MP 8 – MiniJava Compiler

CS 421 – Spring 2013
Revision 1.0

Assigned Thursday, March 7, 2013
Due Thursday, March 14, 2013, 09:30
Extension 48 hours (20% penalty)
Total points 60

1 Change Log

1.0 Initial Release.

2 Objective

You will write a compiler for MiniJava, translating it to an abstract machine language for a machine we will provide.

After completing this MP, you will have learned about the following concepts:

- the role of type-checking (although you will not write a type-checker)
- code generation for statements and expressions

We don't believe this assignment is extremely difficult, but the write-up is very complex. Please start early. We will say exactly what code you need to generate in SOS-like rules for code generation. For the most part, this is just a simple recursive traversal of the AST. Our solution — that is, the part of the solution that you will write — is about 100 lines of OCaml code. We don't think debugging will be that hard, because you will be able to compare the output of your compiler directly to the output of ours.

3 Style Requirements

In this MP, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by `handin`, so acceptance by `handin` does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.

We will only enforce the rules just listed, but a more comprehensive style guide can be found at `http://caml.inria.fr/resources/doc/guides/guidelines.en.html`.

4 What to submit

You will submit `mp8.ml` using the `handin` program. Rename `mp8-skeleton.ml` to `mp8.ml` and start working from there.

As in previous, once you have finished appropriate sections of your interpreter, you can use the `run` and `run_with_args` functions to test individual programs, or simply add programs to the test suite. (`evaluate` is not available for this assignment.)

- Download `mp8grader.tar.gz`. This tarball contains all the files you need, including a MiniJava lexer and parser (as in MP6 and 7).
- As always, once you extract the tarball, rename `mp8-skeleton.ml` to `mp8.ml` and start modifying the file. You will modify only the `mp8.ml` file, and it is the only file that will be submitted.
- Compile your solution with `make`. Run the `./grader` to see how well you do.
- You may use the included `testing.ml` file to run tests interactively. Open the OCaml repl and type `#use "testing.ml";;` to load all of the related modules and enable testing. Further specifics on interactive testing are included in that file.

The functions you write for this assignment will translate MJ programs to a made-up machine language. Once you've run the compiler and produced the machine language program, there are two ways of checking if it is correct: print it using the provided `string_of_prog` function, or run it using the abstract machine emulator (`execute`). The correctness test we will use is execution in the emulator, so we do not require that your machine language programs be exactly the same as ours. (They probably will be the same, because the write-up describes in detail correct machine language for each statement and expression, and there is no particular reason why you would generate different code.) However, in debugging, we strongly recommend that you use the `string_of_prog` function to print your machine language programs; this is much the better way to see if you are generating correct code — and, if not, in what way it is wrong.

5 Overview

You will be translating a type-checked version of MiniJava into code for a made-up machine. The concrete syntax of MJ is the same as always, and we are providing a lexer and parser, as we did for MP6 and 7. The result of compiling and executing MJ programs should be identical to the output from MP7 *for type-correct programs*. The compiler will handle all the statements and expressions that were in MP7, including inheritance. (There is one exception: the compilation rules do not incorporate short-circuit evaluation of boolean expressions. This means that an expression like `true || 3/0` will crash when compiled and executed, whereas it would have returned `true` in MP6 or 7. Rectifying this problem requires short-circuit compilation; this will be discussed in lecture.)

So here are the things that need explaining in this document:

- *Abstract syntax of type-checked programs.* Instead of generating a separate symbol table, we have incorporated the results of type-checking into the AST. This has required that we define a new version of the abstract syntax; this is explained in section 6. (*Type-checking and translation to the new ASTs is done in code provided by us; this is not part of the assignment.*)
- You will be compiling for a machine that was made up just for this assignment. The overall structure of the machine — stack and heap, program counter, machine instructions — is pretty conventional and shouldn't be hard to understand (although the details will certainly take some time to absorb). The machine is described in section 7. (*Again, the program to implement the abstract machine is provided by us, and is not part of the assignment.*)
- The machine-language programs you will produce consist of a map of all class names to methods and then, for each method, the machine language instructions to implement that method. Those instructions are, as just stated, fairly conventional. You will not be asked to implement the map from class names to methods — that is constructed by the type-checker. Your only job is to create the instruction sequences for each method body.

