University of Oxford

Department of Computer Science

Compilers – Model solution to assignment

December 2017

1 Abstract syntax

A new kind of statement replaces the old for statement as a constructor in the abstract syntax tree. Note the slot provided for allocating a hidden variable.

```
type stmt_guts = ...
  | ForStmt of expr * for_elem list * stmt * def option ref

and for_elem =
    OneElem of expr
  | StepElem of expr * expr * expr
  | WhileElem of expr * expr
```

The file tree.ml contains code to pretty-print trees in S-expression format for debugging. This should be enhanced to print the new style of for statement appropriately.

2 Lexer

The lexer needs to be extended with the additional keyword step; the word until is already a keyword because of its use in repeat loops.

3 Parser

The parser needs to be extended with productions for the new construct. The parsing is straightforward. As suggested, to is allowed as an abbreviation for step 1 until.

```
stmt1 : ...
  | FOR variable ASSIGN forelems DO stmts END
                                  { ForStmt ($2, $4, $6, ref None) }

forelems :
    forelem                       { [$1] }
  | forelem COMMA forelems        { $1 :: $3 } ;

forelem :
    expr                          { OneElem $1 }
  | expr STEP expr UNTIL expr     { StepElem ($1, $3, $5) }
  | expr WHILE expr               { WhileElem ($1, $3) }
  | expr TO expr                  { StepElem ($1, const 1 integer, $3) } ;
```

4 Semantic analysis

We simply have to walk over the construct, checking that all sub-expressions have integer type (except the test in a while element, which is a boolean). The quality of error messages here is no worse than in the rest of the compiler. We also check that the controlled variable really is a variable, and recursively check the loop body. The final step is to allocate space for the hidden integer variable that will indicate the currently active for list element.

```
let rec check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with ...
    | ForStmt (var, elems, body, t) →
        let check_elem =
          function
              OneElem e →
                let t = check_expr e env in
                if not (same_type t integer) then
                  sem_error "type error in for list element" [ ];
            | StepElem (e1, e2, e3) →
                let t1 = check_expr e1 env in
                let t2 = check_expr e2 env in
                let t3 = check_expr e3 env in
                if not (same_type t1 integer) || not (same_type t2 integer)
                    || not (same_type t3 integer) then
                  sem_error "type error in for range" [ ];
            | WhileElem (e1, e2) →
                let t1 = check_expr e1 env in
                let t2 = check_expr e2 env in
                if not (same_type t1 integer)
                    || not (same_type t2 boolean) then
                  sem_error "type error in while element" [ ] in

        let vt = check_expr var env in
        if not (same_type vt integer) then
          sem_error "type mismatch in for statement" [ ];
        check_var var false;
        List.iter check_elem elems;
        check_stmt body env alloc;

        if List.length elems > 1 then begin
          let d = make_def (intern "*t*") VarDef integer in
          alloc d; t := Some d
        end
```

5 Code generation

A workable code generation scheme is (as suggested) to keep a hidden variable that indicates which element of the for list is currently active. Code for the loop is in three sections:

- First, code for the list elements. Each element sets the value of the

controlled variable, then (if the element is not exhausted) branches to the loop body. An element that is exhausted falls through to the next element. Each element defines a label to which control should return after executing the loop body.

- Second, code for the loop body.
- Third, a jump table that uses the value of the hidden variable to jump back to the appropriate list element.

To translate a for loop, we first invent labels for the loop body and for the exit from the loop. If there is only one element in the for list, we treat it as a special case; otherwise, we make an index that pairs the integers 0, 1, . . . , $n-1$ with labels that will be used to return to the for list elements.

```
let rec gen_stmt s =
  let code =
    match s.s_guts with ...
    | ForStmt (var, elems, body, t) →
        let bodylab = label () and exitlab = label () in

        let gen_elem el nextlab = ... in     (* see below *)

        let n = List.length elems in
        if n = 1 then begin
          (* special case – see below *)
        end else begin
          let tmp =
            match !t with Some d → d | _ → failwith "for" in
          let index =
            List.map (fun i → (i, label ())) (Util.range 0 (n−1)) in
```

