CMSC 245 Wrap-up

This class is about understanding how programs work

To do this, we're going to have to learn how a computer works

Learned a **ton** in the class

Lexical vs. Dynamic Scoping

Regexp

Parsing

Closures

Objects

Heaps

Racket

Functions

Method dispatch

C++

Stacks

Garbage collection

JS

Classes

Assembly

Calling conventions

To apologize for making you write so much I wrote 732 lines of C++ yesterday

- Today we’re going to design an interpreter
- Our source language will be a **subset** of Scheme
  - Numbers, variables, if, lambdas, let, begin, set!
- We’ll write our own lexer, grammar, and parser
  - Starting from what you already wrote in labs
- Our interpreter will use **data structures** from the course
- And will include **garbage collection** under the hood

Raw Text	
Lexer	Regex
Parser	CFG
AST	C++ (Sub)classes
Interpreter	Methods on AST

Raw Text	
Lexer	`scanner.l` ~20 lines of code
Parser	`parser.cc` —150 lines of code
AST	`interpreter.h` —220 lines of code
Interpreter	`interpreter.cc` —160 lines of code

Lesson

Sometimes the most best way to do something is to find someone who's already done it for you…

Symbol Table

Garbage Collector

HAMT

Boehm GC

HAMT

Hash **A**rray-**M**apped **T**rie

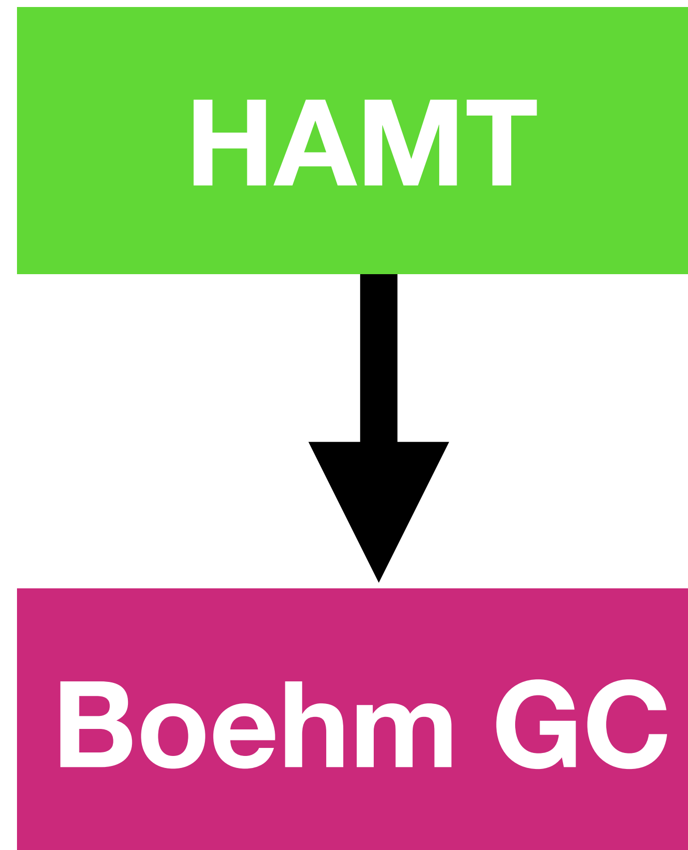
Think of this is as a hash table that is "quick" to copy

Boehm GC

High-performance GC for C

We'll use this to make it so our interpreter is automatically garbage collected

We’ll have our hash table **use** the GC under the hood



So when we put things into HAMT, they are **automatically** GC’d

The grammar…

(In EBNF, allows E+)

```
START -> E $
E -> number
E -> identifier
E -> ( OP E+ )
E -> ( begin E+ )
E -> ( lambda (ID+) E )
E -> ( set! x E )
E -> ( E+ )
OP --> +|-|*|=
```

