Lecture 6: Interpreter runtime structures 2

David Hovemeyer

September 19, 2022

601.428/628 Compilers and Interpreters

Outline

- Closures
- Garbage collection
- Bytecode interpreters
- Thoughts on interpreter implementation

Closures

Closures

Many languages support *closures*, a.k.a. anonymous functions, lambdas

Basic idea: the closure retains a pointer to its parent environment, i.e., the environment in which it was created at runtime

- This may imply that the lifetime of the parent environment is extended to be at least as long as the lifetime of the closure

Closure example

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```

Closure example

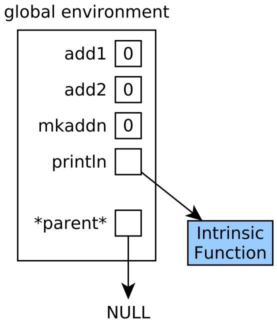
```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));                  -- prints 2
println(add2(1));
```

Closure example

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));              -- prints 3
```

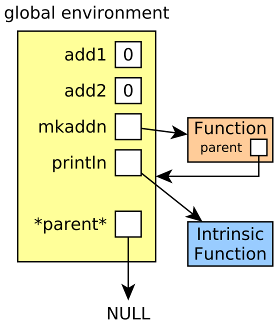
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



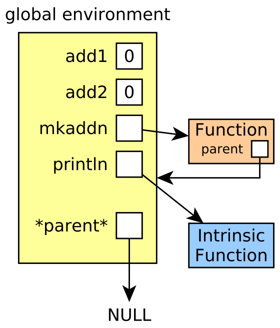
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



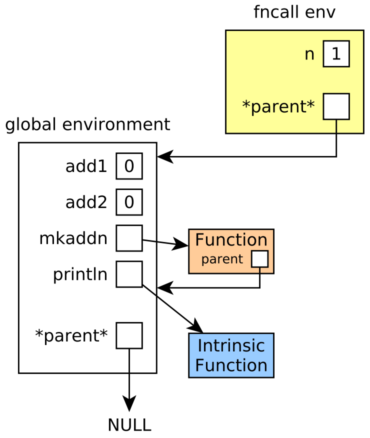
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



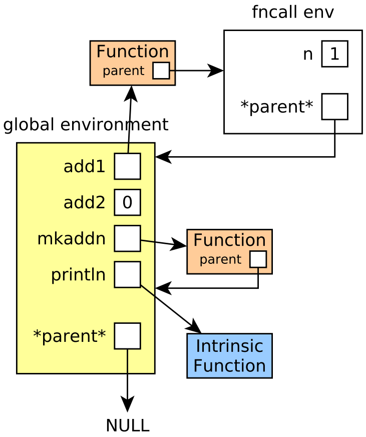
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



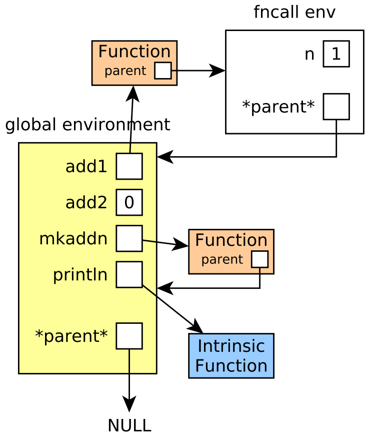
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



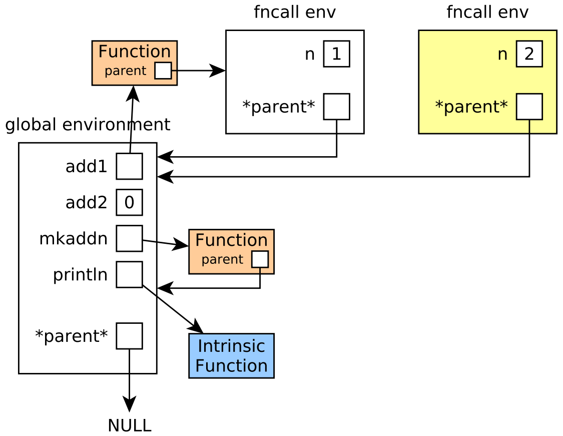
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



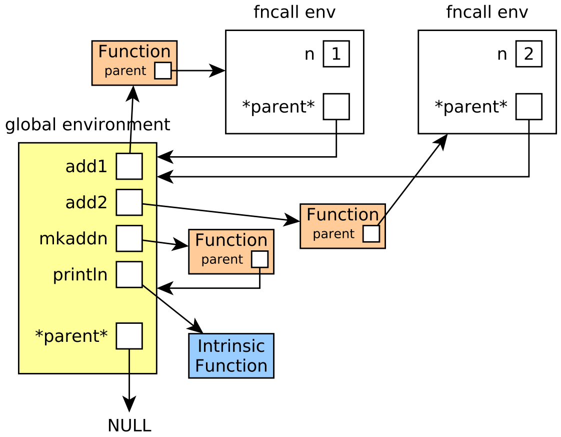
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



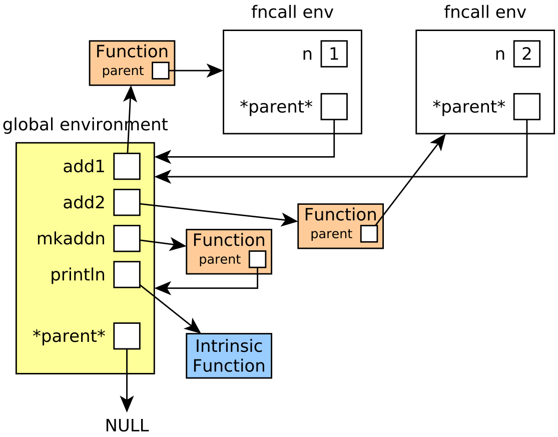
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



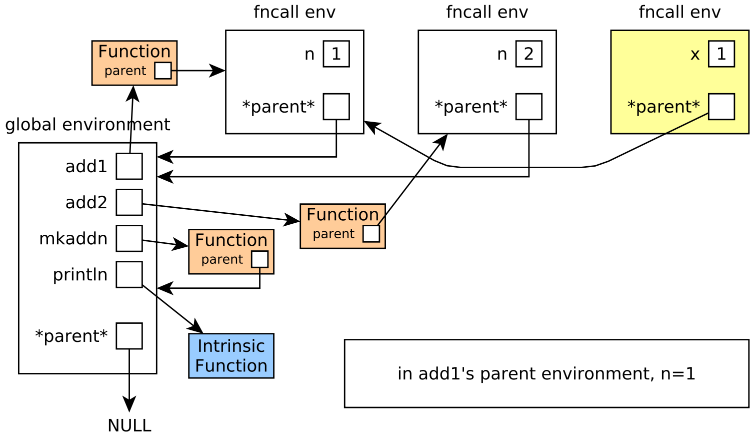
Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



Closure environments

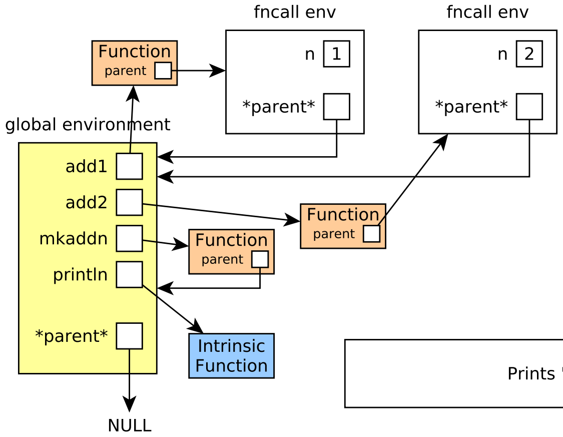
```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