To give an example, consider this class:

```
class Main {
   public boolean main (int n) {
      return this.isOdd(n);
   }
```

```
   public boolean isOdd (int n) {
      boolean b;
      if (n == 0)
         b = false;
      else
         b = this.isEven(n - 1);
      return b;
   }

   public boolean isEven (int n) {
      boolean b;
      if (n == 0)
         b = true;
      else
         b = this.isOdd(n - 1);
      return b;
   }
}
```

We first pass this to `lex_and_parse`, which (as in MP6 and 7) produces an AST. We pass that to `annotateProg` which produces the new version of the AST. Calling `compile` on this AST produces a machine-language program, which can be printed by calling `string_of_prog`, producing this output:

```
class Main
   main Main
   isOdd Main
   isEven Main

method main in Main (3)
0:   INVOKE     0,isOdd,1
     LOADRESULT 2
     RETURN     2

method isOdd in Main (6)
0:   LOADIMM    3,0
     EQUAL      4,1,3
     CJUMP      4,3,6
3:   LOADIMM    3,0
     MOV        2,3
     JUMP       11
6:   LOADIMM    3,1
     SUB        4,1,3
     INVOKE     0,isEven,4
     LOADRESULT 5
     MOV        2,5
11:  RETURN     2

method isEven in Main (6)
0:   LOADIMM    3,0
     EQUAL      4,1,3
     CJUMP      4,3,6
3:   LOADIMM    3,1
     MOV        2,3
     JUMP       11
6:   LOADIMM    3,1
     SUB        4,1,3
     INVOKE     0,isOdd,4
     LOADRESULT 5
     MOV        2,5
```

```
11:  RETURN     2
```

The first four lines of this output is just the class map, saying which functions are in the provided classes. After that is the code you will produce. The machine language is explained below, but it is not too hard to read at a general level. For instance, looking at `isOdd` we see that it loads zero into location 3, compares location 1 (the argument `n`) to location 3, and puts the result in location 4, and then conditionally jumps on the value in location 4 (0 for false, 1 for true). If that value is 1 — indicating that the comparison between `n` and zero was true — it jumps to 3; the code there loads 0 (for false, since 0 is not odd) into location 3, moves that to location 2 (local variable `b`), and jumps to 11 where it returns the false value. (The initial load into 3 and the move to 2 are clearly unnecessary; it is common that a simple translation to machine language produces such inefficiencies.) If the initial comparison yielded false, then there was a jump to 6, which subtracts 1 from `n` and calls `isEven`; upon return, the result of that call is moved to location 5, and thence to location 2, and then returned as the result of this call.

The function `compile` calls `compileMethod` repeatedly to create the code for each method, and calls `makeSupertab` (once) to create the table of classes. `compileMethod` in turn calls auxiliary functions such as `compileStmt`. Your job in this assignment will be to write `compileStmt` and the other auxiliary functions. We provide the definition of `compileMethod`; you can use that both as a model of the `compile` methods, and as an example of how to translate compilation rules to code; in particular, `compileMethod` is our translation of the compilation rule in Figure 4.

We now explain the abstract syntax and the abstract machine, and then (in section 8) your assignment.

6 Abstract syntax after type-checking

MiniJava programs are type-checked by function `annotateProg`, which checks for type errors, and adds type information to every expression node. This is followed by application of function `addLocations`, which annotates each expression with the location on the stack where that expression's value should go. The result of calling these two functions is a copy of the original AST in a slightly changed form. To accommodate the changes, there are new AST types `programT`, `classT`, `methodT`, and so on. These are given in Figure 1.

For the most part, the new abstract syntax types are the same as the old, but with a capital T added to every constructor. Here are the important new parts of these ASTs:

- `classT`: The new `int` field at the end gives the size of objects of this class (i.e. the number of fields in the class).
- `methodT`: The new `int` field give the size of stack frames (environments) for this method; this is the number of arguments and local variables, plus 1 for the receiver (`this`), and a number of locations for temporary values used during expression evaluations.
- Instead of an `Assignment` constructor, type `statementT` has constructors `AssignVarT` and `AssignFieldT`; the type-checker has determined for each assignment whether it is an assignment to a parameter/local variable or to a field. Both of these constructors have a new int field, giving the location of the variable on the stack (for `AssignVarT`) or the location of the field within an object (`AssignFieldT`).
- Where `exp`s were used before, the new abstract syntax uses "annotated exps" of type `annExpT`:

  ```
  annExpT = expT * exp_type * int
  ```

