Project 3 Hints

Jingke Li

Portland State University

The IntVI Interface

```
public interface IntVI {
  // Program and functions
  public void visit(PROG t) throws Exception;
  public void visit(FUNC t) throws Exception;
  public void visit(FUNClist t) throws Exception;
  // Statements
  public int visit(STMTlist t) throws Exception;
  public int visit(MOVE t) throws Exception;
  public int visit(JUMP t) throws Exception;
  ...
  // Expressions
  public int visit(EXPlist t) throws Exception;
  public int visit(ESEQ t) throws Exception;
  public int visit(MEM t) throws Exception;
  ...
}
```

- Statement routines' return value is statement index or status value.
- Expression routines' return value is the expression's value.

Storage Model

The interpreter organizes data in three storage categories: *temps*, *stack*, and *heap*, each is implemented as an array.

- `int temps[maxTemps]` —
  `TEMP` nodes are mapped to the `temp` array directly: `(TEMP` *i*`)` is mapped to array element `temps[`*i*`]`.
- `int stack[maxStack]` —
  Function activation records are allocated and de-allocated on the stack.
- `int heap[maxHeap]` —
  Calls to `'malloc'` result in space allocated in the heap. In our model, heap objects are never de-allocated.

Initial Setup

```
public class InterpVisitor implements IntVI {
  private final int maxTemps = 512;
  private final int maxStack = 2048;
  private final int maxHeap  = 4096;
  private final int wordSize = 1;
  private final int STATUS_DEFAULT = -1;
  private final int STATUS_RETURN  = -2;
  private int[] temps = new int[maxTemps];
  private int[] stack = new int[maxStack];
  private int[] heap  = new int[maxHeap];
  private int sp = maxStack - 2; // stack starts from high index; reserve
  private int fp = maxStack - 2; //   1 slot for main class' (PARAM 0)
  private int hp = maxHeap - 1;  // heap also starts from high index
  private int retVal = 0;        // special storage for return value
  private FUNClist funcs = null; // keeping a copy of program's funcs
  private STMTlist stmts = null; // keeping a copy of current stmt list
  ...                            // you may add other variables
  public InterpVisitor() {}
}
```

Top-Level Routines

```
public void visit(PROG p) throws Exception {
  funcs = p.funcs;
  FUNC mf = findFunc("main");
  mf.accept(this);
}

public void visit(FUNC f) throws Exception {
  stmts = f.stmts;
  sp = sp - f.varCnt - f.argCnt - 1;
  f.stmts.accept(this);
  sp = sp + f.varCnt + f.argCnt + 1;
}
```

The Main Fetch-Execute Loop

```
public int visit(STMTlist sl) throws Exception {
  int ret = STATUS_DEFAULT;
  int i = 0;
  while (i < sl.size()) {
    int next = ((STMT) sl.elementAt(i)).accept(this);
    if (next == STATUS_RETURN) {
      ret = STATUS_RETURN;
      break;
    }
    i = (next >= 0) ? next : i+1;
  }
  return ret;
}
```

Statements and Expression Samples

```
public int visit(MOVE s) throws Exception {
  int val = s.src.accept(this);
  if (s.dst instanceof TEMP) {
    temps[((TEMP) s.dst).num] = val;
  } else if (s.dst instanceof MEM) {
  ...
  return STATUS_DEFAULT;
}

public int visit(JUMP s) throws Exception {
  return findStmtIdx(s.target);
}

public int visit(BINOP e) throws Exception {
  ... // evaluate both operands to lval and rval
  switch (e.op) {
  case BINOP.ADD: return lval + rval;
  ...
}
```

Handling Call Nodes

```
public int visit(CALLST s) throws Exception {
  ...
  if (fname.equals("print")) {
    ... // Call System.out.println()
  } else {
    ...   // evaluate args
    stack[sp] = fp;
    fp = sp;
    f.accept(this);
    sp = fp;
    fp = stack[sp];
    ...
  }
  ...
}
```

Extra Credit: Handling Floating-Point

- Need to change array declaration to allow both types:

```
private Val[] temps = new Val[maxTemps];
private Val[] stack = new Val[maxStack];
private Val[] heap  = new Val[maxHeap];
```

- Need to define a "union" class to handle both types:

```
public class Val {
  boolean isfloat;  // type indicator
  int n;            // either an int or a bit-pattern for float
}
```

  Use `Float.floatToIntBits(`*f*`)` and `Float.intBitsToFloat(`*n*`)` to convert between floating-point and integer bit-pattern representations.

- Or use OOP subtyping feature:

```
public abstract class Val {...}
public class iVal extends Val { int v; }
public class fVal extends Val { float v; }
```