in add1's parent environment, n=1

Closure environments

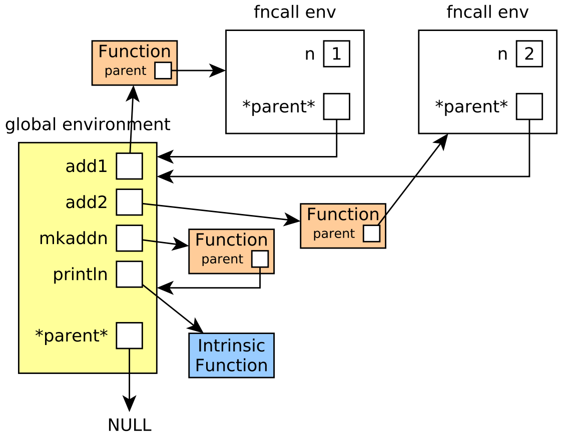
```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



Prints "2"

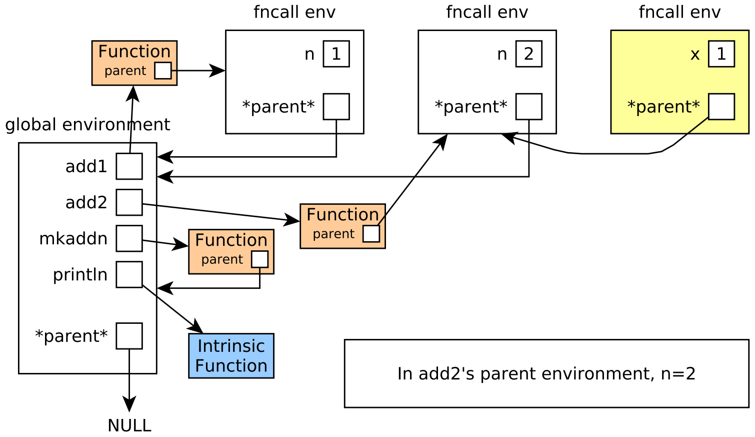
Closure environments

```
  function mkaddn(n) {
    function(x) { x + n; };
  }

  var add1;
  var add2;
  add1 = mkaddn(1);
  add2 = mkaddn(2);

  println(add1(1));
→ println(add2(1));
```



Closure environments

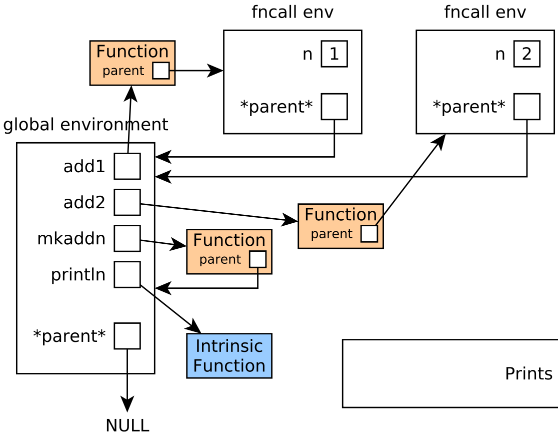
```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



In add2's parent environment, n=2

Closure environments

```
function mkaddn(n) {
  function(x) { x + n; };
}

var add1;
var add2;
add1 = mkaddn(1);
add2 = mkaddn(2);

println(add1(1));
println(add2(1));
```



Prints "3"

Implementing closures

A few possible implementation techniques:

- Dynamically-allocate function call environments
  - Closure values retain a pointer to parent environment
  - Could use reference counting to know when to delete dynamically-created environments
    - Environment is destroyed when there are no remaining references to it
- Closure retains a *copy* of its parent environment (and grandparent, etc.)
  - Or, copies of just the variables that are actually referenced by the body of the closure function

Garbage collection

Dynamically allocated values

- We noted last time that runtime values may require dynamically-allocated storage
  - Strings, vectors, list nodes, objects, etc.
- How to ensure that dynamically allocated memory gets reclaimed when no longer used?
- A couple standard approaches:
  - Reference counting
  - Garbage collection

Reference counting

- All dynamically allocated objects have a *reference count* field
  - Is just an integer indicating how many pointers are pointing to the object
- Language runtime must take care to increment and decrement references counts
  - C++ smart pointers can help a lot with this
  - `Value` class in Assignments 1&2 has this role
- When reference count reaches 0, deallocate
- Problem: object graphs with cycles can't be reclaimed

Reclaiming cyclical structures

How to address this problem?

