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 1: IR Code Generation (First Version)

(Due Wednesday 1/25/12)

This project is to implement a first version of an IR code generator for the MINI compiler. The target IR is the IR tree language discussed in class. The generator is to be implemented as a visitor to the MINI AST nodes. The visitor interface is defined in `TransVI.java` in the `ast` subdirectory.

When the visitor `IrgenVisitor0` is invoked on a MINI AST `Program` node, an IR tree rooted at a `PROG` node should be generated. The following is a guide on how to translate specific AST nodes. Since this is a first version of the IR code generator, some nodes' translations are simplified. As an example, all variables retain their names in this version. (This is not true in the final version of the IR code.)

Declarations

- *VarDecls* — If a `VarDecl` has no initialization expression, no action is needed; otherwise, it is treated as an assignment statement, and is translated into a `MOVE` node.
- *MethodDecls* — Each `MethodDecl` node is translated into a `FUNC` node. To do this,
  - translate the method's `VarDeclList` into a (possibly empty) `STMTlist`, and
  - translate the method's `StmtList` into a `STMTlist`.

  Merge the two `STMTlists`. The list classes in our IR tree are defined as extensions of Java's `Vector` class, hence the method `addAll` could be used to merge two lists. Use the result to create a `FUNC` node.

  The `FUNC`'s `label` field gets the Method's name. Its two other fields, `varCnt` and `argCnt`, are not used in this project, and hence should be set to 0s.
- *ClassDecls* — A `ClassDecl` node consists of a list of `VarDecls` and a list of `MethodDecls`. `VarDecls` are to be ignored here, since they will be handled in `NewObj` node. Each `MethodDecl` is translated into a `FUNC` node, so a `ClassDecl` is translated into a `FUNClist`. The order of elements in a `FUNClist` does not affect the program's correctness.

  The class id and its parent pointer are not used in this project.
- *Program* — The `FUNClists` from multiple `ClassDecls` are merged into a single `FUNClist`, and is used to create a `PROG` node.

Statements

Translations for statement nodes are mostly straightforward. Only the `If` node needs some attention, since `CJUMP` node has only one branch.

- *Assign* — Translate it into a `MOVE` node.
- *CallStmt* — Translate it into a `CALLST` node. (A `Call` node similarly translates into a `CALL` node.)
- *If and While* — Translate each into a combination of `CJUMP`/`JUMP` nodes, with value-represenation approach (as discussed in class).
- *Return* — Translate it straightforwardly into a `RETURN` node.
- *Print* — Translate it into a `CALLST` node with function name `print`.

Expressions

- *Arithmetic binary operations* — Translate them into corresponding `BINOP` nodes.
- *Unary operations* — For both the `NEG` and `NOT` unary operations, translate them into a binary `SUB` node with constant 1. For instance, `(Unop - (Id x))` is translated into `(BINOP - (CONST 1) (NAME x))`.
- *Boolean expressions* — Translate them using the value-representation approach, as studied in class.
- *Identifiers* — Simply translate them into `NAME` nodes (for now).
- *NewArray* — Needs to allocate space and initialize elements:
  - create a `CALL` node to "malloc" space (with one extra cell for holding array length), and a `MOVE` node to save the return value in a `TEMP`;
  - create a statement to save the array element count into the first cell;
  - create statements to initialize array elements to 0; and
  - create an `ESEQ` node to glue the statements together and return the temp.
- *Array Elements* — Translate them into `MEM` nodes with array pointer and an offsets for fetching the stored elements. I.e. `a[0]` gets translated into `(MEM (BINOP + <a's addr> (BINOP * (CONST 1) (NAME wSZ))`. Here `wSZ` is a predefined constant name, representing the word-size.
- *ArrayLength* — Translate it into a `MEM` node for fetching the stored array length value (always as the first cell of the allocated array object).
- *NewObj* — In this project, translate it into a call to `"malloc"` with a synthetic argument. The argument has the form `NAME("<class-name>_obj_size")`, where `<class-name>` is to be substituted with the actual class name appeared in the `NewObj` node.
- *Field* — Translate it into `NAME(<field-name>)` in this project.
- *This* — Simply translate it into `NAME("this")` in this project.
- *IntVal* — Translate it into a `CONST` node.
- *FloatVal* — Translate it into a `FLOAT` node.
- *BoolVal* — Translate it into a `CONST` node with value 1 or 0.
- *StrVal* — Translate it into a `STRING` node.

Code Organization

Copy and decompress the file `proj1-code.tar`. You'll see a `proj1` directory and several subdirectories:

- `ast` — the AST node definitions. They are the same as last term except that a new `TransVI` interface is added to each class.
- `astpsr` — the ast parser program.
- `ir` — the IR tree node definitions.
- `irgen0` — this is where your program should be placed; a driver program is already there.
- `tst` — some test programs.

There is also a `Makefile` and a `runi0` script. To compile your program, do: `make irgen0`. To run `IrgenVisitor0` on `tst/test01.ast`, do: `./runi0 tst/test01.ast`.

What and How to Turn in Your Program

Submit your program `IrgenVisitor0.java` through the "Dropbox" on the D2L class webpage.