The grammar…

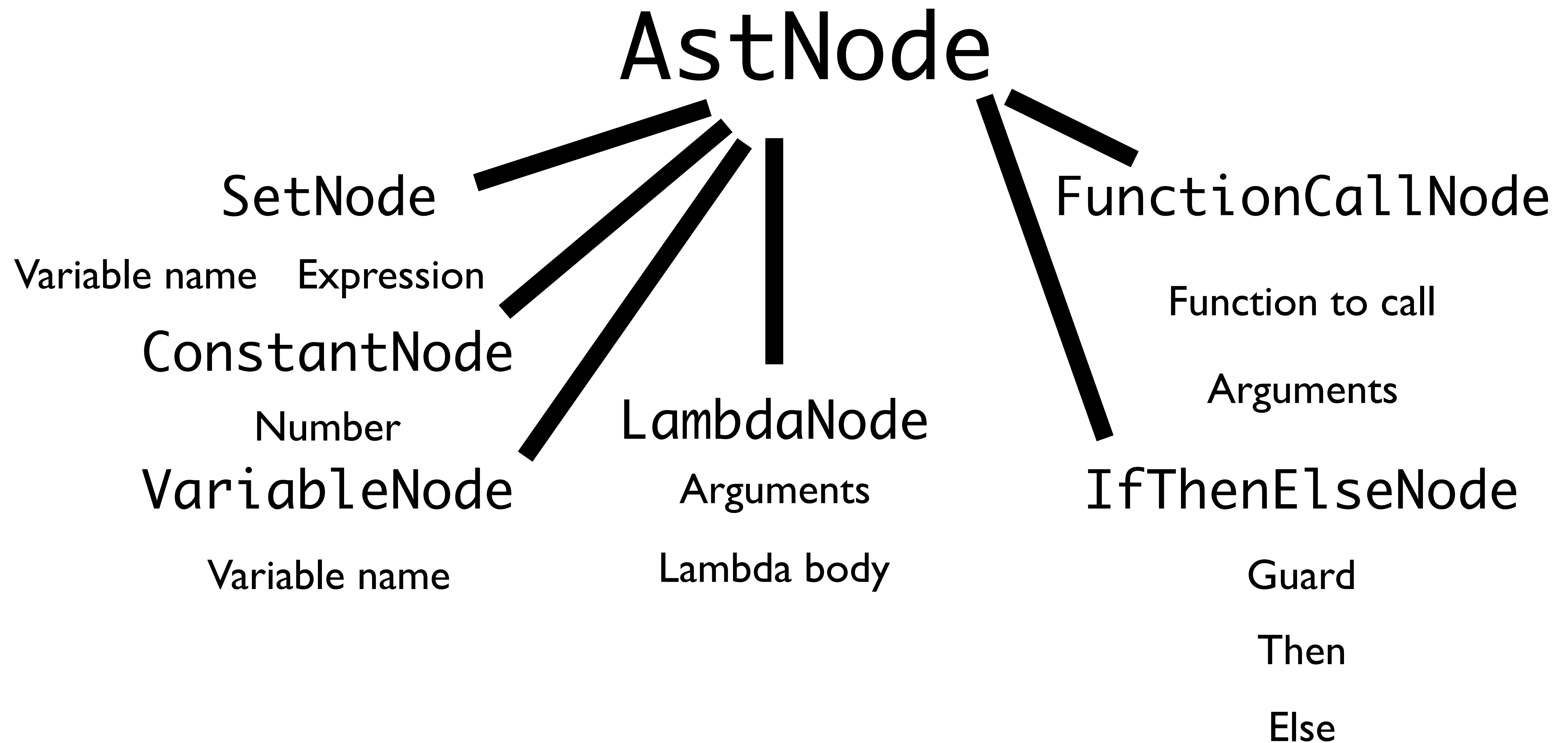
(In EBNF, allows E+)

```
START -> E $
E -> number
E -> identifier
E -> ( OP E+ )
E -> ( begin E+ )
E -> ( lambda (ID+) E )
E -> ( set! x E )
E -> ( E+ )
OP --> +|-|*|=
```