According to the scheme outlined above, the code consists of a section for the list elements, the loop body, and a jump table that uses the value of *tmp* to return to the appropriate list element. Each element of the for list corresponds to code that first sets *tmp* to the correct value, followed by code specific to the kind of element.

```
⟨SEQ,
  ⟨SEQ, @(List.map2 (fun (i, nextlab) elem →
              ⟨SEQ,
                ⟨STOREW, ⟨CONST i⟩,     (* tmp := i *)
                  address tmp⟩,
                gen_elem elem nextlab⟩) index elems)⟩,
  ⟨JUMP exitlab⟩,
  ⟨LABEL bodylab⟩,
  gen_stmt body,
  ⟨JCASE (List.map snd index, exitlab),
    ⟨LOADW, address tmp⟩⟩,
  ⟨LABEL exitlab⟩⟩
end
```

The detailed code for each kind of element is generated by a helper function *gen_elem*. For simple expressions, the first execution sets the controlled variable to the value of the expression, then enters the loop body. When

control returns from the body to label *nextlab*, we simply fall through to the next element of the list.

```
let gen_elem el nextlab =
  match el with
      OneElem e →
        ⟨SEQ,
          ⟨STOREW, gen_expr e, gen_addr var⟩,     (* var := e *)
          ⟨JUMP bodylab⟩,
          ⟨LABEL nextlab⟩⟩
```

For an element *lo* step *st* until *hi*, the first execution sets the controlled variable to *lo*, and subsequent executions increase it by the value of *st*. In both cases, the value is then compared with the value of *hi*, with the sense of the comparison depending on the sign of *st*.

```
    | StepElem (lo, st, hi) →
        let lab1 = label () and lab2 = label () and lab3 = label () in
        ⟨SEQ,
          ⟨STOREW, gen_expr lo, gen_addr var⟩,     (* var := lo *)
          ⟨JUMP lab3⟩,
          ⟨LABEL nextlab⟩,
          ⟨STOREW,                  (* var := var + st *)
            ⟨BINOP Plus, gen_expr var, gen_expr st⟩,
            gen_addr var⟩,
          ⟨LABEL lab3⟩,
          ⟨JUMPC (Leq, lab1),      (* if st ≤ 0 goto lab1 *)
            gen_expr st, ⟨CONST 0⟩⟩,
          ⟨JUMPC (Leq, bodylab),     (* if var ≤ hi goto body *)
            gen_expr var, gen_expr hi⟩,
          ⟨JUMP lab2⟩,
          ⟨LABEL lab1⟩,
          ⟨JUMPC (Geq, bodylab),     (* if var ≥ hi goto body *)
            gen_expr var, gen_expr hi⟩,
          ⟨LABEL lab2⟩⟩
```

Note that, in the common case where *st* is a constant, the conditional jump after label lab_3 will always go the same way. We will rely on the optimiser to remove such jumps.

For a while element, every execution sets the controlled variable and then performs the test, implemented here using the compiler function *gen_cond*.

```
    | WhileElem (e1, e2) →
        let lab1 = label () in
        ⟨SEQ,
          ⟨LABEL nextlab⟩,
          ⟨STOREW, gen_expr e1, gen_addr var⟩,     (* var := e1 *)
          gen_cond e2 bodylab lab1,     (* if e2 goto body *)
          ⟨LABEL lab1⟩⟩ in
```

The code generated by the above scheme is needlessly complicated when there is only one element in the list. In that case, we can dispense with the hidden variable and the jump table, and use just a simple jump.

```
if n = 1 then begin
  let nextlab = label () in
  ⟨SEQ, gen_elem (List.hd elems) nextlab,
    ⟨JUMP exitlab⟩,
    ⟨LABEL bodylab⟩,
    gen_stmt body,
    ⟨JUMP nextlab⟩,
    ⟨LABEL exitlab⟩⟩
end else ...
```

This improvement was anticipated by not allocating the hidden variable in the case where there is only one element. That refinement is unimportant if all we are saving is one word in the stack frame, but becomes more significant if the hidden variable may be allocated to a register.

