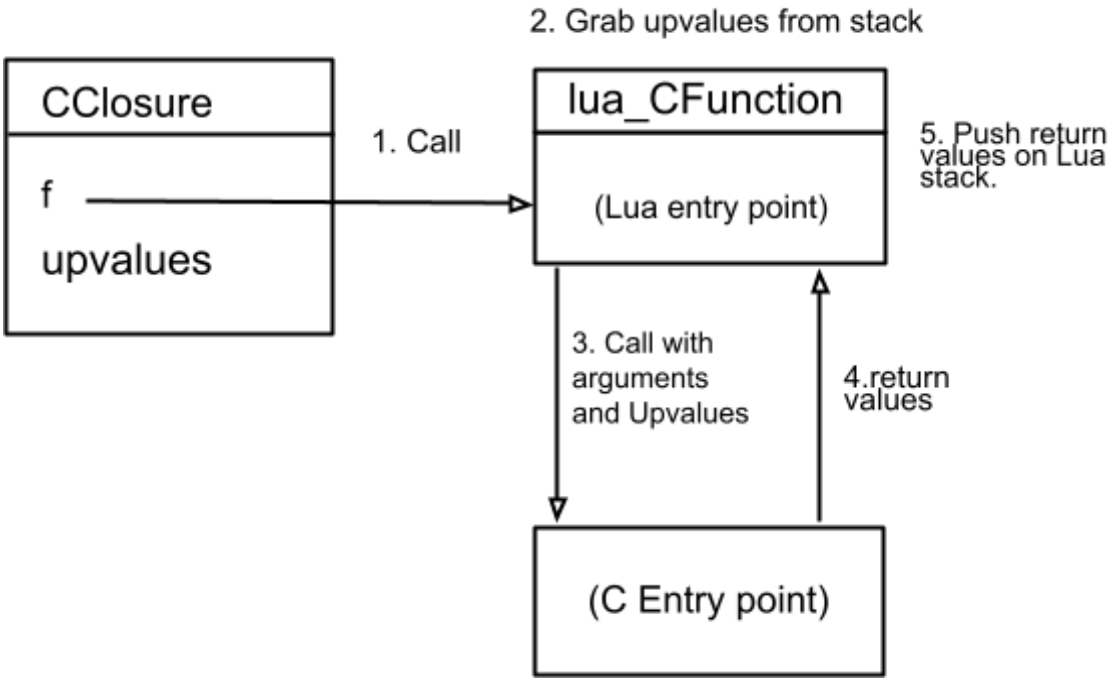
Since the upvalue **x** is never mutated by the closure, it suffices to pass it by value.
The Lua entry point is used when a closure returned by **make_adder** is called from a Lua script, for instance:

```
1 local m = require "adder"
2 local add10 = m.make_adder(10)
3 print(add10(20))
```

When **add10** is called in line 3, the following steps can be followed:

1. `add10` (a `CClosure`) is called, receiving `20` as an argument.
2. Control is handed over to the C-API's `lambda_lua`.
3. `lambda_lua` retrieves the arguments and upvalues from the stack, storing them in C local variables.
4. `lambda_lua` calls `lambda_c`, forwarding the argument `y` and upvalue `x`.
5. `lambda_c` computes the sum and returns it to `lambda_lua`.
6. `lambda_lua` pushes the sum onto the Lua stack and hands control back to the Lua program.

When called from within a local Pallene function, `lambda_c` can be called directly without having to manipulate the stack or calling `lambda_lua` in the process.



Calling a Pallene closure from Lua

For closures that need to store and mutate values the upvalues can be stored in `Udata` "boxes" instead of being passed by value.

Terminology

The specification will be stated assuming the following terminology:

- **Block scope:** The scope inside `do ... end` blocks, bodies of loops and conditionals.

- **Function scope:** The scope present inside the body of a function. The 'outer function scope' of a block is the scope of the function surrounding the current function we are in.