- Do nothing and don't worry about it
- Periodically run a garbage collection algorithm (which generally have no trouble reclaiming cyclical garbage)
- Support weak references (which don't increment the pointed-to object's reference count)
  - Example: in a tree where children keep a pointer to their parent, make it a weak reference
  - One challenge is to invalidate the weak reference when the object's reference count reaches 0

Garbage collection

- Language runtime keeps track of references to dynamic objects
- Periodically, it determines which objects are reachable
  - Unreachable objects are reclaimed
- There are *many* ways to do this! (Tons of research, we could do an entire course on this topic)
  - We'll briefly discuss two

Root set

- The *root set* of references are the starting point for determining which objects are reachable
- It consists of:
    - Objects referenced by global variables
    - Objects referenced by the activation records (i.e., function call environments) of currently-executing functions (on the call stack)
- Objects not directly or indirectly reachable from the root set can be assumed to be garbage

Assumptions

- The garbage collector can find all of the dynamically allocated objects
- Given a pointer to an object, the garbage collector knows what pointers to other objects are stored in it

Mark and sweep

- Starting from the root set, do a graph traversal to find all reachable objects, and mark them as “live”
- Traverse all dynamically allocated objects, and reclaim the memory of those not marked as alive
    - Also, clear the “live”‘ mark on objects that are still alive

Copying

- Heap is divided into *semispaces*
- New objects are allocated in the current semispace
- To collect garbage:
  - Starting from the root set, do a graph traversal of reachable objects
  - For each live object, copy it into the other semispace (keeping track of mapping from old location to new location)
  - Once all objects are copied, update all pointers in root set and live objects to reflect the updated object locations
  - Switch semispaces

Bytecode interpreters

Pros and cons of AST-based interpreters

- Assignments 1 and 2 involve implementing an AST-based interpreter
  - The AST is the program representation used to execute the source program
- Advantages of AST-based interpreters:
  - Easy to implement
- Disadvantages of AST-based interpreters:
  - Slow
  - Poor cache locality

Bytecode interpreters

- Another common approach is to implement a *bytecode interpreter*
- Functions are translated into *bytecode*
  - Essentially a machine language for a software-defined CPU
- Requires *compilation* of source to bytecode instructions (per-function)
  - Bytecode instructions can correspond closely to constructs in AST
  - So, this compilation process can be relatively straightforward

A bytecode language

Concepts:

- A function is encoded as a sequence of bytes
- A function has a *string pool* containing literal string values needed by instructions
- Each instruction starts with an *opcode* byte
- The opcode byte may followed by additional bytes to encode additional information about the instruction:
    - Literal integer (N)
    - Local variable number ($Lnum$)
    - Index of a string in the string pool ($Sidx$)
    - Index of an instruction within the function (for branches) ($InsIdx$)

Stack-based vs. register-based bytecodes

- An important concern in a bytecode language is how to store and refer to temporary values
- E.g., before an operator is applied to its operands, they must be evaluated, and their values stored somewhere
- Two main approaches:
  - Stack-based: temporary values are pushed onto an *operand stack*
  - Register-based: temporary values are stored in “registers”
- These slides will present a very simple stack-based bytecode language
  - Modeled on Java bytecode