6 Optimisation

As noted above, in the most common case the code contains a conditional branch that always goes the same way. This branch can be eliminated in the simplification phase of the compiler by adding the following rule in simp.ml.

```
let rec simp t =
  match t with ...
    | ⟨JUMPC (w, lab), ⟨CONST a⟩, ⟨CONST b⟩⟩ →
        if do_binop w a b ≠ 0 then ⟨JUMP lab⟩ else ⟨NOP⟩
```

This rule replaces the conditional branch with either an unconditional branch or a no-op. In either case, code in one side of the conditional becomes inaccessible, and will be deleted by the jump optimiser. The results of this can be seen in the test cases below.

7 Label values

In the code shown in Section 5, the variable *tmp* takes small integer values that are indices into a jump table at the bottom of the loop. As an alternative (Plan B), *tmp* may be given values that are code addresses, with the jump table replaced by an indirect jump.

To implement this, we need to add two new operations to the intermediate language: one to get as a value the address associated with a code label, and another to jump to a specified address. We will use the following forms:

- As an expression ⟨*LABEL lab*⟩ evaluates to the address of label *lab*.[1]
- As a statement ⟨*IJUMP*, *t*⟩ is an indirect jump to the address that is that value of *t*.

We then replace the code pattern,

```
let index = List.map (fun i → (i, label ())) (Util.range 0 (n−1)) in
⟨SEQ,
```

[1] This is in addition to the use of ⟨*LABEL lab*⟩ as a statement to place the label *lab*.

```
⟨SEQ, @(List.map2 (fun (i, nextlab) elem →
            ⟨SEQ, ⟨STOREW, ⟨CONST i⟩, address tmp⟩,
              gen_elem elem nextlab⟩) index elems)⟩,
⟨JUMP exitlab⟩,
⟨LABEL bodylab⟩,
gen_stmt body,
⟨JCASE (List.map snd index, exitlab),
  ⟨LOADW, address tmp⟩⟩,
⟨LABEL exitlab⟩⟩
```

with the following.

```
⟨SEQ,
  ⟨SEQ, @(List.map (fun elem →
            let nextlab = label () in
            ⟨SEQ, ⟨STOREW, ⟨LABEL nextlab⟩, address tmp⟩,
              gen_elem nextlab elem⟩) elems)⟩,
  ⟨JUMP exitlab⟩,
  ⟨LABEL bodylab⟩,
  gen_stmt body,
  ⟨IJUMP, ⟨LOADW, address tmp⟩⟩,
  ⟨LABEL exitlab⟩⟩
```

The code is simpler because the mapping *index* between integer indices and labels is no longer needed.

To implement the new operations, we must add to the translation phase. In *e_reg* we add a case for ⟨LABEL *lab*⟩.

```
let rec eval_reg t r =
  match t with ...
    | ⟨LABEL lab⟩ →
        gen_reg "ldr $, =$" [register r; codelab lab]
```

Then in *e_stmt*, we add cases for ⟨IJUMP, *t*⟩.

```
let rec exec_stmt t =
  match t with ...
    | ⟨IJUMP, ⟨LABEL lab⟩⟩ →
        gen "b" [codelab lab]
    | ⟨IJUMP, t1⟩ →
        let v1 = eval_reg t1 R_any in
        gen "bx" [v1]
```

The first rule is not used in the implementation of **for** statements, but is an important short-cut when it applies.

Overall, the code for this alternative is a bit simpler and probably runs with about the same overhead. The price is the need to add to the repertoire of operations supported by the back end.

8 Test cases

Test cases should demonstrate the following:

- Fast, compact code is generated for simple loops;

- Loops with multiple elements in the for list function correctly;
- Nested loops function correctly;
- In a step–until element, the upper bound is re-evaluated before each iteration;
- By providing a step expression with a side-effect, that the step is evaluated *twice* for each iteration.

