
MP 7 Addendum: Extra Credit

CS 421 – Spring 2012

Revision 1.0

Assigned Thursday, March 8, 2012

Due Thursday, March 15, 2012, 09:30

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2.0 Fixed while rule in section 4.5.

2 Objective

For this extra credit, you will extend your solution to MP7 in up to four ways:

1. (5 pts) Add while statements.
2. (15 pts) Add `break` and `continue` to while statements.
3. (10 pts) Compile boolean expressions using short-circuit evaluation (version 1).
4. (25 pts) Compile boolean expressions using short-circuit evaluation (version 2)

We have provided compilation rules for each of these problems in the same level of detail as the regular assignment.

After completing these problems, you will have a better understanding of short-circuit evaluation, and, more generally, of “compiling in context.” Note that this is material that will be covered in class and will be on the exam, so doing the extra credit can only help.

This document is an addendum to MP7, and everything said there still applies; in particular, no collaboration is permitted on the extra credit problems.

3 Points

You should submit one solution, and indicate in a comment at the start of the code which problems you are solving. However, this can include only one of problems 1 and 2, and only one of problems 3 and 4. The idea is that problem 2 subsumes problem 1; and 4 subsumes 3. So the possible combinations are: 1, 2, 3, 4, 1 and 3, 2 and 3, 1 and 4, and 2 and 4. In our opinion, problem 3 is somewhat easier than problem 2, while 1 is the easiest and 4 the hardest; so we would suggest solving the problems in order 1, 3, 2, 4. But note that the code for 2 will replace 1, and the code for 4 will replace 3, so if you feel certain you will do all four problems, and are very sure of yourself, you can skip straight to problems 2 and 4, and avoid writing code you will end up discarding. Finally, if you do both 2 and 4, we are offering five extra points, because there is a bit of extra work in combining these.

4 Problems

The extra credit problems require that you change the methods `compileStmt` or `compileExp`, or add the new compilation method `compileBoolExp`.

4.1 While statements (10 pts)

Compiling while statements is a straightforward addition to `compileStmt`. The compilation rule is given in Figure 1. It may look odd — it starts with a jump, and the condition appears *after* the body — but this is the way loops are actually compiled. You can verify that it is correct, and it saves one instruction per iteration over the more obvious compilation rule.

4.2 break and continue statements (20 pts)

While statements can contain `break` and `continue` statements. (`break` is also legal within a `switch` statement, but we are not implementing that; `continue` is legal only within a loop.) They do the following: `break` causes an immediate termination of the loop; `continue` causes an immediate termination of the current iteration; it jumps to the condition and continues as if that iteration had terminated normally.

The difficulty of compiling these statements is that it is impossible to do so with only the information we are providing in the arguments to `compileStmt`. Without knowing the locations of the condition and of the end of the loop, there is no way to generate code for `break` and `continue`. So, these must be passed in as arguments. This changes the type of `compileStmt`.

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

This is reflected in the new rules, which are given in Figure 2. These have the form:

$$S, b, c, m \rightsquigarrow il, m'$$

meaning, “*S* generates code *il* starting at location *m* and ending at location *m'* − 1; any `break` statement that occurs in *S* should jump to location *b*, and any `continue` statement should jump to *c*.” Only the while statement changes in a substantive way; `if` and `block` just pass along the new arguments to their substatements, and assignment doesn’t change at all.

There is a rather tricky issue that comes up here, however, in the rule for the while statement. To generate the body of the while, we need to provide the break and continue targets, which are the code locations just after the final conditional jump and just after We don’t know what these are until we have generated the code for the body. Thus, we need to know the length of the code for the body *before* we can generate that code. The problem is reflected in this subcompilation for the body of the while statement, where *m'* is both an argument and a result:

$$S, m'', m', m + 1 \rightsquigarrow ils, m'$$

This is not an error in the compilation rules. But in implementing the rules, you have to find a way to deal with this circular dependency. We recommend the following, which is inefficient but simple: Call `compileStmt` on the body of the while twice: first give it dummy arguments (zeros) for *b* and *c*, and use this call only to determine the length of the compiled code (which does not depend on the values of these arguments); given this, compile the body again, this time with the correct addresses for those arguments. (The alternative is to call `compileStmt` with dummy arguments as above, and then go back into the code and fix the addresses of the jump targets. This is called “backpatching.” It requires keeping track of every place where a JUMP to the break or continue target occurred, so it is a bit complicated. This is the way it would be done in real life, but we do not recommend it for this assignment.)

