Implementing a Language

Given type-checked AST program representation:
- might want to run it
- might want to analyze program properties
- might want to display aspects of program on screen for user
- ...

To run program:
- can interpret AST directly
- can generate target program that is then run recursively

Tradeoffs:
- time till program can be executed (turnaround time)
- speed of executing program
- simplicity of implementation
- flexibility of implementation

Interpreters

Create data structures to represent run-time program state
- **values** manipulated by program
- **activation record** (a.k.a. stack frame) for each called method
  - **environment** to store local variable bindings
  - pointer to lexically-enclosing activation record/environment (**static link**)
  - pointer to calling activation record (**dynamic link**)

EVAL loop executing AST nodes

Pros and cons of interpretation

+ simple conceptually, easy to implement
+ fast turnaround time
+ good programming environments
+ easy to support fancy language features

- slow to execute
  - data structure for value vs. direct value
  - variable lookup vs. registers or direct access
  - EVAL overhead vs. direct machine instructions
  - no optimizations across AST nodes

An interpreter for MiniJava

In `Evaluator` subdirectory:

Data structure to represent run-time values: `Value` hierarchy
- analogous to `ResolvedType` hierarchy

```
Value
    IntValue
    BooleanValue
    ClassValue
    NullValue
```

Data structure to store `Values` for each variable: `Environment` hierarchy
- analogous to `SymbolTable` hierarchy

```
Environment
    GlobalEnvironment
    NestedEnvironment
        ClassEnvironment
        CodeEnvironment
            MethodEnvironment
```

`evaluate` methods for each kind of AST class

Activation records

Each call of a method allocates an **activation record** (instance of `MethodEnvironment`)

- mapping from names to `Value`s, for each formal and local variable in that scope (**environment**)
- lexically enclosing activation record (**static link**)
- calling activation record (**dynamic link**)

Each "invocation" of a nested block allocates a `CodeEnvironment`

- environment + static link=dynamic link

Each declaration of a class allocates a `ClassEnvironment`

- set of methods (to support run-time method lookup)
- static link (to global environment)
- *not* instance variable values!
  - instance variable values stored in class instances, i.e., in `ClassValue`s

Activation records vs. symbol tables

For each method/nested block scope in a program:

- exactly one symbol table, storing **types** of names
- possibly many activation records, one per invocation, each storing **values** of names

For recursive procedures, can have several activation records for same procedure on stack simultaneously

All activation records have same "shape," described by single symbol table

Example

```
...

class Fac {
  public int ComputeFac(int num) {
    int numAux = 0;
    if (num < 1) {
      numAux = 1;
    } else {
      numAux = num * this.ComputeFac(num-1);
    }
    return numAux;
  }
}
```


Generic evaluation algorithm

Parallels the generic typechecking algorithm

To evaluate a program, recursively evaluate each of the nodes in the program's AST, each in the context of the environment for its enclosing scope

- on the way down, create any nested environments & context needed
- recursively evaluate child subtrees
- on the way back up, compute the parent's result/effect from the children's results
- parent controls order of evaluation of children, whether to evaluate children

Each AST node class defines its own `evaluate` method, which fills in the specifics of this recursive algorithm

Generally:

- declaration AST nodes add *value* bindings to the current environment
- statement AST nodes evaluate (some of) their subtrees
- expression AST nodes evaluate their subtrees and compute & return a result value

Some key AST evaluation operations

```
void Program.evaluate()
  throws EvalCompilerExn;
```

- evaluate the whole program:
  - evaluate each of the class declarations
  - invoke the main class's main method

```
void ClassDecl.evaluateDecl(GlobalEnvironment)
  throws EvalCompilerExn;
```

- evaluate a class declaration

```
void Stmt.evaluate(CodeEnvironment)
  throws EvalCompilerExn;
```

- evaluate a statement in the context of the given environment

```
Value Expr.evaluate(CodeEnvironment)
  throws EvalCompilerExn;
```

- evaluate an expression in the context of the given environment, returning the result

An example evaluation operation

```
class IntLiteralExpr extends Expr {
  int value;

  Value evaluate(CodeEnvironment env)
      throws EvalCompilerException {
    return new IntValue(value);
  }
}
```