Attached are four test cases that cover some of these points.

```
--- ../../labs/lab4/check.ml    2023-10-08 15:18:06.767585400 +0100
+++ check.ml    2023-10-08 15:24:34.377061155 +0100
@@ -346,20 +346,38 @@
         if not (same_type ct boolean) then
           sem_error "type mismatch in repeat statement" []

-    | ForStmt (var, lo, hi, body, upb) ->
+    | ForStmt (var, elems, body, t) ->
+        let check_elem =
+          function
+              OneElem e ->
+                let t = check_expr e env in
+                if not (same_type t integer) then
+                  sem_error "type error in for list element" [];
+            | StepElem (e1, e2, e3) ->
+                let t1 = check_expr e1 env in
+                let t2 = check_expr e2 env in
+                let t3 = check_expr e3 env in
+                if not (same_type t1 integer) || not (same_type t2 integer)
+                    || not (same_type t3 integer) then
+                  sem_error "type error in for range" [];
+            | WhileElem (e1, e2) ->
+                let t1 = check_expr e1 env in
+                let t2 = check_expr e2 env in
+                if not (same_type t1 integer)
+                    || not (same_type t2 boolean) then
+                  sem_error "type error in while element" [] in
+
         let vt = check_expr var env in
-        let lot = check_expr lo env in
-        let hit = check_expr hi env in
-        if not (same_type vt integer) || not (same_type lot integer)
-            || not (same_type hit integer) then
+        if not (same_type vt integer) then
           sem_error "type mismatch in for statement" [];
         check_var var false;
+        List.iter check_elem elems;
         check_stmt body env alloc;

-        (* Allocate space for hidden variable.  In the code, this will
-           be used to save the upper bound.  *)
-        let d = make_def (intern "*upb*") VarDef integer in
-        alloc d; upb := Some d
+        if List.length elems > 1 then begin
+          let d = make_def (intern "*t*") VarDef integer in
+          alloc d; t := Some d
+        end

     | CaseStmt (sel, arms, deflt) ->
         let st = check_expr sel env in
--- ../../labs/lab4/jumpopt.ml  2023-10-08 15:18:06.783585291 +0100
+++ jumpopt.ml  2023-10-08 15:28:34.975564184 +0100
@@ -46,6 +46,10 @@
   y2.y_refct := !(y1.y_refct) + !(y2.y_refct);
   Hashtbl.add label_tab y1.y_id (Equiv y2.y_id)

+(* We assume that uses of |LABEL| as a value all appear in the simple
+   form |<STOREW, <LABEL lab>, _>|; otherwise it becomes necessary to
+   traverse the whole of each tree to find and rename potential label refs. *)
+
 (* do_refs -- call function on each label in an instruction *)
 let do_refs f =
```

```
   function
@@ -54,6 +58,10 @@
     | <JCASE (labs, def), _> ->
         List.iter (fun x -> f (ref_count x)) labs;
         f (ref_count def)
+    | <IJUMP, <LABEL x>> ->
+        f (ref_count x)
+    | <STOREW, <LABEL x>, _> -> (* Plan B *)
+        f (ref_count x)
     | _ -> ()

 (* rename_labs -- replace each label by its equivalent *)
@@ -64,6 +72,10 @@
     | <JUMPC (w, x), t1, t2> -> <JUMPC (w, rename x), t1, t2>
     | <JCASE (labs, def), t1> ->
         <JCASE (List.map rename labs, rename def), t1>
+    | <IJUMP, <LABEL x>> ->
+        <IJUMP, <LABEL (rename x)>>
+    | <STOREW, <LABEL x>, t2> -> (* Plan B *)
+        <STOREW, <LABEL (rename x)>, t2>
     | t -> t

 (* optstep -- optimise to fixpoint at current location *)
--- ../../labs/lab4/optree.mli  2023-10-08 15:18:06.787585264 +0100
+++ optree.mli  2023-10-08 15:24:34.377061155 +0100
@@ -55,6 +55,7 @@
   | JUMP of codelab             (* Unconditional branch (dest) *)
   | JUMPC of op * codelab       (* Conditional branch (cond, dest) *)
   | JCASE of codelab list * codelab (* Jump table *)
+  | IJUMP                       (* Indirect jump (Plan B) *)

