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

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);
      class_env.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 uninitialized binding to the current environment

An example evaluation operation

```
class VarExpr extends Expr {
   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 {
   String lhs;
   Expr rhs;

   void evaluate(CodeEnvironment env)
         throws EvalCompilerException {
      // (record lhs_iface during typechecking)
      Value rhs_value = rhs.evaluate(env);
      lhs_iface.assignVar(env, rhs_value);
   }
}
```

`assignVar` looks at the kind of variable being assigned to, and does the right thing

- local variable:
  `env.assignLocalVar(name, rhs_value);`
  - updates binding for `name` in `env` (or enclosing env)
- 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