  These give the expression, but also give its type and, more importantly, the location on the stack where its value should go. In the new `expT` type, for all operators that have subexpressions, those subexpressions have type `annExpT`, so you know where the value of every subexpression should go when it is calculated.
- `expT` has several new constructors:
  - `FieldRef` is used whenever an identifier used in an expression was determined by the type-checker to be a field (replacing `Id f`); the int argument gives the location of that field within its object.
  - `NewIdAlloc` replaces `NewId`. It still gives the name of the class whose object is being created, but it also gives the size of its objects. So this instruction has enough information to construct an object of that class, without consulting any other tables.

- `CvtIntToString` and `CvtBoolToString` are new unary operations that are inserted into the AST when the type-checker determines that an int or bool is to be converted to a string (because it appears as an operand of + where the other operand is a string).
- There is a new `binary_operation` called `Strcat`, which replaces `Plus` in the AST when the type-checker determines that the + operation actually refers to string concatenation. For example, if `s` is a variable of type `string` and `i` a variable of type `int`, then the original AST `Operation(Var "s", Plus, Var "i")` will become `OperationT(VarT "s", Strcat, CvtIntToStringT (VarT "i"))` (and this will, in turn, be decorated with types and locations).

Two of the AST operations in `expT` — `FieldT` and `NewIdT` — are actually never included in the resulting AST (they are just used temporarily during its construction), so you don't have to be concerned about compiling them.

The upshot of all this is that an AST of type `programT` can be compiled entirely on its own, with no reference to any additional tables. When compiling such a tree, there is never a need to use a variable name, because values can be stored in the locations given in the AST nodes. In particular, when compiling an assignment statement, the locations given in the instruction (whether it is an assignment to a variable or a field) are sufficient to generate correct code; the name of the variable or field is irrelevant. Furthermore, there is never a need to look at the type of an expression during compilation; any time a particular type is needed — for example, when an int is needed as the argument of an operation, or a bool as the condition in an if statement — the type-checker has already determined that the expression has the correct type. This makes the compilation process a straightforward traversal of the AST.

7 The abstract machine

We have defined a machine that is a greatly simplified version of an actual one. Its instructions are listed in Figure 2, and it is defined formally in Figure 3.

This machine has a stack and a heap. At any given time, it is executing code from a particular method, at a given location (the program counter, or pc). The heap contains objects and strings, and also has a value called `Unallocated`; objects contain a class name and an array of fields (initialized to zeros). We have no garbage collector, but simply allocate new memory from the top of the heap, which is given by a variable `heaptop`. Finally, there is a single register, `reg0`, used only to pass results back from method calls.

In summary, the machine's state contains six values: pc, code of the current method, stack, heap, heaptop, and reg0. Furthermore, the stack is a list of frames, each frame containing three parts: the environment of the current call, which is just an array of integers, and the code and pc of the method that called this one. The head of the stack is considered the top of the stack, that is, the "current stack frame." Technically, the environment is the first component of the current stack frame, but we will take liberties with the notation and just say "get location i from the current stack frame."

The instructions do things similar to instructions in actual machines. Some examples are:

- `MOV tgt,src` moves the integer at location `src` in the current stack frame to location `tgt` of the current stack frame.
- `ADD tgt,src1,src2` gets the values out of locations `src1` and `src2` in the current stack frame, and puts their sum in location `tgt`.
- `JUMP` just changes the pc to its argument. `CJUMP` changes the pc to either its second or third argument, depending whether the first argument (a location in the current stack frame) contains a 1 (for true) or a 0 (for false).
- `NEWSTRING tgt,string` allocates a new location in the heap and puts the string there, then stores the heap address of the string in location `tgt` of the current stack frame.
- `INVOKE rcvr,f,[arg1;...;argn]` calls the method `f` of the receiver's object. `rcvr` and the arguments are locations in the current stack frame; `rcvr` contains the location of an object. To execute this instruction, the machine does the following: (1) It gets the heap address in location `rcvr`; suppose the heap contains object (C,fields). (2) Using the `super_table` and `method_table` described below, the machine finds the appropriate definition of `f` and gets its code. (3) It creates a new stack frame by constructing a new environment

(using the frame size found in the `method_table`), and creating a triple with the current pc and code, and pushes that triple onto the stack. (4) Lastly, it sets the new code to the instructions found in the table, and sets the pc to 0.