We'll dig into this in a few mins

AST

Idea: Represent using subclasses



The lexer…

```
[ \t]           { continue; }
[\n]            { tokenCount++; return NEWLINE; }
";".*           { continue; }
"("             { tokenCount++; return LPAREN; }
")"             { tokenCount++; return RPAREN; }
"+"             { tokenCount++; return PLUS; }
"-"             { tokenCount++; return MINUS; }
"*"             { tokenCount++; return TIMES; }
"lambda"        { tokenCount++; return LAMBDA; }
"let"           { tokenCount++; return LET; }

"<EOF>"         { tokenCount++; return END_OF_INPUT; }

-?{digit}+      { tokenCount++; return INT; }
{identifier}    { tokenCount++; return IDENTIFIER; }

.               { scannerError(); continue; }
```

The Parser

I started from code we gave you in Lab 5…

But I cheated because it's not LL(1)

See `parser.cc`

(5 minute tour)

The Symbol Table

Is a dictionary that takes strings to addresses in the heap

Means most things are stored on heap

Necessitates GC (we'll discuss next)

The Symbol Table

Wrapper for strings   Representation of pointers

```
typedef hamt<HashedString, Address> environment;
```

HAMT is a dictionary

Two methods:

- Get: Takes a dictionary and key, gives us address
  - Which we then look up in heap
- Insert: Takes a dictionary, key, and value
  - Returns a **new** dictionary

The Heap

Stores two possible things:

- Plain old numbers
- Closures
- You could add other things (strings, etc..)

To find x, we look up address in symbol table, then use that address to look up through the heap

Symbol tables

Wrapper around std::string

```
typedef hamt<HashedString, Address>
        environment;
```

Symbol tables

Wrapper around std::string

```
typedef hamt<HashedString, Address>
            environment;
```

Closures

```
struct Closure {
  AstNode *function;
  environment *environment;
};
```

Symbol tables

Wrapper around std::string

```
typedef hamt<HashedString, Address>
              environment;
```

Closures

```
struct Closure {
  AstNode *function;
  environment *environment;
};
```

Values

```
typedef variant<int, Closure> value;
```

Variant is **new** in C++17

Container that allows me to store anything from any set of types

```
get<int>(x) // gets the integer value assuming
            // x is an integer
```

Symbol tables

Wrapper around std::string

```
typedef hamt<HashedString, Address>
                environment;
```

Closures

```
struct Closure {
  AstNode *function;
  environment *environment;
};
```

Values

```
typedef variant<int, Closure> value;
```

Heap

```
hamt<Address, value> *heap
      = new hamt<Address, value>();
```

```
Address *putValueInHeap(value v) {
  heapSize++;
  Address* addr =
    new ((Address*)GC_MALLOC(sizeof(Address)))
          Address({heapSize});
  value * val =
      new ((value*)GC_MALLOC(sizeof(value))) value(v));
  heap =
    const_cast<hamt<Address, value> *>
                   (heap->insert(addr,val));
  return addr;
}

value getValueFromHeap(Address a) {
  return *heap->get(&a);
}
```

```
Address *putValueInHeap(value v) {
  heapSize++;
  Address* addr =
    new ((Address*)GC_MALLOC(sizeof(Address)))
          Address({heapSize});
  value * val =
      new ((value*)GC_MALLOC(sizeof(value))) value(v));
  heap =
    const_cast<hamt<Address, value> *>
                     (heap->insert(addr,val));
  return addr;
}
```

Tracks the object with GC

```
value getValueFromHeap(Address a) {
  return *heap->get(&a);
}
```

Every `AstNode` implementation has a method

`execute : symbol table -> value`

There is a "top level" symbol table where global variables go

(top of `interpreter.cc`)

REPL

```
int main() {
  while (true) {
    cout << "> ";
    AstNode *AST = parseE();;
    executeToplevelAst(AST);
  }
};
```

```
void executeToplevelAst(AstNode *node) {
  value result = node->execute(globalEnvironment);
  if (holds_alternative<int>(result)) {
    cout << get<int>(result) << endl;
  } else {
    get<Closure>(result).function->render();
    cout << endl;
  }
}
```