One final point on this version of `compileStmt`: This completely replaces the old version of `compileStmt`, which means the call from `compileMethod` needs to be changed also. But for this call, it is okay to just supply dummy arguments (e.g. zeros), because they will never be called: you can assume the type-checker has already ensured that `break` and `continue` can only occur inside while statements.

4.3 Short-circuit evaluation of boolean expressions (v. 1) (10 pts)

The definition of Java and MiniJava (as well as C, C++, and most languages) dictates that boolean connectives `&&` and `||` be “non-strict” — that is, when evaluating expression *e*₁ `&&` *e*₂, *e*₂ should *not* be evaluated if *e*₁ is false, and when evaluating *e*₁ `||` *e*₂, *e*₂ should *not* be evaluated if *e*₁ is true. This part of the definition was respected in MP5 and 6, but not in the basic part of MP7.

Short-circuit evaluation affects only the two non-strict operations `&&` and `||`, but unfortunately it is not possible to implement it simply by changing the clauses for those two operations in `compileExp`. The difficulty is that this expression can only be compiled correctly by including branching instructions in the code, and for that, the location of the code itself needs to be known. It was for this reason that the location argument (m) was included in `compileStmt`, and the location value returned. So we need to do something similar now. Since this is not needed for non-boolean expressions,¹ we will instead add a new method `compileBoolExp`. `compileBoolExp` will be called from `compileStmt` in these circumstances: compiling the condition of an `if`; compiling the right hand-side of an assignment to a variable or field of type `bool`; compiling the condition of a `while` (if you have done problem 1).

The rules for this version of the short-circuit evaluator are given in Figure 3. We use the notation:

$$e, \text{loc}, m \rightsquigarrow_1 il, m'$$

meaning that il will compute the value of e and put the result in loc , and the code will start at location m and end at location $m' - 1$. This is reflected in the type of `compileBoolExp`.

```
compileBoolExp ((ex,te,loc) as ae:annExpT) (m:int) : int * ml_instr list
```

Note that in cases where there is a subexpression of non-boolean type, it simply calls `compileExp`.

This requires a change in `compileStmt`, as explained above: `compileBoolExp` must be called instead of `compileExp` when the statement contains a boolean expression. The actual compilation rules for those statements do not change, except in that the correct code address need to be passed as an argument. The code location returned from the expression can be used, or it can be ignored; it will always be the same as the starting location plus the length of the returned instruction list.

4.4 Short-circuit evaluation of boolean expressions (v. 2) (20 pts)

The first version of short-circuit evaluation gives the language the correct semantics, but is inefficient for this reason: When a boolean expression appears as the condition of an `if` or `while` — which almost all boolean expressions do — its value is calculated and put in a location, and that location is immediately used in a conditional jump instruction. If the expression has the form $e_1 \&\& e_2$, suppose e_1 evaluates to false (i.e. 0). In version 1, we test this and then branch to an instruction that stores the value, and then we again test it in the conditional jump. There is no reason for this — we may as well jump to the false branch as soon as we know the condition is false.

We will instead compile boolean expression using a new rule:²

$$e, m, t, f \rightsquigarrow_2 il, m'$$

where t and f are branch targets. This rule says, “ il is code that will jump to t if e is true, and to f if e is false; the code starts at location m and ends at location $m' - 1$.” We no longer need the location of the result, because we will never store the value of the expression. The compilation rules for this are in Figure 4.³

Again, this is reflected in the type of `compileBoolExp`:

```
compileBoolExp ((ex,te,loc) as ae:annExpT) (m:int) (t:int) (f:int) : int * ml_instr list
```

`compileStmt` needs to be modified to pass the correct arguments to the new version of `compileBoolExp`.

There is a problem here — the same one that came up in compiling `break` and `continue` statements. Look at the rule for `if` statements. To compile the condition, we need to pass in the addresses of the true and false statements, but those can only be compiled by giving them their starting addresses, which depend upon the length of the code for the condition. So we again have a circular dependency in these rules.

The problem is seen in the subcomputation for the condition:

¹This is not strictly true: A boolean expression can appear as a sub-expression of a non-boolean expression in MiniJava in one circumstance: when the boolean expression is an argument in a method call. In that case, `compileExp` should, to be really correct, call `compileBoolExp`. However, as a simplification of the problem, we will ignore this, and allow strict evaluation in that context. So you should leave the existing clauses for boolean expressions in your `compileExp`.

²Note that the m argument appears here just after the expression, unlike the rules for statements in Figure 2, where it appears after the b and c arguments. We mention this in the hope that it will help allay confusion.