   (* Artificial instructions *)
   | LINE of int                 (* Line number *)
--- ../../labs/lab4/optree.ml   2023-10-08 15:18:06.771585372 +0100
+++ optree.ml   2023-10-08 15:24:34.377061155 +0100
@@ -66,6 +66,7 @@
   | JUMP of codelab             (* Unconditional branch (dest) *)
   | JUMPC of op * codelab       (* Conditional branch (cond, dest) *)
   | JCASE of codelab list * codelab (* Jump table *)
+  | IJUMP                       (* Indirect jump (Plan B) *)

   (* Extra instructions *)
   | LINE of int                 (* Line number *)
@@ -98,6 +99,7 @@
     | JUMP l ->         fMeta "JUMP $" [fLab l]
     | JUMPC (w, l) ->   fMeta "JUMPC $ $" [fOp w; fLab l]
     | JCASE (labs, def) -> fMeta "JCASE $ $" [fNum (List.length labs); fLab def]
+    | IJUMP ->          fStr "IJUMP"
     | LINE n ->         fMeta "LINE $" [fNum n]
     | NOP ->            fStr "NOP"
     | SEQ ->            fStr "SEQ"
--- ../../labs/lab4/tgen.ml     2023-10-08 15:18:06.787585264 +0100
+++ tgen.ml     2023-10-08 15:24:34.377061155 +0100
@@ -306,21 +306,77 @@
             gen_cond test l2 l1,
             <LABEL l2>>

-      | ForStmt (var, lo, hi, body, upb) ->
-          (* Use previously allocated temp variable to store upper bound.
-             We could avoid this if the upper bound is constant. *)
-          let tmp = match !upb with Some d -> d | _ -> failwith "for" in
-          let l1 = label () and l2 = label () in
```

```
-          <SEQ,
-            <STOREW, gen_expr lo, gen_addr var>,
-            <STOREW, gen_expr hi, address tmp>,
-            <LABEL l1>,
-            <JUMPC (Gt, l2), gen_expr var, <LOADW, address tmp>>,
-            gen_stmt body,
-            <STOREW, <BINOP Plus, gen_expr var, <CONST 1>>, gen_addr var>,
-            <JUMP l1>,
-            <LABEL l2>>
+      | ForStmt (var, elems, body, t) ->
+          let bodylab = label () and exitlab = label () in

+          let gen_elem nextlab =
+          function
+              OneElem e ->
+                <SEQ,
+                  <STOREW, gen_expr e, gen_addr var>,
+                  <JUMP bodylab>,
+                  <LABEL nextlab>>
+            | StepElem (lo, st, hi) ->
+                let lab1 = label () and lab2 = label () and lab3 = label () in
+                <SEQ,
+                  <STOREW, gen_expr lo, gen_addr var>,
+                  <JUMP lab3>,
+                  <LABEL nextlab>,
+                  <STOREW, <BINOP Plus, gen_expr var, gen_expr st>,
+                    gen_addr var>,
+                  <LABEL lab3>,
+                  <JUMPC (Leq, lab1), gen_expr st, <CONST 0>>,
+                  <JUMPC (Leq, bodylab), gen_expr var, gen_expr hi>,
+                  <JUMP lab2>,
+                  <LABEL lab1>,
+                  <JUMPC (Geq, bodylab), gen_expr var, gen_expr hi>,
+                  <LABEL lab2>>
+            | WhileElem (e1, e2) ->
+                let lab1 = label () in
+                <SEQ,
+                  <LABEL nextlab>,
+                  <STOREW, gen_expr e1, gen_addr var>,
+                  gen_cond e2 bodylab lab1,
+                  <LABEL lab1>> in
+
+          let n = List.length elems in
+          if n = 1 then begin
+            let nextlab = label () in
+            <SEQ,
+              gen_elem nextlab (List.hd elems),
+              <JUMP exitlab>,
+              <LABEL bodylab>,
+              gen_stmt body,
+              <JUMP nextlab>,
+              <LABEL exitlab>>
+          end else begin
+            let tmp =
+              match !t with Some d -> d | _ -> failwith "fortmp" in
+(* Plan A:
+              let index =
+                List.map (fun i -> (i, label ())) (Util.range 0 (n-1)) in
+              <SEQ,
+                <SEQ, @(List.map2 (fun (i, lab) elem ->
+                          <SEQ, <STOREW, <CONST i>, address tmp>,