Instruction set

iconst	*N*	*Push integer constant N onto operand stack*
strconst	*Sidx*	*Push string constant onto operand stack*
ret		*Return from function*
add		*Pop right operand, pop left operand, compute and push sum*
sub		*Pop right operand, pop left operand, compute and push difference*
mul		*Pop right operand, pop left operand, compute and push product*
div		*Pop right operand, pop left operand, compute and push quotient*
pop		*Remove top operand from stack*
dup		*Push duplicate of top operand*
getvar	*Sidx*	*Get named variable (from outer environment), push on stack*
setvar	*Sidx*	*Pop value from stack, store in named variable (from outer environment)*
ldlocal	*Lnum*	*Get local variable, push its value on stack*
stlocal	*Lnum*	*Pop value from stack, store in local variable*
cmplt		*Pop right operand, pop left operand, push boolean lhs<rhs*
cmplte		*Pop right operand, pop left operand, push boolean lhs<=rhs*
cmpgt		*Pop right operand, pop left operand, push boolean lhs>rhs*
cmpgte		*Pop right operand, pop left operand, push boolean lhs>=rhs*
cmpeq		*Pop right operand, pop left operand, push boolean lhs==rhs*
cmpneq		*Pop right operand, pop left operand, push boolean lhs!=rhs*
jmpt	*InsIdx*	*Pop boolean, jump to target instruction if true*
jmpf	*InsIdx*	*Pop boolean, jump to target instruction if false*
jmp	*InsIdx*	*Unconditional branch to target instruction*
call	*Sidx*	*Call function named by string constant*

Local variables

- The bytecode interpreter defines *local variables* to serve as storage for parameters and local variables within the function body
- Local variables are numbered starting from 0
  - Local variable 0 is the first parameter
- By analyzing variable declarations in the function AST, the bytecode compiler can assign each local variable in the body of the function to a local variable number
  - Local variables with non-overlapping lifetimes can use the same local variable number

Bytecode compilation

- Code generation for a stack-based instruction set is incredibly simple
- Basic idea: code generated to evaluate an expression pushes the result value onto the stack
- Evaluating a binary operator: recursively generate code for left and right subexpressions (pushing their values onto the stack), then emit a computation instruction (`add`, `sub`, etc.) which will pop the operands, then push the result of the computation
- This strategy works for arbitrarily-complicated expressions!

Evaluating $1 + 2$

```
iconst   1
iconst   2
add
```

Evaluating 3 + (4 * 5)

```
iconst   3
iconst   4
iconst   5
mul
add
```

Observation

Generating stack-based bytecode instructions is essentially the same idea as translating expressions into *postfix* form, where each operator follows its operands

Statement lists

To evaluate a sequence of statements:

- Generate code for the statement
- If the statement was not the last statement in the sequence, emit a `pop` instruction

The result of evaluating the last statement is left on the stack

Code generation

Helper functions for code generation:

- `emit` appends a byte to the bytecode
- `emit_i16` appends a 16 bit integer (as two bytes)
- `emit_i32` appends a 32 bit integer
- `intern` returns the index in the string pool of a specified string value, adding it to the pool if it is not present already
- `set_i16` modify a 16 bit integer at specified offset in the bytecode (helpful for resolving branch targets)

Code generation function

```
void BytecodeCompiler::gen_code(Node *ast, Value env) {
  int ast_tag = ast->get_tag();
  switch (ast_tag) {
    ...lots of cases...
  }
}
```

Note that `env` is an `Environment` representing the current scope, wrapped in a `Value`. Its job is to keep track of which local variables exist, and which local variable number each one has.

Generating code for an integer literal

```
case AST_INT_LITERAL:
  emit(OP_ICONST, ast->get_loc());
  emit_i32(std::stoi(ast->get_str()));
  break;
```

The integer literal's lexeme is converted to an integer value and encoded into the `iconst` instruction.

Generating code for a variable reference

```
case AST_VARREF:
  {
    Environment::Binding var_binding = env.get_env()->find(ast->get_str());
    if (var_binding.is_valid()) {
      int varnum = var_binding.get_value().get_ival();
      emit(OP_LDLOCAL, ast->get_loc());
      emit_i16(varnum);
    } else {
      emit(OP_GETVAR, ast->get_loc());
      emit_i16(intern(ast->get_str(), ast->get_loc()));
    }
  }
  break;
```

Local variables emit an `ldlocal` instruction, variables outside the scope of the function emit `getvar` instruction (which handles the case where a variable in a scope enclosing the function is accessed.)