³We have omitted assignment to fields from this set of rules, and you should not implement it. There is a technical problem with the allocation of temporary locations: In principle, since boolean expressions no longer compute their values, there is no reason to allocate temporary space, but for field assignment we need a temporary location (since there is no `LOADIMM` instruction for fields). This is not a hard problem to fix, but is still not worth the trouble of fixing here.

While (*e*) *S*, $m \rightsquigarrow [\text{JUMP } m'] @ \text{ils} @ \text{ile} @ [\text{CJUMP } \text{loc}, m + 1, m''], m''$ (where $m'' = m' + |\text{ile}| + 1$)
 $S, m + 1 \rightsquigarrow \text{ils}, m'$
 $e, \text{loc} \rightsquigarrow \text{ile}$

Figure 1: Extra credit: Compilation rule for while

$x = e, b, c, m \rightsquigarrow \text{il} @ [\text{MOV}(\text{addr } x, \text{loc})], m + |\text{il}| + 1$ (*x* a variable)
 $e, \text{loc} \rightsquigarrow \text{il}$

$x = e, b, c, m \rightsquigarrow \text{il} @ [\text{PUTFLD}(\text{offset } x, \text{loc})], m + |\text{il}| + 1$ (*x* a field)
 $e, \text{loc} \rightsquigarrow \text{il}$

$\{ S_1 \dots S_n \}, b, c, m \rightsquigarrow \text{il}_1 @ \dots @ \text{il}_n, m_n$
 $S_1, b, c, m \rightsquigarrow \text{il}_1, m_1$
 $S_2, b, c, m_1 \rightsquigarrow \text{il}_2, m_2$
 $\vdots$
 $S_n, b, c, m_{n-1} \rightsquigarrow \text{il}_n, m_n$

if (*e*) S_1 else $S_2, b, c, m \rightsquigarrow \text{il} @ [\text{CJUMP } \text{loc}, m + |\text{il}| + 1, m' + 1] @ \text{il}_1 @ [\text{JUMP } m''] @ \text{il}_2, m''$
 $e, \text{loc} \rightsquigarrow \text{il}$ or $e, \text{loc}, m \rightsquigarrow_1 \text{il}, _$
 $S_1, b, c, m + |\text{il}| + 1 \rightsquigarrow \text{il}_1, m'$
 $S_2, b, c, m' + 1 \rightsquigarrow \text{il}_2, m''$

While (*e*) $S, b, c, m \rightsquigarrow [\text{JUMP } m'] @ \text{ils} @ \text{ile} @ [\text{CJUMP } \text{loc}, m + 1, m''], m''$ (where $m'' = m' + |\text{ile}| + 1$)
 $S, m'', m', m + 1 \rightsquigarrow \text{ils}, m'$
 $e, \text{loc} \rightsquigarrow \text{ile}$ or $e, \text{loc}, m' \rightsquigarrow_1 \text{ile}, _$

break, $b, c, m \rightsquigarrow [\text{JUMP } b], m + 1$

continue, $b, c, m \rightsquigarrow [\text{JUMP } c], m + 1$

Figure 2: Extra credit: Compilation rule for break and continue statements

$e, m, m', m'' + 1 \rightsquigarrow_2 \text{ile}, m'$

where m' is both an argument and a result.

Again, this is not an error in the rules; rather, the implementation must find a way to generate code that satisfies the rules. We recommend the same solution: generate the code for the boolean expression twice, first to find its length and then to get the actual code. (The alternative is again to use “backpatching,” which is much more complicated and which we therefore do not recommend for this assignment.)

4.5 (2) and (4) (45 pts)

The compilation of while with break and continue, and the short-circuit compilation of boolean expressions, are orthogonal, so combining involves no special difficulties. We will not give all the compilation rules for this case, because it would be tedious; every rule for statements in Figure 4 has to change slightly to add the *b* and *c* arguments. The only rule that changes in an interesting way is the one for while statements, which becomes:

While (*e*) $S, b, c, m \rightsquigarrow [\text{JUMP } m'] @ \text{ils} @ \text{ile}, m''$
 $S, m'', m', m + 1 \rightsquigarrow \text{ils}, m'$
 $e, m', m + 1, m'' \rightsquigarrow_2 \text{ile}, m''$

We are awarding an extra 5 points for implementing this last bit.

