Compilers: Sample practical report

Mike Spivey

Michaelmas Term, 2017

The Compilers course will be assessed partly by a practical task to be done over the Christmas vacation and written up in a report. The task will build on some of the laboratory exercises done during term. This document contains a sample report on a made-up exercise, as a sample of the kind of report that will be expected. The exercise is to add repeat loops to the compiler from Lab 4, a compiler for a Pascal-like language that targets the ARM processor. As a matter of fact, the exercise is doubly made-up, because the Lab 4 compiler already implements repeat loops, but if they were removed, then this report shows how to put them back.

The actual assessment will also be based on one of the compilers studied in the lab exercises, and may ask for some of the improvements made during the lab sessions, in addition to further work that goes beyond what was asked for in the labs. It may be greater in extent than the task written up here, but it will not be different in kind, and it can be written up in a similar style.

1 The report

The task was to implement repeat loops in the style of Pascal. This report describes briefly the changes made to the compiler and the new test cases that were written. Appendices show the detailed changes made (sample.diff) and the four test cases, including embedded compiler output (rep1–4.p).

1.1 Abstract syntax

A representation of the new kind of statement must be added to the type of abstract syntax tree. In the file tree.mli:

```
type stmt_guts = ...
  | RepeatStmt of stmt * expr
```

The same change needs to be made in the file tree.ml, and the pretty-printer for abstract syntax trees in that file was also extended to print the new kind of statement.

Note that the concrete syntax permits a sequence of statements between repeat and until, and this can be accommodated in the abstract syntax using the *Seq* constructor, so that the body of the repeat construct is a single statement in the abstract syntax.

1.2 Concrete syntax

The lexer and parser are easily extended to implement the new construct. For the lexer, all that is needed is to add repeat and until and keywords in the hash table used to look up each identifier.

```
let symtable =
  Util.make_hash 100
    [ ...; ("repeat", REPEAT); ("until", UNTIL); ... ]
```

In the parser, we add *Repeat* and *Until* as new token types.

```
%token          REPEAT UNTIL
```

The existing categories *stmts* (for a *sequence* of statements) and *expr* can be used to write a production for repeat statements, building the appropriate abstract syntax tree.

```
stmt1 : ...
  | REPEAT stmts UNTIL expr        { RepeatStmt ($2, $4) }
```

1.3 Semantic checks

Following the pattern for other constructs, such as while, the semantic analyser just needs to check recursively the body and the condition, and verify that the condition is a boolean.

```
(* |check_stmt| - check and annotate a statement *)
let rec check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with ...
    | RepeatStmt (body, test) ->
        check_stmt body env alloc;
        let ct = check_expr test env in
        if not (same_type ct boolean) then
          sem_error "boolean expression needed after 'repeat'" []
```

1.4 Translation

The repeat construct can be implemented by jumping code that, as might be expected, contains code for the loop body followed by a conditional jump that leads back to the start if the condition is false. To enable use the translator function *gen_cond*, we place labels lab_1 at the top of the loop and lab_2 at the end; the call *gen_cond test* lab_2 lab_1 then produces code that branches to lab_2 if the condition is true, exiting the loop, and to lab_1 if the condition is false, beginning another iteration.

```
(* |gen_stmt| - generate code for a statement *)
let rec gen_stmt s =
  match s.s_guts with ...
    | RepeatStmt (body, test) ->
        let lab1 = label () and lab2 = label () in
        <SEQ,
          <LABEL lab1>,
          gen_stmt body,
          gen_cond test lab2 lab1,
          <LABEL lab2>>
```

1.5 Code generation

The existing code generator is able to deal with the labels and conditional branches that are used to translate repeat statements. During testing, however, one test case (rep3.p) revealed that sub-optimal code is generated if repeat ... until false is used for an infinite loop, with an exit from the loop body *via* a return statement. Tracing the compiler revealed that the condition was being translated into the optree,

```
<JUMPC (Neq, lab), <CONST 0>, <CONST 0>>,
```

a conditional jump that is taken if $0 \neq 0$, i.e., never. To remove this kind of jump, we can add the following rule to the simplifier in file simp.ml.

```
(* |simp| – simplify an expression tree at the root *)
let rec simp t =
  match t with ...
    | <JUMPC (Neq, lab), <CONST a>, <CONST b>> ->
        if a = b then <NOP> else <JUMP lab>
```

The simplification then leaves a no-op ⟨*NOP*⟩ that blocks the jump optimiser, unless it also is extended with a rule that deletes ⟨*NOP*⟩ trees. In file jumpopt.ml:

```
match !code with ...
  | <NOP> :: _ ->
      delete 0
```

A similar observation applies to a program that uses repeat ... until true for a 'loop' that executes only once (see rep4.p), and the above simplification rule catches this case also.

