CSC488 Assignment 5: Team 01

Tian Ze (Peter) Chen, Badrul Chowdhury,
Felix Lu, Anthony Vandikas, Vishrant Vasavada

Index of Java Classes

The following classes and interfaces have been added to the compiler:

- `compiler488.codegen.ir.Address`
  Represents an address. This has two subclasses, `StackAddress` and `PlainAddress`.

  Stack addresses refer to locations on the stack, and contain a scope level and offset for use with the `ADDR` instruction. This is used to refer to variables.

  "Plain addresses" refer to any location in memory, and are used to refer to routine names, labels and function return addresses.

- `compiler488.codegen.ir.Cell`
  Represents various tokens, such as opcodes, scalar constants, labels, as well as anchors for backpatching.

- `compiler488.codegen.ir.Entity`
  To avoid name clashes with the `Symbol` type used for semantic checking, we refer to our symbols as `Entities`. Each `Entity` keeps track of its address and required amount of memory. The `Entity` interface has two main implementations:
  - `PlainEntity` - Refers to anything that is represented by a block of memory, i.e. all variables.
  - `RoutineEntity` - Refers to functions and procedures.

  The above two classes are themselves base classes for more specific implementations for scalar, 1D array, 2D array, function, and procedure types.

- `compiler488.codegen.ir.Generator`
  Contains the bulk of the generation code. Unlike our implementation of semantic analysis, code generation functionality is not added by extending the AST. Instead, we have functions that handle families of AST nodes by checking their types.

- `compiler488.codegen.ir.LexScope`
  As with symbols, we refer to our code-generation-level symbol table as a `LexScope` to avoid name clashes.

  According to the language specification, variables declared in minor scopes are accessible anywhere in the enclosing major scope after their declaration. To facilitate

this, we collapse all minor scopes into the nearest enclosing major scope and allocate the variables all at once. The `LexScope` interface provides two functions for adding symbol table entries, `addLocal` (for symbols that allocate storage space) and `addRefer` (for symbols that are allocated elsewhere). Each implementation of `LexScope` keeps track of the closest enclosing major scope, so minor scopes implement these functions by delegating the call to the outer scope.

- `compiler488.codegen.ir.Snippet`
  `Snippet` objects represent generated code fragements. This class contains numerous methods for generating specific opcodes, pseudo instructions (see below), pretty printing (for checking our A4 code), and submitting instructions to the `Machine` class.

  Most functions return an instance of the object being called (`this`), providing a fluent interface with which we can chain multiple calls to the same object.

In addition to the above, the `generateCode` method in the `compiler488.compiler.Main` class has been modified to instantiate and invoke the `Generator`.

Testing Strategy

Every code generation rule in the assignment handout is covered by at least one test case. Unlike our previous submissions, we did not automatically generate our test cases, and instead opted to write them by hand.

Our test script has been modified to pipe an input file to each test case and match the program's actual output against an expected output file. The input and output files are located by replacing the .488 extension of each test case by .in and .out respectively.

Other Notes

As we specified in our Assignment 4 submission, we have defined a number of "pseudo-instructions" that represent certain collections or patterns of instructions. For example, a "NOT" instruction is a short form for "PUSH FALSE; EQ".

There are very few discrepancies between our Assignment 4 submission and our Assignment 5 code, as the two were developed simultaneously (we had a working code generator before Assignment 4 was due). Most of the changes made between the Assignment 4 due date and now were optimizations suggested in the feedback we received from Assignment 4. The only significant *semantic* change was implemented when we realized that indices for arrays with no explicit lower bound started from one, not zero.