$\text{True}, \text{loc}, m \rightsquigarrow_1 il, m + |il|$
 $\text{True}, \text{loc} \rightsquigarrow il$

$\text{False}, \text{loc}, m \rightsquigarrow_1 il, m + |il|$
 $\text{False}, \text{loc}, m \rightsquigarrow il$

$\text{VarT } id, \text{loc} \rightsquigarrow [], m$

$\text{FieldRef } n, \text{loc} \rightsquigarrow [\text{GETFLD } \text{loc}, n], m + 1$

$e_1 < e_2, \text{loc}, m \rightsquigarrow_1 il, m + |il|$
 $e_1 < e_2, \text{loc} \rightsquigarrow il$

$e_1 == e_2, \text{loc}, m \rightsquigarrow_1 il, m + |il|$
 $e_1 == e_2, \text{loc} \rightsquigarrow il$

$e_1 \& e_2, \text{loc}, m \rightsquigarrow_1 il1 @ [\text{CJUMP}(\text{loc}1, m' + 1, m'' + 2)] @ il2 @ [\text{MOV}(\text{loc}, \text{loc}2); \text{JUMP } m'' + 3; \text{LOADIMM}(\text{loc}, 0)], m'' + 3$
 $e_1, \text{loc}1, m \rightsquigarrow_1 il1, m'$
 $e_2, \text{loc}2, m' + 1 \rightsquigarrow_1 il2, m''$

$e_1 || e_2, \text{loc}, m \rightsquigarrow_1 il1 @ [\text{CJUMP}(\text{loc}1, m'' + 2, m' + 1)] @ il2 @ [\text{MOV}(\text{loc}, \text{loc}2); \text{JUMP } m'' + 3; \text{LOADIMM}(\text{loc}, 1)], m'' + 3$
 $e_1, \text{loc}1, m \rightsquigarrow_1 il1, m'$
 $e_2, \text{loc}2, m' + 1 \rightsquigarrow_1 il2, m''$

$\text{NotT } e, \text{loc}, m \rightsquigarrow_1 il @ [\text{LOADIMM}(\text{loc}, 1); \text{SUB}(\text{loc}, \text{loc}, \text{loc}1)], m' + 2$
 $e, \text{loc}1, m \rightsquigarrow_1 il, m'$

Figure 3: Extra credit: Short-circuit compilation rule for boolean expressions, version 1

$\text{While } (e) \ S, m \rightsquigarrow [\text{JUMP } m'] @ \text{ils} @ \text{ile}, m''$
 $\quad S, m + 1 \rightsquigarrow \text{ils}, m'$
 $\quad e, m', m + 1, m'' \rightsquigarrow_2 \text{ile}, m''$

$\text{If } (e) \ S_1 \text{ else } S_2, m \rightsquigarrow \text{ile} @ \text{il1} @ [\text{JUMP } m'''] @ \text{il2}, m'''$
 $\quad e, m, m', m'' + 1 \rightsquigarrow_2 \text{ile}, m'$
 $\quad S_1, m' \rightsquigarrow \text{il1}, m''$
 $\quad S_2, m'' + 1 \rightsquigarrow \text{il2}, m'''$

$\text{x=e}, m \rightsquigarrow \text{il} @ [\text{LOADIMM loc(x),1; JUMP } m' + 3; \text{LOADIMM loc(x),0}], m' + 3$
 $\quad e, m, m', m' + 2 \rightsquigarrow_2 \text{il}, m'$

$\text{True}, m, t, f \rightsquigarrow_2 [\text{JUMP } t], m + 1$

$\text{False}, m, t, f \rightsquigarrow_2 [\text{JUMP } f], m + 1$

$\text{x}, m, t, f \rightsquigarrow_2 [\text{CJUMP loc(x),t,f}], m + 1$

$e_1 < e_2, m, t, f \rightsquigarrow_2 \text{il} @ [\text{CJUMP}(\text{loc}, t, f)], m + |\text{il}| + 1$
 $\quad e_1 < e_2, \text{loc} \rightsquigarrow \text{il}$

$e_1 \& e_2, m, t, f \rightsquigarrow_2 \text{il1} @ \text{il2}, m''$
 $\quad e_1, m, m', f \rightsquigarrow_2 \text{il1}, m'$
 $\quad e_2, m', t, f \rightsquigarrow_2 \text{il2}, m''$

$e_1 || e_2, m, t, f \rightsquigarrow_2 \text{il1} @ \text{il2}, m''$
 $\quad e_1, m, t, m' \rightsquigarrow_2 \text{il1}, m'$
 $\quad e_2, m', t, f \rightsquigarrow_2 \text{il2}, m''$

$!e, m, t, f \rightsquigarrow_2 \text{il}, m'$
 $\quad e, m, f, t \rightsquigarrow_2 \text{il}, m'$

Figure 4: Extra credit: Short-circuit compilation rule for boolean expressions, version 2