1.6 Tests

All existing test cases continue to pass. In addition, four tests for the new construct are provided:

- Nested repeat loops.
- A repeat loop with an empty body, but a test with a side-effect that makes progress towards termination.
- An 'infinite' repeat loop with a return statement in the body.
- An once-only repeat loop.

In each case, compiler output embedded in the test case shows that good code is generated. The code shows that repeat loops interact well with register variables, and that CSE is possible between the loop body and the condition.

```
diff --git a/check.ml b/check.ml
--- a/check.ml
+++ b/check.ml
@@ -325,6 +325,11 @@
         if not (same_type ct boolean) then
           sem_error "type mismatch in while statement" [];
         check_stmt body env alloc
+    | RepeatStmt (body, test) ->
+        check_stmt body env alloc;
+        let ct = check_expr test env in
+        if not (same_type ct boolean) then
+          sem_error "type mismatch in repeat statement" []

     | ForStmt (var, lo, hi, body, upb) ->
         let vt = check_expr var env in
diff --git a/jumpopt.ml b/jumpopt.ml
--- a/jumpopt.ml
+++ b/jumpopt.ml
@@ -103,6 +103,8 @@
       | <LABEL lab> :: _ ->
           (* Delete unused labels *)
           if !(ref_count lab) = 0 then delete 0
+      | <NOP> :: _ ->
+          delete 0

       (* Tidy up line numbers *)
       | <LINE m> :: <LINE n> :: _ ->
diff --git a/lexer.mll b/lexer.mll
--- a/lexer.mll
+++ b/lexer.mll
@@ -18,7 +18,8 @@
       ("proc", PROC); ("record", RECORD);
       ("return", RETURN); ("then", THEN); ("to", TO);
       ("type", TYPE); ("var", VAR); ("while", WHILE);
-      ("pointer", POINTER); ("nil", NIL); ("for", FOR);
+      ("pointer", POINTER); ("nil", NIL);
+      ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
       ("elsif", ELSIF); ("case", CASE);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
       ("not", NOT); ("mod", MULOP Mod) ]
diff --git a/parser.mly b/parser.mly
--- a/parser.mly
+++ b/parser.mly
@@ -21,7 +21,7 @@
 %token                 ARRAY BEGIN CONST DO ELSE END IF OF
 %token                 PROC RECORD RETURN THEN TO TYPE
 %token                 VAR WHILE NOT POINTER NIL
-%token                 FOR ELSIF CASE
+%token                 REPEAT UNTIL FOR ELSIF CASE

 %type <Tree.program>   program
 %start                 program
@@ -115,6 +115,7 @@
   | RETURN expr_opt                   { Return $2 }
   | IF expr THEN stmts elses END      { IfStmt ($2, $4, $5) }
   | WHILE expr DO stmts END           { WhileStmt ($2, $4) }
+  | REPEAT stmts UNTIL expr           { RepeatStmt ($2, $4) }
   | FOR name ASSIGN expr TO expr DO stmts END
                                       { let v = make_expr (Variable $2) in
                                         ForStmt (v, $4, $6, $8, ref None) }
diff --git a/simp.ml b/simp.ml
--- a/simp.ml
```

```
+++ b/simp.ml
@@ -36,6 +36,8 @@
          <CONST (do_binop w a b)>
      | <MONOP w, <CONST a>> ->
          <CONST (do_monop w a)>
+     | <JUMPC (Neq, lab), <CONST a>, <CONST b>> ->
+         if a = b then <NOP> else <JUMP lab>
 
      (* Static bound checks *)
      | <BOUND, <CONST k>, <CONST b>> ->
diff --git a/tgen.ml b/tgen.ml
--- a/tgen.ml
+++ b/tgen.ml
@@ -303,6 +303,14 @@
              <JUMP l1>,
              <LABEL l3>>
 
+       | RepeatStmt (body, test) ->
+           let l1 = label () and l2 = label () in
+           <SEQ,
+             <LABEL l1>,
+             gen_stmt body,
+             gen_cond test l2 l1,
+             <LABEL l2>>
+
        | ForStmt (var, lo, hi, body, upb) ->
            (* Use previously allocated temp variable to store upper bound *)
            let tmp = match !upb with Some d -> d | _ -> failwith "for" in
diff --git a/tree.ml b/tree.ml
--- a/tree.ml
+++ b/tree.ml
@@ -35,6 +35,7 @@
    | Return of expr option
    | IfStmt of expr * stmt * stmt