Generating code for binary expressions

```
case AST_ADD:
case AST_SUB:
case AST_MULTIPLY:
case AST_DIVIDE:
case AST_LT:
case AST_LTE:
case AST_GT:
case AST_GTE:
case AST_EQ:
case AST_NEQ:
  gen_code(ast->get_kid(0), env);
  gen_code(ast->get_kid(1), env);
  emit(binop(ast_tag), ast->get_loc());
  break;
```

The `binop` function maps an AST tag (of a binary operator) to the corresponding bytecode instruction.

Generating code for a variable definition

```
case AST_VARDEF:
  {
    Environment::Binding localvar_binding =
      env.get_env()->create(ast->get_kid(0)->get_str());
    localvar_binding.set_value(Value(m_cur_num_locals));
    m_cur_num_locals++;
    if (m_cur_num_locals > m_max_num_locals)
      m_max_num_locals = m_cur_num_locals;
    // push dummy evaluation result
    emit(OP_ICONST, ast->get_loc());
    emit_i32(0);
  }
  break;
```

The next unused local variable number is assigned for the new local variable. (Note that every statement must push one value, hence the `iconst 0`.)

Statement list (block)

```
case AST_STATEMENT_LIST:
  {
    Value block_env(new Environment(env));
    for (auto i = ast->cbegin(); i != ast->cend(); ++i) {
      if (i != ast->cbegin())
        emit(OP_POP);
      gen_code(*i, block_env);
    }
    // any local variables defined in this block are
    // now no longer needed
    m_cur_num_locals -= block_env.get_env()->get_size();
  }
  break;
```

Each statement list (block) gets a nested `Environment` so that it may define local variables. These variables cease to exist when control exits the block.

If/else statement

```
case AST_IF_ELSE:
  {
    gen_code(ast->get_kid(0), env); // gen code for condition
    emit(OP_JMPF);
    unsigned pc_target_iffalse = m_bytecode.size();
    emit_i16(0);
    gen_code(ast->get_kid(1), env); // if true part
    emit(OP_POP);
    emit(OP_JMP);
    unsigned pc_target_done = m_bytecode.size();
    emit_i16(0);
    set_i16(pc_target_iffalse, m_bytecode.size());
    gen_code(ast->get_kid(2), env); // if false part
    emit(OP_POP);
    set_i16(pc_target_done, m_bytecode.size());
    emit(OP_ICONST); // done, push dummy value
    emit_i32(0);
  }
  break;
```

The main complication is that the byte index of a control target isn't known until after the point where the branch instruction is emitted. (So, generate a dummy index and then fix it later.)

While loop

This is left as an exercise for the reader ☺

A while loop is like an if statement that repeats, so not too hard to implement.

Function calls

```
case AST_FNCALL:
  {
    // evaluate argument expressions
    Node *arglist = ast->get_kid(1);
    for (auto i = arglist->cbegin(); i != arglist->cend(); ++i) {
      gen_code(*i, env);
    }
    // generate code to evaluate function
    gen_code(ast->get_kid(0), env);
    // emit call instruction
    emit(OP_CALL, ast->get_loc());
    emit_i16(int16_t(arglist->get_num_kids()));
  }
  break;
```

Fairly straightforward: generate code to evaluate and push arguments, then generate code to look up and push the function value. The call will clear argument and function values, then push the function's result.