- `RETURN src` first puts the value of `src` in the current stack frame into `reg0`, and then reverses the process of `INVOKE`.

All of these instructions are defined precisely in Figure 3. Here is some explanation of that figure. Each line says how the state of the machine changes when an instruction is executed. The state of the machine consists of six values, as noted above: the pc, code, stack, heap, heaptop, and retval. In the table, these are named (p, c, s, h, t, r). As mentioned above, we have taken some liberties with the notation; The stack is a list of triples of the form (environment, pc, code), and the current stack frame is the head of this list; however, we write "s(src)" to mean "the value in the environment of the top frame of the stack;" in other words, "s(src)" really means "nth src (fst_of_3 (hd s))". Similarly, "s[val/loc]" is the environment on the top of the stack modified to have value val at location loc. Other notations: We are representing booleans by integers, so when we write $i < j$ we really mean: 1, if $i < j$, 0 otherwise. Similarly, logical operations `&&` and || have numerical meanings (again, 0 is false, 1 is true). "int2str i" is the string version of i (`string_of_int i`, in OCaml), and "bool2str i" is `"true"`, if i=1, `"false"` otherwise.

8 Compilation

A machine language program is an element of type

```
type mlprogram = method_table * super_tabl
```

where

```
type super_table = (classname, (methodname, classname) mapping) mapping
type method_table = (methodname * classname, int * funcode) mapping
```

and

```
type ('a,'b) mapping = ('a * 'b) list
```

The `super_table` is a table that, given a class name C, gives the table of methods for C; this table provides, for any method m, the class in which objects of class C will find the definition of m. It exists to account for inheritance. For example, if class C defines method g and inherits method f from B, then `super_table`(C) is a table mapping g to C and f to B. It simply tells the machine where to find the definition of any method that is invoked. In fact, you will not need to look at this table at all; it is used by the machine emulator to execute the `INVOKE` function.

When a method m is invoked on a receiver, we use the super_table to find the class s in which m is defined (either in the receiver's class or a superclass), and then look up the pair (m,s) in the `method_table`. This provides, for each method, the code for the body of that method, and the size of the stack frames for that method. An appropriate stack frame is constructed, and the machine jumps to the start of the code for the invoked method's body.

The latter is already contained in the AST, so the function `compileMethod` just has to produce the instructions for that method. We provide the code (the `compile` program) that is responsible for calling `compileMethod` for every method in the program, and constructing the `method_table`.

Each method, statement, and expression generates a sequence of machine language instructions. These instructions are specified in the "SOS" rules for compilation given in Figures 4, 5, and 6. As is often the case, some additional information must be calculated in some cases. The rules have these forms:

Methods: $M \rightsquigarrow il$

Statements: $S, m \rightsquigarrow il, m'$

Expressions: $e, loc \rightsquigarrow il$

A method just generates a sequence of machine language instructions. The rule for expressions says that e is compiled to the sequence of machine language instructions il, and when il is done, the value of the expression will be in stack location loc. The rule for statements say that S compiles to il, but also says that the code for S starts at location m (an integer) and ends at location $m' - 1$ (so that m' is the location where the next instruction will go). These locations are needed so that it will be possible to know the targets of jumps in `if` statements.

These rules correspond, as usual, to the types of the compilation functions in MP8, specifically:

```
compileMethod (meth:methodT) : ml_instr list
compileStmt (stmt:statementT) (m:int) : int * (ml_instr list)
compileExp ((ex,te,loc) as ae:annExpT) : ml_instr list
```

These types correspond to the translation rules, with one exception: Every expression node in the annotated AST gives the location where the value should go, so it isn't provided as a separate argument to `compileExp`; the loc in the SOS rule is just the `loc` in the argument `ae`. (As mentioned earlier, the type expression `te` actually plays no role in `compileExp`.)

To help get you started, we have provided the definition of `compileMethod`. Compare that code to the SOS rule in figure 4.

One aspect of Figure 6 needs explanation. In the rule for compiling binary operations, we compile `OperationT(`$e1$`,bop,`$e2`)` by using a machine operation we call BOP. This is just a shorthand way of saying that the AST operation should be translated to the corresponding machine language instruction: Plus to ADD, Minus to SUB, Multiplication to MULT, Division to DIV, And to AND, Or to OR , LessThan to LESS, Equals to EQUAL, Strcat to CATSTRINGS. (In the implementation, `compileExp` has one clause for binary operations, which calls `applyOp` to do this translation.)

We give an example, explaining it informally and showing how the compilation rules work. Suppose `x` is a variable of type `int` appearing as a local variable in a method f; f has one parameter, and `x` is the first local variable declared, so its address is 2 (location 0 is for `this`, and location 1 for the parameter). Consider the assignment statement `x = x+1;`. The fully annotated AST for this statement is:

```
AssignVarT ( "x",
             (OperationT ((VarT "x", IntType, 2),
                          Plus, (IntegerT 1, IntType, 3)),
              IntType, 4),
             2);
```

`AssignVarT` has three components: the name of the variable being assigned, the expression on the right hand side, and the location of the variable. The expression is, recall, an `annExpT` with three parts: the AST (`OperationT(...,3)`), its type (`IntType`), and its location (4). (The locations for intermediate values are not picked very cleverly; in this case, the value of the expression could have gone directly into location 2; instead, it will be put in location 4 and then moved to 2.) The expression (`x+1`) consists of two arguments, each of them an `annExpT`. The variable `x` gives its own location, while the constant `1` goes into a temporary location (3).

The compilation rule for assignment statements (to local variables) says to compile the expression and then move its value to the location of the variable. Here, the location of `x` is 2, and the location given in the expression is 4. So this statement will compile to the code for the expression followed by instruction "MOV(2, 4)". The rule for expressions says to compile the addition by first compiling the arguments and than adding an ADD instruction. So, we compile first `x` and then `1`. There's a strange thing, though: the rule for variables say to do nothing! Actually, that's correct: the ADD operation just needs a location to find its argument, and the value of variable `x` is already in location 2, so there is nothing to do. To compile `1`, we need to put a 1 in a temporary location, using LOADIMM. The AST says it should go into location 3, so we use LOADIMM(3,1).

We are now done, except for one thing, which is the locations of the generated code (the m in the rules). Suppose the code for this assignment is being generated at location 10 in the code stream. The rule for assignments says to get the length of the code for the expression, which is 2, then add 1 (for the MOV instruction), and add them both to 10. Now we have our complete result:

x=x+1, 10 $\rightsquigarrow$ [LOADIMM(3,1);ADD(4,2,3);MOV(2,4)], 13

Note that this entire process made no reference to any of the names or types in the AST.

The compilation rules in Figures 4, 5, and 6 explain everything about the compilation process. Here are brief explanations of the functions that you need to implement:

```
compileStmts (stmts:statementT list) (m:int) : int * (ml_instr list)
```

— Compile each of the statements and concatenate the instruction lists; the final location of the last statement is the final location of the entire statement list.

```
compileStmt (stmt:statementT) (m:int) : int * (ml_instr list)
```

— Compile the statement.

```
and applyOp (bop:binary_operation) (tgt:int) (opnd1:int) (opnd2:int) : ml_instr list
```

— Follow the rule for binary operations. We are not using short-circuit evaluation for boolean operations. `applyOp` just translates `binary_operation`s to their corresponding machine language instructions.

```
and compileExp ((ex,te,loc) as ae:annExpT) : ml_instr list
```

— Compile the expression according to the compilation rules, where the location is just `loc` given in the argument.

```
and compileExpList (aelis: annExpT list) : ml_instr list
```

— Compile list of expressions and return the combined list of instructions. You don't need to return a list of locations, since the caller of this function can get those out of the expressions.

```
(* Type-annotated programs *)
type programT = ProgramT of (class_declT list)

and class_declT = ClassT of id * id
        * ((var_kind * var_decl) list)
        * (method_declT list)
        * int (* number of fields *)

and method_declT = MethodT of exp_type
        * id * (var_decl list) * (var_decl list)
        * (statementT list) * annExpT
        * int (* size of stack frame: # of arg + # of locals
                                      + max # of temporaries *)

and statementT = BlockT of (statementT list)
    | IfT of annExpT * statementT * statementT
    | AssignVarT of id * annExpT * int
    | AssignFieldT of id * annExpT * int

and annExpT = expT * exp_type * int

and expT = OperationT of annExpT * binary_operation * annExpT
    | MethodCallT of annExpT * id * (annExpT list)
    | IntegerT of int | TrueT | FalseT
    | VarT of id | FieldT of id | FieldRef of int
    | ThisT | NewIdT of id | NewIdAlloc of id * int
    | NotT of annExpT | NullT
    | StringT of string
    | CvtIntToStringT of annExpT
    | CvtBoolToStringT of annExpT

and binary_operation = And | Or | LessThan | Plus | Minus
    | Multiplication | Division | Equals | Strcat

and exp_type = ArrayType of exp_type | BoolType | IntType
    | ObjectType of id | StringType | FloatType
```

Figure 1: MiniJava abstract syntax (type-checked version).

```
type stackloc = int
and  instrloc = int
and  classname = string
and  methodname = string

type ml_instr =
     MOV of stackloc * stackloc
   | LOADIMM of stackloc * int
   | ADD of stackloc * stackloc * stackloc
   | SUB of stackloc * stackloc * stackloc
   | MULT of stackloc * stackloc * stackloc
   | DIV of stackloc * stackloc * stackloc
   | LESS of stackloc * stackloc * stackloc
   | AND of stackloc * stackloc * stackloc
   | OR of stackloc * stackloc * stackloc
   | EQUAL of stackloc * stackloc * stackloc
   | JUMP of instrloc
   | CJUMP of stackloc *  instrloc *  instrloc
   | INT2STRING of stackloc * stackloc
   | BOOL2STRING of stackloc * stackloc
   | GETFLD of stackloc * int
   | PUTFLD of int *  stackloc
   | NEWSTRING of stackloc * string
   | CATSTRINGS of stackloc *  stackloc *  stackloc
   | NEWOBJECT of stackloc * classname * int
   | RETURN of stackloc
   | LOADRESULT of stackloc
   | INVOKE of stackloc * methodname * (stackloc list)
   | JUMPIND of stackloc
   | NEWARRAY of stackloc * stackloc
   | ARRAYREF of stackloc * stackloc * stackloc
```

Figure 2: Machine instructions (the last three are not used in this assignment)

MOV tgt,src	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src)/tgt], h, t, r)
LOADIMM tgt,i	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[i/tgt], h, t, r)
NEWSTRING tgt,str	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[t/tgt], h[str/t], t+1, r)
ADD tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1)+s(src2)/tgt], h, t, r)
SUB tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1)-s(src2)/tgt], h, t, r)
MULT tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1)*s(src2)/tgt], h, t, r)
DIV tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1)/s(src2)/tgt], h, t, r)
LESS tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1)<s(src2)/tgt], h, t, r)
AND tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1) && s(src2)/tgt], h, t, r)
OR tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1) || s(src2)/tgt], h, t, r)
EQUAL tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[s(src1) = s(src2)/tgt], h, t, r)
CATSTRINGS tgt,src1,src2	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[t/tgt], h[str1+str2/t], t+1, r) where h(s(src1)) = str1 and h(s(src2)) = str2
JUMP iloc	(p, c, s, h, t, r)	$\mapsto$	(iloc, c, s, h, t, r)
CJUMP src,iloct,ilocf	(p, c, s, h, t, r)	$\mapsto$ $\mapsto$	(iloct, c, s, h, t, r), if s(src)==1 (ilocf, c, s, h, t, r) if s(src)==0
JUMPIND src	(p, c, s, h, t, r)	$\mapsto$	(s(src), c, s, h, t, r)
INT2STRING tgt,src	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[int2str(s(src))/tgt], h, t, r)
BOOL2STRING tgt,src	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[bool2str(s(src))/tgt], h, t, r)
RETURN src	(p, c, (f,p',c')::s, h, t, r)	$\mapsto$	(p', c', s, h, t, f(src))
LOADRESULT tgt	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[r/tgt], h, t, r)
GETFLD tgt,i	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[flds(i)/tgt], h, t, r) where h(s(0)) = Obj(C,flds)
PUTFLD i,src	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s, h[Obj(C,flds[s(src)/i])/s(0)], t, r) where h(s(0)) = Obj(C,flds)
NEWOBJECT tgt,C,i	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[t/tgt], h[obj/t], t+1, r) where obj = Obj(C,[0,0,...,0]) (i times)
ARRAYREF tgt,src,indx	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[i/tgt], h, t, r) where h(s(src)) = Array values, and i = values[s(indx)]
NEWARRAY tgt,src	(p, c, s, h, t, r)	$\mapsto$	(p+1, c, s[t/tgt], h[arr/t], t+1, r) where arr = Array([0,0,...,0]) (s(src) times)
INVOKE rcvr,m,args	(p, c, s, h, t, r)	$\mapsto$	(0, c', (f,p,c)::s, h, t, r) where f = [s(rcvr),s(args0), ..., s(argsn), 0, 0, ... 0], h(s(rcvr)) = Obj(C, ...), and C's definition of m has code c' and frame size fs, and |f| = fs.

Figure 3: Machine specification

τ f ($args$) $\{locals; S_1; \ldots S_n;$ `return` $e;\} \rightsquigarrow il_1$ @ il_2 @ $\ldots$ @ il_n @ il @ [RETURN loc]
$S_1, 0 \rightsquigarrow il_0, m_1$
$S_2, m_1 \rightsquigarrow il_1, m_2$
$\vdots$
$S_n, m_{n-1} \rightsquigarrow il_n, m_n$
e, loc $\rightsquigarrow il$

Figure 4: Compilation rule for methods

x=e, $m \rightsquigarrow il$ @ [MOV(addr x, loc)], m + $|il|$ + 1 (x a variable)
e, loc $\rightsquigarrow il$

x=e, $m \rightsquigarrow il$ @ [PUTFLD(offset x, loc)], m + $|il|$ + 1 (x a field)
e, loc $\rightsquigarrow il$

$\{ S_1 \ldots S_n \}, m \rightsquigarrow il_1$ @ $\ldots$ @ il_n, m_n
$S_1, m \rightsquigarrow il_1, m_1$
$S_2, m_1 \rightsquigarrow il_2, m_2$
$\vdots$
$S_n, m_{n-1} \rightsquigarrow il_n, m_n$

if (e) S_1 else S_2, $m \rightsquigarrow il$ @ [CJUMP loc,$m + |il| + 1, m' + 1$] @ il_1 @ [JUMP m''] @ il_2, m''
e, loc $\rightsquigarrow il$
$S_1, m + |\text{il}| + 1 \rightsquigarrow il_1, m'$
$S_2, m' + 1 \rightsquigarrow il_2, m''$

Figure 5: Compilation rules for statements

IntegerT i, loc $\rightsquigarrow$ [LOADIMM(loc,i)]

StringT s, loc $\rightsquigarrow$ [NEWSTRING(loc, s)]

TrueT, loc $\rightsquigarrow$ [LOADIMM(loc,1)]

FalseT, loc $\rightsquigarrow$ [LOADIMM(loc,0)]

NullT, loc $\rightsquigarrow$ [LOADIMM(loc,0)]

VarT id, loc $\rightsquigarrow$ []

ThisT, 0 $\rightsquigarrow$ []

NotT e, loc $\rightsquigarrow$ il_0 @ [LOADIMM(loc,1);SUB(loc,loc,loc')]
    e, loc' $\rightsquigarrow$ il_0

OperationT(e_1,bop,e_2), loc $\rightsquigarrow$ il_1 @ il_2 @ [BOP loc,loc1,loc2] (see explanation of bop vs. BOP in section 8)
    e_1, loc1 $\rightsquigarrow$ il_1
    e_2, loc2 $\rightsquigarrow$ il_2

CvtIntToString e, loc $\rightsquigarrow$ il @ [INT2STRING loc,loc1]
    e, loc1 $\rightsquigarrow$ il

CvtBoolToString e, loc $\rightsquigarrow$ il @ [BOOL2STRING loc,loc1]
    e, loc1 $\rightsquigarrow$ il

FieldRef n, loc $\rightsquigarrow$ [GETFLD loc,n]

NewIdAlloc(c,sz), loc $\rightsquigarrow$ [NEWOBJECT(loc,c,sz)]

MethodCallT(e_0,f,[e_1,...,e_n]) $\rightsquigarrow$ il_0 @ ... @ il_n @ [INVOKE(loc0,f,[loc1;...;locn]); LOADRESULT loc]
    e_0, loc0 $\rightsquigarrow$ il_0
    e_1, loc1 $\rightsquigarrow$ il_1
    $\vdots$
    e_n, locn $\rightsquigarrow$ il_n

Figure 6: Compilation rules for expressions