An example evaluation operation

```
class AddExpr extends Expr {
  Expr arg1;
  Expr arg2;

  Value evaluate(CodeEnvironment env)
      throws EvalCompilerException {
    Value arg1_value = arg1.evaluate(env);
    Value arg2_value = arg2.evaluate(env);
    return new IntValue(
      arg1_value.getIntValue()
      +
      arg2_value.getIntValue());
  }
}
```

`getIntValue` asserts that the value is an int and returns its value

(Real version factors most of `evaluate` into `ArithmeticBinopExpr` superclass)

An example overloaded evaluation operation

```
class EqualExpr extends Expr {
  Expr arg1;
  Expr arg2;

  Value evaluate(CodeEnvironment env)
      throws EvalCompilerException {
    Value arg1_value = arg1.evaluate(env);
    Value arg2_value = arg2.evaluate(env);
    if (arg1.getResultType().isIntType() &&
        arg2.getResultType().isIntType()) {
      return new BooleanValue(
        arg1_value.getIntValue()
        ==
        arg2_value.getIntValue());
    } else if (arg1.getResType().isBoolType() &&
               arg2.getResType().isBoolType()) {
      return new BooleanValue(
        arg1_value.getBooleanValue()
        ==
        arg2_value.getBooleanValue());
    } else {
      throw new InternalCompilerError(...);
    }
  }
}
```

An example evaluation operation

```
class NewExpr extends Expr {
   String class_name;

   Value evaluate(CodeEnvironment env)
         throws EvalCompilerException {
      ClassEnvironment class_env =
         env.lookupClass(class_name);
      ClassValue instance =
         new ClassValue(class_env);
      ClassSymbolTable class_st =
         getResultType().getClassInterface();
      class_st.initializeInstanceVars(instance);
      return instance;
   }
}
```

`lookupClass` looks up the environment for the given class

`initializeInstanceVars` initializes all the instance variables of the instance to their default values

An example evaluation operation

```
class VarDeclStmt extends Stmt {
   String name;
   Type type;

   void evaluate(CodeEnvironment env)
         throws EvalCompilerException {
      env.declareLocalVar(name);
   }
}
```

`declareLocalVar` adds a new binding to the current environment

(Real version also handles initializing rhs expression)

An example evaluation operation

```
class VarExpr extends AssignableExpr {
   String name;

   Value evaluate(CodeEnvironment env)
         throws EvalCompilerException {
      // (record var_iface during typechecking)
      return var_iface.lookupVar(env);
   }
}
```

`lookupVar` looks at the kind of variable being read, and does the right thing

- local variable:
  `return env.lookupLocalVar(name);`
  - returns contents of binding for `name` in `env` (or enclosing env)
- instance variable:
  `Value rcvr = env.lookupLocalVar("this");`
  `return rcvr.lookupInstVar(name);`
  - returns contents of binding for `name` in `rcvr` instance
- (static class variable?)

An example evaluation operation

```
class AssignStmt extends Stmt {
   AssignableExpr lhs;
   Expr rhs;
   void evaluate(CodeEnvironment env) ... {
      lhs.evalAssign(env, rhs);
   }
}
class VarExpr extends AssignableExpr {
   void evalAssign(CodeEnv env, Expr rhs) ... {
      // (record var_iface during typechecking)
      Value rhs_value = rhs.evaluate(env);
      var_iface.assignVar(env, rhs_value);
   }
}
```

`assignVar` looks at the kind of variable being assigned to

- local variable:
  `env.assignLocalVar(name, rhs_value);`
  - updates binding for `name` in env where it is declared
- instance variable:
  `Value rcvr = env.lookupLocalVar("this");`
  `rcvr.assignInstVar(name, rhs_value);`
  - updates binding for `name` in `rcvr` instance
- (static class variable?)

An example evaluation operation

```
class IfStmt extends Stmt {
   Expr test;
   Stmt then_stmt;
   Stmt else_stmt;

   void evaluate(CodeEnvironment env)
         throws EvalCompilerException {
      Value test_value = test.evaluate(env);
      if (test_value.getBooleanValue()) {
         then_stmt.evaluate(env);
      } else {
         else_stmt.evaluate(env);
      }
   }
}
```

`getBooleanValue` asserts that the value is a boolean and returns its value

Controls which substatement gets evaluated