    | WhileStmt of expr * stmt
+   | RepeatStmt of stmt * expr
    | ForStmt of expr * expr * expr * stmt * def option ref
    | CaseStmt of expr * (expr * stmt) list * stmt
 
@@ -135,6 +136,8 @@
          fMeta "(IF $ $ $)" [fExpr test; fStmt thenpt; fStmt elsept]
      | WhileStmt (test, body) ->
          fMeta "(WHILE $ $)" [fExpr test; fStmt body]
+     | RepeatStmt (body, test) ->
+         fMeta "(REPEAT $ $)" [fStmt body; fExpr test]
      | ForStmt (var, lo, hi, body, _) ->
          fMeta "(FOR $ $ $ $)" [fExpr var; fExpr lo; fExpr hi; fStmt body]
      | CaseStmt (sel, arms, deflt) ->
diff --git a/tree.mli b/tree.mli
--- a/tree.mli
+++ b/tree.mli
@@ -50,6 +50,7 @@
    | Return of expr option
    | IfStmt of expr * stmt * stmt
    | WhileStmt of expr * stmt
+   | RepeatStmt of stmt * expr
    | ForStmt of expr * expr * expr * stmt * def option ref
    | CaseStmt of expr * (expr * stmt) list * stmt
```

```
var i, j, k: integer;
begin
  i := 0;
  repeat
    j := 1;
    repeat
      j := j+1; k := k+1;
    until j > i;
    i := i+1
  until i > 10;
  print_num(k); newline()
end.

(*<<
56
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ i := 0;
        mov r0, #0
        ldr r1, =_i
        str r0, [r1]
.L2:
@ j := 1;
        mov r0, #1
        ldr r1, =_j
        str r0, [r1]
.L4:
@ j := j+1; k := k+1;
        ldr r4, =_j
        ldr r0, [r4]
        add r5, r0, #1
        str r5, [r4]
        ldr r4, =_k
        ldr r0, [r4]
        add r6, r0, #1
        str r6, [r4]
@ until j > i;
        ldr r4, =_i
        ldr r7, [r4]
        cmp r5, r7
        ble .L4
@ i := i+1
        add r5, r7, #1
        str r5, [r4]
        cmp r5, #10
        ble .L2
@ print_num(k); newline()
        mov r0, r6
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool
```

```
        .comm _i, 4, 4
        .comm _j, 4, 4
        .comm _k, 4, 4
        .section .note.GNU-stack
@ End
]]*)
```

```
var i: integer;

proc inc(var x: integer): integer;
begin
  x := x+1;
  return x
end;

begin
  i := 0;
  repeat until inc(i) > 10;
  print_num(i); newline()
end.

(*<<
11
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc inc(var x: integer): integer;
        .text
_inc:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ x := x+1;
        ldr r4, [fp, #40]
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
@ return x
        ldr r0, [fp, #40]
        ldr r0, [r0]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ i := 0;
        mov r0, #0
        ldr r1, =_i
        str r0, [r1]
.L3:
@ repeat until inc(i) > 10;
        ldr r4, =_i
        mov r0, r4
        bl _inc
        cmp r0, #10
        ble .L3
@ print_num(i); newline()
        ldr r0, [r4]
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool
```

```
        .comm _i, 4, 4
        .section .note.GNU-stack
@ End
]]*)
```

```
proc foo(): integer;
  var i: integer;
begin
  i := 3;
  repeat
    i := i + 2;
    if i > 10 then return i end;
  until false
end;

begin
  print_num(foo()); newline()
end.

(*<<
11
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc foo(): integer;
        .text
_foo:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ i := 3;
        mov r4, #3
.L2:
@ i := i + 2;
        add r4, r4, #2
@ if i > 10 then return i end;
        cmp r4, #10
        ble .L2
        mov r0, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_num(foo()); newline()
        bl _foo
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .section .note.GNU-stack
@ End
]]*)
```

```
begin
  repeat
    print_string("Hello"); newline()
  until true
end.

(*<<
Hello
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_string("Hello"); newline()
        mov r1, #6
        ldr r0, =__s1
        bl print_string
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .data
__s1:
        .byte 72, 101, 108, 108, 111
        .byte 0
        .section .note.GNU-stack
@ End
]]*)
```