+                            gen_elem lab elem>) index elems)>,
```

```
+              <JUMP exitlab>,
+              <LABEL bodylab>,
+              gen_stmt body,
+              <JCASE (List.map snd index, exitlab), <LOADW, address tmp>>,
+              <LABEL exitlab>>
+*)
+(* Plan B: *)
+            <SEQ,
+              <SEQ, @(List.map (fun elem ->
+                        let lab = label () in
+                        <SEQ, <STOREW, <LABEL lab>, address tmp>,
+                          gen_elem lab elem>) elems)>,
+              <JUMP exitlab>,
+              <LABEL bodylab>,
+              gen_stmt body,
+              <IJUMP, <LOADW, address tmp>>,
+              <LABEL exitlab>>
+          end
       | CaseStmt (sel, arms, deflt) ->
           (* Use one jump table, and hope it is reasonably compact *)
           let deflab = label () and donelab = label () in
--- ../../labs/lab4/lexer.mll   2023-10-08 15:18:06.767585400 +0100
+++ lexer.mll   2023-10-08 15:24:34.377061155 +0100
@@ -18,7 +18,7 @@
       ("type", TYPE); ("var", VAR); ("while", WHILE);
       ("pointer", POINTER); ("nil", NIL);
       ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
-      ("elsif", ELSIF); ("case", CASE);
+      ("elsif", ELSIF); ("case", CASE); ("step", STEP);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
       ("not", NOT); ("mod", MULOP Mod) ]

--- ../../labs/lab4/parser.mly  2023-10-08 15:18:06.763585427 +0100
+++ parser.mly  2023-10-08 15:24:34.377061155 +0100
@@ -22,7 +22,7 @@
 %token                  ARRAY BEGIN CONST DO ELSE END IF OF
 %token                  PROC RECORD RETURN THEN TO TYPE
 %token                  VAR WHILE NOT POINTER NIL
-%token                  REPEAT UNTIL FOR ELSIF CASE
+%token                  REPEAT UNTIL FOR ELSIF CASE STEP

 %type <Tree.program>    program
 %start                  program
@@ -116,9 +116,8 @@
   | 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) }
+  | FOR variable ASSIGN forelems DO stmts END
+                                       { ForStmt ($2, $4, $6, ref None) }
   | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) } ;

 elses :
@@ -126,6 +125,16 @@
   | ELSE stmts                          { $2 }
   | ELSIF line expr THEN stmts elses    { make_stmt (IfStmt ($3,$5,$6), $2) } ;

+forelems :
+    forelem                            { [$1] }
+  | forelem COMMA forelems             { $1 :: $3 } ;
```

```
+
+forelem :
+    expr                                  { OneElem $1 }
+  | expr STEP expr UNTIL expr             { StepElem ($1, $3, $5) }
+  | expr WHILE expr                       { WhileElem ($1, $3) }
+  | expr TO expr                          { StepElem ($1, const 1 integer, $3) } ;
+
 arms :
     arm                                   { [$1] }
   | arm VBAR arms                         { $1 :: $3 } ;
--- ../../labs/lab4/simp.ml     2023-10-08 15:18:06.787585264 +0100
+++ simp.ml     2023-10-08 15:24:34.377061155 +0100
@@ -30,6 +30,8 @@
         <CONST (do_binop w a b)>
     | <MONOP w, <CONST a>> ->
         <CONST (do_monop w a)>
+    | <JUMPC (w, lab), <CONST a>, <CONST b>> ->
+        if do_binop w a b <> 0 then <JUMP lab> else <NOP>

     (* Static bound checks *)
     | <BOUND, <CONST k>, <CONST b>> ->
--- ../../labs/lab4/tran.ml     2023-10-08 15:18:06.783585291 +0100
+++ tran.ml     2023-10-08 15:28:35.947558205 +0100
@@ -99,6 +99,8 @@
         gen_reg "ldr ip, =$1 / add $0, fp, ip" r [number n]
     | <GLOBAL x> ->
         gen_reg "ldr $0, =$1" r [symbol x]
+    | <LABEL lab> ->
+        gen_reg "ldr $0, =$1" r [codelab lab]
     | <TEMP n> ->
         gen_move r (Regs.use_temp n)
     | <(LOADW|LOADC), <REGVAR i>> ->
@@ -241,6 +243,13 @@

     | <JUMP lab> -> gen "b" [codelab lab]

+    | <IJUMP, <LABEL lab>> -> (* Plan B *)
+        gen "b" [codelab lab]
+    | <IJUMP, t1> ->
+        (* