

CS126: Introduction to Compilers

Assignment 8: ICODE

Out: 10/31/03

Due: 11/14/03

1.0 Introduction

The next step in your compiler involves taking the fully resolved abstract syntax trees you have been working hard to achieve and translate them into an intermediate representation that is suitable for optimization and eventually assembler code generation.

2.0 Intermediate Code

The intermediate code representation we will use was described in class and is defined in the interface `DecafIcode`. The most relevant data types are:

- `DecafIcode.Routine`: this class contains the intermediate code for a single routine. It provides access to the flow graph for the routines, maintains the set of temporaries and constants, and provides a few high-level utilities to service optimization.
- `DecafIcode.Block`: this class represents a flow graph basic block. It provides access to the instructions in the block (in forward or reverse order), and the the edges into and out of the block. It also provides methods for adding instructions to the block and for linking blocks in the flow graph.
- `DecafIcode.Edge`: this class is used to link blocks in the flow graph. It is created and maintained by the `Block` class, but is available to provide information about the different edges.
- `DecafIcode.Instruction`: this class holds a single intermediate code instruction, designated by its op code and its set of arguments. It provides access to the op code and arguments and separate access to the target and to temporary (as opposed to constant) arguments.
- `DecafIcode.Constant`: this class holds constant arguments for instructions. Constants are generally created through appropriate calls in `Routine` and their use should be limited to those arguments of instructions that require constants (as opposed to temporaries).
- `DecafIcode.Temporary`: this class represents a temporary within the intermediate code. Temporaries are created by appropriate calls in `Routine`, either to represent variables or to represent the output of a subexpression. Note that these calls will

ensure that two identical subexpressions are assigned the same temporary. `Temporary` provides access to the number, type, and data type of the corresponding temporary.

- `OPC_XXX`: this is an enumeration of the different op codes that can be used in intermediate code instructions.
- `DecafIcode.CodeType`: this is the set of data types that can be assigned to temporaries.

3.0 Allocation Pass

Intermediate code generation is done using visitors on the abstract syntax tree. As described in class, there are two visitors that you will have to define.

The first one is charged with determining the class and vtable offsets for each declared field and method respectively. It should call:

- `Ast.Field.setOffset(int)` for each field, specifying the offset for that field in bytes from the start of the class record. Note that this value should be suitably aligned. (You can assume that all classes are initially allocated at an address that is 8-byte aligned.)
- `Ast.Method.setOffset(int)` for each non-static method, specifying the offset into the vtable for that method.
- `Ast.ClassType.setLengths(int c,int v)` for each class, specifying the size of the class and the size of the vtable for the class respectively.

4.0 Code Generation Pass

The second visitor should do the actual code generation. For each routine in the abstract syntax tree, it should create a `RoutineInfo` object. This object should then be used to create the appropriate flow blocks, temporaries, and constants. The flow blocks thus created should be used to contain the appropriate instructions for the program reflected by the abstract syntax tree and to create and link other flow blocks.

This pass should make use of the information that was requested in the previous assignment, in particular, what instruction sequences should be generated for each abstract syntax tree node. It should also make use of a stack that represents the current values that reflect the result of generating code for a subtree as described in class.

5.0 Top-Level Interface

You should augment your factory class with two methods that yield your visitors.

```
createAllocationPass(DecafErrorHandler,DecafAsmGenerator)
```

should return the allocation visitor.

```
createIcodeGenerator(DecafErrorHandler,DecafIcode.Icode,DecafAsmGenerator)
```

should return the intermediate code generation visitor.