- **Mutable upvalues:** A captured variable whose value is changed (using the '=' operator) either in the body of the function capturing it, or outside. Note that 'mutation' only constitutes reassignment. So in the code snippet given below, `table` is not a mutable upvalue, rather a read-only upvalue as the variable is never reassigned a new value.

```
function outer()
  local t = { a = 1 }
  function inner(x)
      -- this mutates the table, but does not reassign to the
      -- variable 't' itself. The upvalue still holds the same table.
      t.a = x
      return t
  end
  return inner
end
```

In the Lua snippet given below, all closures present in the table `fs` capture the same upvalue `i` with a value of 6. Note that `i` is being mutated after being initialized, therefore it counts as a mutable upvalue.

```
local fs = {}
local i = 1
while i <= 5 do
   fs[i] = function() return i end
   i = i + 1
end
```

For convenience, variables that are initialized after declaration will also be considered mutable. The initial value of any uninitialized variable is considered `nil`. Therefore, `x` in the snippet below is also considered mutable.

```
  local x: integer
  if math.random(1, 10) > 5 then
     x = 10
  else
     x = 20
  end
```

Implementation.

To facilitate the above representation and usage of closures, several changes will have to be made to the compiler passes. This section elaborates upon a possible approach.

Parser
Firstly, the parser would have to allow functions in expression contexts. This can be achieved by adding a new derivation under the expression non-terminal in Pallene's grammar and updating `parser.lua` to reflect the same.
The grammar and the recursive descent parser can mimic Lua's parser from lparser.c. Look for the parts that mention FUNCTION or TK_FUNCTION.

Checker
The type checker will then check the bodies of the function expressions, in addition to regular top-level functions. Nested functions will be able to 'see' variables in outer blocks as long as new blocks are added to the symbol table for every function body.

Recognizing and categorizing upvalues.
At this point, the closure bodies have been type checked. We now need to know the upvalues that a closure captures and differentiate between mutable and read-only upvalues. To facilitate this, a new pass over the compiler can be introduced.

- This new pass will add annotations to the AST. Specifically, it will:
    - Annotate every `ast.Exp.Lambda` with an `upvalues` table containing all the upvalues that this closure captures. Each entry in the upvalues table is either:
        - A local variable from the enclosing function or block scope.
        - An upvalue captured by the enclosing function.
    - Mark a captured upvalue as either '`local`' or '`foreign`'.
- The above can be achieved in a single pass over the AST. To distinguish the upvalues, the `_name` field of the `ast.Var.Name` nodes can be examined to determine the declaration scope.

IR
Every Pallene closure is a `CClosure`. The Lua C-API provides the function `luaF_newClosure` to create Lua closure objects. To represent this in the IR, a `NewClosure` IR instruction can be added.

Reading from and writing to upvalues can be done using `setuvalue` and `getuvalue` helpers provided by Lua. This might not necessarily require new IR instructions, and can be desugared in the IR generation pass itself.

IR generation and lambda lifting.
The nested closures will have to be lifted to the top-level scope in the C code.
The lambda lift can be performed in the IR generation pass.
Whenever a nested function is found in a block's statement list or inside an expression, two important actions that need to be performed are:

- Use the `NewClosure` IR instruction to create the closure, passing as an operand the number of upvalues associated with it.
- Add a local function to the top-level `ir.Module`.
- Whenever an upvalue is read from or written to, generate the corresponding IR.
- When calling closures using the `CallStat` instruction, also pass along the upvalues.

Code generation
After all the above work has been laid out, the code generation can be modified with the following changes.

- Generate C code for the `NewClosure` IR instruction.
- Add extra parameters to hold upvalues in a closure's C entry point.

Representing upvalue boxes in C.
Mutable upvalues cannot be passed by value as extra parameters in C.
To retain the mutability of these upvalues, they need to be boxed inside some kind of table or wrapper. The 'box' in C can be represented in two ways:

1. **As `Udata` objects**. Wherein the mutable upvalue will be a TValue associated with the Udata.
2. **As Pallene records** inside `Udata`. Since Pallene records get compiled to C structs, the compiler will need to figure out the 'shape' of a closure. A closure's shape in this context, is the type signature of the Pallene record required to hold all it's upvalues.

To illustrate the point made about shapes, consider an example.

```
function make_closure(): integer -> (integer, integer)
      local x: integer = 1;
      local y: integer = 2;

      return function swap(): (integer, integer)
            x, y = y, x
            return x, y
      end
end
```

make_closure returns a closure which captures two upvalues x and y. Both these upvalues are mutable as they appear on the LHS of an assignment. For the closure 'swap' returned by make_closure the shape is:

```
record shape
  x: integer
  y: integer
end
```

When mutable upvalue boxes are represented using Pallene records, the compiler will need to figure the corresponding shape for every closure that captures at least 1 mutable upvalue.

To be able to use C structs as boxes, the type signature for the shape of each closure will have to be inferred by the additional pass over the compiler that comes after type checking.