MP 6 – Code Generation

CS 421 – Spring 2011

Revision 1.0

Assigned March 1, 2011
Due March 7, 2011 23:59
Extension 48 hours (20% penalty)
Total Points 50 (+10 extra credit)

1 Change Log

1.0 Initial Release.

2 Overview

In this MP you will generate executable "machine code" for a given program.

3 Collaboration

Collaboration is NOT allowed on this assignment.

4 What to submit

You will submit your `mp6.ml` file using the `handin` program as usual.

5 AST Structure

The input to the function you write will be the abstract syntax of a MiniJava program. The AST structure is the one we have been working on since MP2.

```
type program = Program of (class_decl list)

and class_decl = Class of id * id
        * (var_decl list)
        * (method_decl list)

and var_decl = Var of var_kind * exp_type * id

and var_kind = Static | NonStatic

and method_decl = Method of exp_type
        * id
        * ((exp_type * id) list)
        * (var_decl list)
        * (statement list)
        * exp

and statement = Block of (statement list)
    | If of exp * statement * statement
    | While of exp * statement
    | Println of exp
    | Assignment of id * exp
    | ArrayAssignment of id * exp * exp
    | Break
    | Continue
    | Switch of exp
        * ((int * (statement list)) list)   (* cases *)
        * (statement list)   (* default *)

and exp = Operation of exp * binary_operation * exp
    | Array of exp * exp
    | Length of exp
    | MethodCall of exp * id * (exp list)
    | Integer of int
    | True
    | False
    | Id of id
    | This
    | NewArray of exp_type * exp
    | NewId of id
    | Not of exp
    | Null
    | String of string
    | Float of float

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

and exp_type = ArrayType of exp_type
    | Boolean
    | IntType
    | ObjectType of id
    | FloatType

and id = string
```

You will not be generating code for the entire abstract syntax; instead, we impose a number of restrictions:

- The input program is type-correct; there are no ill-typed statements or expressions. There are no dangling break or continue statements.
- The program consists of exactly one class, and that class contains a method named "main".
- There are no fields in the input class.
- Only integer variables and values exist. There are no strings, floats, or booleans. Boolean values will be represented by integers (as in C): 1 will stand for true, and 0 for false.
- Every function has exactly one argument (of type int), and has a result of type int.
- No arrays or object. In particular, programs will not include array assignment statements, and will not include expressions involving array references, the length operator, this, new, or null.
- No switch statement.

Because there are no objects, the method call notation `e.f(a)` does not really make sense; however, we do not want to change the abstract syntax, so you should just ignore the first argument of the MethodCall constructor. (For example, just put any identifier in that position.)

6 The Low Level Language

We will generate code in a made-up low level language that we call LowLevel. In this language, methods are represented as a triple: method name, parameter list, and list of instructions, with obvious meanings. Below is the definition of instructions.

```
type memloc = string
and  label = string
and  name = string

type instr =
   MOVE of expr * memloc
 | LABEL of label
 | JUMP of label
 | CJUMP of memloc * label * label
 | PRINT of memloc
 | RET of memloc

and expr =
   CONST of int
 | BINOP of binary_operation * memloc * memloc
 | CALL of name * memloc
 | LOAD of memloc

type methodLL = name * memloc list * instr list

type prog = methodLL list
```

The instructions have the following meanings:

- MOVE(e,m): Evaluate e, which is a LowLevel expression, and put that value into the destination m.
- LABEL(l): Mark the current execution point with the label l.
- JUMP(l): Continue execution from the point labeled l.
- CJUMP(m,l_1,l_2): If the value kept in the location m is a non-zero value (meaning true), continue execution from the point labeled l_1. Otherwise continue execution from the point labeled l_2.
- PRINT(m): Print the value kept in the location m.
- RET(m): Read the value kept in the location m and return it as the value of the method.

Note that the MOVE instruction contains a Low Level expression. An expression is defined as:

- CONST(i): The constant value of integer i.
- BINOP(p, m_1,m_2): The binary operation p as applied to the values kept in the locations m_1 and m_2.
- CALL(f,m): Execute the instructions of the method f with the argument of f bound to the value kept in location m.
- LOAD(m): Read the value kept in the location m.

You will be provided with an emulator for LowLevel. The emulator gets an input of type `prog`, which is a list of methods (i.e. list of triples).

```
type methodLL = name * memloc list * instr list
and prog = methodLL list
```

Note that parameters are also memory locations. Methods are assumed to have only one parameter. The emulator starts execution from the method named "main" by giving it the argument 0.

Example

Consider the program

```
    class A {public int main(int i){ System.out.println(i); return i;}}
```

Following code may be generated for the main method:

```
# let insts = [MOVE (LOAD "i", "t1"); PRINT "t1"; RET "t1"];;
val insts : Mp6common.instr list =
  [MOVE (LOAD "i", "t1"); PRINT "t1"; RET "t1"]
```

This list of instructions has the meaning "Put the value kept in the location `i` into the location `t1`, print `t1`, and return `t1`". We can put this code together with the argument list (`["i"]`) into a program:

```
# let prog =  [("main", ["i"], insts)];;
val prog : (string * string list * Mp6common.instr list) list =
  [("main", ["i"], [MOVE (LOAD "i", "t1"); PRINT "t1"; RET "t1"])]
```

Now, execute the program. (Recall that the execution starts with the "main" method, passing 0 as the argument.) `execProg` is the execution function provided by the emulator.

```
# execProg prog;;
0
- : int = 0
```

You do not need to worry about how the emulator works. However, you need to understand the meanings of the instructions.

7 The Mechanics

Here are the instructions of this MP:

- Download `mp6grader.tar.gz`. This tarball contains all the files you need, including the MiniJava lexer and parser.
- As always, extract the tarball, rename `mp6-skeleton.ml` to `mp6.ml` and start modifying the file. You will modify only the `mp6.ml` file, and submit this file only.
- Compile your solution with `make`. Run the `./grader` to see how well you do.
- Make sure to add several more test cases to the `tests` file.
- The following will allow you to run the solution interactively:

```
        Objective Caml version 3.12.0

# #load "mp6common.cmo";;
# #load "minijavaparse.cmo";;
# #load "minijavalex.cmo";;
# #load "solution.cmo";;
# open Mp6common;;
# let parse s = Minijavaparse.program Minijavalex.tokenize (Lexing.from_string s);;
val parse : string -> Mp6common.program = <fun>
# let compile s = Solution.compile (parse s);;
val compile : string -> Mp6common.prog = <fun>
# let eval s = Mp6common.execProg (compile s);;
val eval : string -> int = <fun>
```

Now, to execute a program you can do

```
# eval "class A {public int main(int i){ return i;}}" ;;
- : int = 0
```

To see the LowLevel code generated for methods, compile the program and obtain the `prog` list.

```
# compile "class A {public int main(int i){ return i;}}";;
- : Mp6common.prog =
[("main", ["i"], [LABEL "label1"; MOVE (LOAD "i", "t1"); RET "t1"])]
```

You may want to copy and paste these commands to a file named `init.ml`, and `#use "init.ml"` for your convenience. If you replace "solution" with "student", you will be able to do the same for your own code. Note that in this case, each time you change your code, you will have to first `make`, then re-load the "student.cmo" file and re-define the compile and eval functions.

The common file

You are provided with two helper functions in the common file: `genloc` and `newlabel`. These functions take unit as their argument, and return a fresh string that you can use as a memory location or a label, respectively.

```
# genloc;;
- : unit -> string = <fun>
# genloc();;
- : string = "t138"
# genloc();;
- : string = "t139"
# newlabel;;
- : unit -> string = <fun>
# newlabel();;
- : string = "label66"
# newlabel();;
- : string = "label67"
```

There are no limits in your use of these methods. You may assume that you have infinite memory; you can generate as many new locations or labels as you want. (You are practically limited with the highest integer value that can be represented in OCaml. After that value you will start getting same strings. We will use small enough programs to test your files; you do not need to worry about running out of fresh locations unless you abuse the functions.)