Example bytecode translation

```
function add(x, y) {
  x + y;
}
```

```
Function 'add'
Parameters: x, y
Code:
    0   ldlocal 0
    3   ldlocal 1
    6   add
    7   ret
```

Example bytecode translation

```
function fib(n) {
  var result;
  if (n < 2) {
    result = n;
  } else {
    result = fibhelp(0, 1, n - 1);
  }
  result;
}
```

```
Function 'fib'
Parameters: n
Code:
     0    iconst 0
     5    pop
     6    ldlocal 0
     9    iconst 2
    14    cmplt
    15    jmpf 29
    18    ldlocal 0
    21    dup
    22    stlocal 1
    25    pop
    26    jmp 59
    29    iconst 0
    34    iconst 1
    39    ldlocal 0
    42    iconst 1
    47    sub
    48    getvar fibhelp
    51    call 3
    54    dup
    55    stlocal 1
    58    pop
    59    iconst 0
    64    pop
    65    ldlocal 1
    68    ret
```

Example bytecode translation

```
function fibhelp(a, b, n) {
  var result;
  if (n == 0) {
    result = b;
  } else {
    result = fibhelp(b, a + b, n - 1);
  }
  result;
}
```

```
Function 'fibhelp'
Parameters: a, b, n
Code:
     0   iconst 0           55   stlocal 3
     5   pop                58   pop
     6   ldlocal 2          59   iconst 0
     9   iconst 0           64   pop
    14   cmpeq              65   ldlocal 3
    15   jmpf 29            68   ret
    18   ldlocal 1
    21   dup
    22   stlocal 3
    25   pop
    26   jmp 59
    29   ldlocal 1
    32   ldlocal 0
    35   ldlocal 1
    38   add
    39   ldlocal 2
    42   iconst 1
    47   sub
    48   getvar fibhelp
    51   call 3
    54   dup
```

Bytecode execution

```
Value Interpreter::execute_bytecode(Value fn_val, Value env) {
  BytecodeFunction *func = fn_val.get_bytecode_function();
  const std::vector<uint8_t> &bytecode = func->get_bytecode();
  const std::vector<std::string> &strpool = func->get_strpool();

  std::vector<Value> locals(func->get_num_locals());
  // ...copy argument values to locals...

  std::vector<Value> stack;
  unsigned pc = 0; // "Program Counter"
  unsigned op_pc;  // PC value of current opcode
  int32_t lhs, rhs;
  int16_t off;
  bool done = false;

  // ...bytecode execution loop...

  return stack.back(); // result is on top of stack
}
```

Executing bytecode instructions

```
while (!done) {
  op_pc = pc++;
  uint8_t opcode = bytecode[op_pc];

  switch (opcode) {
    ...lots of cases...
  }
}
```

Integer and string literals

```
case OP_ICONST:
  pc = decode_i32(bytecode, pc, lhs);
  stack.push_back(Value(lhs));
  break;

case OP_STRCONST:
  pc = decode_i16(bytecode, pc, off);
  assert(off >= 0);
  stack.push_back(Value(new String(strpool[off])));
  break;
```

Binary operators

```
#define EXECUTE_BINOP(op) \
do { \
  check_binop_operands(op_pc, stack, func); \
  rhs = stack.back().get_ival(); \
  stack.pop_back(); \
  lhs = stack.back().get_ival(); \
  stack.pop_back(); \
  stack.push_back(Value(lhs op rhs)); \
} while (0)
```

Binary operators

```
case OP_ADD:
  EXECUTE_BINOP(+);
  break;

case OP_SUB:
  EXECUTE_BINOP(-);
  break;

case OP_MUL:
  EXECUTE_BINOP(*);
  break;
```

Handling function calls, closures

- Handling function calls:
  - Could have the interpreter make a recursive call to `execute_bytecode` (easiest option)
  - Could build support for function calls/returns into the bytecode interpreter (more difficult, but likely better performance this way)
- Closures:
  - In bytecode loop shown above, locals are a `vector` in the stack frame of `execute_bytecode`
  - If a closure is created, how to allow local variables to become part of the closure environment?

Thoughts on interpreter implementation

Some general advice

- Get the parser working first
- Visualize your tree
- Use assertions
- Testing: start with the simplest possible tests, then increase complexity incrementally
- `gdb`