Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu), Classes: MW 16:40-18:30, URBAN 204, Office Hours: MW 14:00-15:00.

Project 3: IR Code Interpretation
(Due Wednesday 2/22)

This project is to implement an interpreter for the IR tree language. The approach is the same as before, *i.e.* the interpreter is to be implemented as a visitor, although it is to visit the IR nodes.

The Visitor Interface

The visitor interface for the interpreter is called `IntVI`. It is defined in the file `ir/IntVI.java`:

```
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;
  ...
}
```

The visit routine for an expression node evaluates the expression and returns the evaluation result. In other words, the visit routine returns the value of the expression. The visit routine for a statement node returns an integer representing either a status value or an index to a statement. There are two status values: `STATUS_DEFAULT(-1)` and `STATUS_RETURN(-2)`. Any non-negative return value is treated as an index to a statement. (Detail is shown in the `STMTlist` section below.)

Storage Organization

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

- The `temps` array is a direct-mapping array, meaning that `(TEMP` i`)` is mapped to array element `temps[`i`]`.
- Both the `stack` and `heap` arrays are used dynamically. Every time a method is called, a new frame is allocated on the stack. When the call returns, the frame is removed. The `heap` array simulates a heap. When a new object is created, a block of storage is allocated in this array. Unlike a real heap, the allocated objects never get de-allocated in our simplified model.

The following is a suggested declaration list for constants and variables inside the visitor class:

```
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 1 slot
  private int fp = maxStack - 2;     //   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 input program's funcs
  private STMTlist stmts = null;     // keeping a copy of current statement list
  ...                                // you may add other variables
}
```

Interpreting STMTlist

As we learned in class, the main structure of an interpreter program is a *fetch-execute* loop over a list of statements. For our interpreter, the visit routine to the `STMTlist` node serves this purpose. To track control-flow-changing statements, every visit routine to a statement returns a status/index value:

```
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;
}
```

For `JUMP` and `CJUMP` nodes, the return value represents the jump target's statement index. For a `RETURN` node, the `STATUS_RETURN` flag is returned. For other statement nodes, the `STATUS_DEFAULT` flag is returned.

Interpreting Statements

- `MOVE` — The interpreter evaluates the lhs to a storage location, the rhs to a value, and assigns the value to the location.

  Note that some expression nodes (*e.g.* `VAR`) can appear on either side of a `MOVE` statement. However, there is only one visit routine to each expression node, we have to choose whether to have it return the l-value or the r-value of the expression. We choose the latter. This means when dealing with a `MOVE` node, we can invoke recursive visit to the rhs expression node, but we have to handle the lhs expression node manually to produce an l-value.

  The lhs expression of a `MOVE` node must be a temp or a memory location, which means it has to be one of the following nodes: `TEMP`, `MEM`, `FIELD`, `PARAM`, or `VAR`. To produce an l-value, a `TEMP` node is evaluated to a `temps` array cell; a `MEM` or `FIELD` node is evaluated to a `heap` array cell; and a `PARAM` or `VAR` node is evaluated to a `stack` array cell. (Read the later expression section for actual indexing rules for these nodes.)

- `JUMP/CJUMP` — For a `JUMP`, search for the label node corresponding to the jump target in the current statement list; and return the label node's index. For a `CJUMP`, evaluate its condition to decide whether to return the statement index or the default status value.

- `CALLST` — For the `print` routine, call `System.out.println` to print out the argument's value. Otherwise, take the following steps:

  - push parameters onto the stack (at `stack[sp+1]`, `stack[sp+2]`, etc.);
  - adjust the frame pointer (save the current `fp` in `stack[sp]`, and set `sp` to be the new `fp`);
  - find the `FUNC` node corresponding to the routine from the program's funcs list, and switch there to execute (by invoking `accept()` on the `FUNC` node);
  - after control returns, restore the saved frame pointer.

- `RETURN` — If there is an associated expression, evaluate the expression and assign the value to the special `retVal` variable. Return the `STATUS_RETURN` status value.
- `LABEL` — No action is needed. Just return the default status value.

Interpreting Expressions

Every expression is evaluated to an integer value. Floating-point constants are all converted to integers.

- `BINOP` — evaluate the operands and apply the binary operation; then return the result.
- `TEMP` — simply return the value from the corresponding `temps` array cell.
- `CALL` — if the routine is `malloc`, allocate space in the `heap` array, otherwise take similar steps as in the `CALLST` case, plus returning `retVal` as result.
- `MEM` — evaluate the address expression into an integer, use it as an index to the `heap` array, fetch and return the value from the corresponding cell.
- `FIELD` — evaluate the `obj` component to an index to the `heap` array, adjust the index with the `idx` component's value, then return the value of the corresponding cell. For example, for `(FIELD (VAR 1) 2)`, first evaluate `(VAR 1)` to an integer, say `4023`, then adjust it by the index value `2` to `4025`, and return the value of `heap[4025]`.
- `PARAM/VAR` — convert the node's index to an offset to the frame pointer `fp`, and return the value from the corresponding `stack` cell. Variables are on the positive side of `fp` and parameters are on the negative side. For example, for `(VAR 3)`, the value of `stack[fp-3]` should be returned; for `(PARAM 3)`, the value of `stack[fp+4]` should be returned (note the offset adjustment in this case.)
- `CONST` — return the constant's value.
- `FLOAT` — convert the floating-point value to an integer, and return the integer value.
- `(NAME "wSZ")` — return `wordSize`'s value (which is 1).

The `ESEQ` is included in the `IntVI` interface for the sake of completeness. It should not appear in the input program to the interpreter. You may provide an empty implementation for its visit routine or throw an exception from the routine.

More on the Handling of `CALL/CALLST` Nodes

For each method invocation, a frame is allocated on the stack, and the frame pointer is switched to point to the new frame; when a method returns, the reverse actions are performed.

- *Allocating and de-allocating a frame* — This corresponds to adjusting the stack pointer's value and it happens when a `FUNC` node is being interpreted:

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

- *Adjusting the frame pointer* — When control is switched from a caller to a callee, the frame pointer `fp` needs to be adjusted accordingly. This happens at the `CALLST/CALL` node:

```
public int visit(CALLST/CALL ...) {
  ...
  stack[sp] = fp;
  fp = sp;
  f.accept(this);  // f is the callee FUNC node
  sp = fp;
  fp = stack[sp];
  ...
}
```

Extra Credit: Proper Handling of Floating-Point Values

The above implementation simplifies the handling of floating-point values by converting them to integers. To handle floating-point values properly, you need to allow them to be stored in any storage location and used in any operation.

One possible approach is to define a "union" class for holding both types of values:

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

For a floating-point value f, you can use `Float.floatToIntBits(`f`)` to convert it to an integer bit-representation n; and use `Float.intBitsToFloat(`n`)` to restore the floating-point form.

Another approach is to use the standard OOP subtyping feature:

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

In either case, you need to change the storage array declarations to

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

and change every operation that performs on these values.

You'll earn two extra points if you handle this part correctly.

Code Organization and What to Turn In

As usual, you should first download and untar the file `proj3-code.tar` to create the directory structure. The interpreter program should be called `InterpVisitor.java` and placed in the subdirectory `interp`. The script for running the interpreter is called `runint`. Submit your program `InterpVisitor.java` through the "Dropbox" on the D2L class webpage.