The skeleton file

The skeleton file provided for you contains empty-bodied functions to generate code for each of: an expression, a statement, a method, a class, and a program.

compExpr takes the AST of an expression and returns a pair: The LowLevel location that the value of the expression is put in, and the list of LowLevel instructions generated for the expression.

```
# let rec compExpr e = ...;;
val compExpr : Mp6common.exp -> Mp6common.memloc * Mp6common.instr list =
  <fun>
```

compStmt takes the AST of a statement, the continue label that the execution should jump to if a `continue` statement is encountered, and the break label that the execution should jump to if a `break` statement is encountered. It returns the list of LowLevel instructions generated for the statement.

```
# let rec compStmt stmt contlab breaklab = ...;;
val compStmt :
  Mp6common.statement ->
  Mp6common.label -> Mp6common.label -> Mp6common.instr list = <fun>
```

compMethod takes the AST of a method and returns a triple, where the first element is the name of the method, second element is the list of parameters, and the third is the list of the LowLevel instructions generated for the body of the method.

```
# let compMethod (Method(ty,(Identifier f),args,vars,stmts,e)) = ...;;
val compMethod : Mp6common.method_decl -> Mp6common.methodLL = <fun>
```

compClass takes the AST of a class, and returns a `prog`: a list of triples where each triple represents a method.

```
# let compClass (Class(_, _, vars, methods)) =
val compClass : Mp6common.class_decl -> Mp6common.prog = <fun>
```

compile takes the AST of a program, and returns the `prog` of the class contained in the program. This function has been implemented for you. This is the top-level function we will call when grading your solution.

8 The Assignment

Step 1

Implement the `compExpr : Mp6common.exp -> Mp6common.memloc * Mp6common.instr list` function. You may generate new locations and labels if needed. The following expressions will not occur: array access, array creation, `length`, `this`, object creation (with `new`), `null`, String and float literals. You do not need to handle them.

You should handle the boolean true as the constant 1, and false as the constant 0. Because there is only one class with no fields, the target object of a method invocation is ignored. So, effectively `a.m()` is considered just like `m()` (which would fail parsing with our parser).

What you should return is the list of LowLevel instructions that would calculate the input expression when executed, and the name of the location of the expression's value.

Step 2

Implement the `compStmt : Mp6common.statement -> Mp6common.label -> Mp6common.label -> Mp6common.instr list` function. You may generate new locations and labels if needed. The following statements will not occur: array assignment, switch. You do not need to handle them.

Step 3

Implement the `compMethod : Mp6common.method_decl -> Mp6common.methodLL` function. You may generate new locations and labels if needed. Assume that all the methods return int, and that they have exactly one parameter which is of type int. You do not need to look at variable declarations; any variables that occur in the program can be assumed to have declared with type int.

In this function, you might need to convert Mp6common.id into Mp6common.name and Mp6common.id into Mp6common.memloc. You can use `id_to_name` and `id_to_memloc` which are defined in mp6common.ml.

Step 4

Implement the `compClass : Mp6common.class_decl -> Mp6common.prog` function. This function simply generates code for each method, puts the method representations in a list, and returns the list.

Extra Credit

Implement logical AND and OR operations with short-circuit evaluation. When testing for extra credit, you need to put cases in the `extra_rubric` list. Normal rubric does not perform short-circuiting. Also, to check the solution interactively, short-cirtuiting must be turned on:

```
# Mp6common.turnOnShortCircuit();;
- : unit = ()
```

Similarly you may go back to normal evaluation:

```
# Mp6common.turnOffShortCircuit();;
- : unit = ()
```

Point Schema

Code generation for	Approx. points
expression	15
statement	25
method and class	10
Total	50
Extra Credit	10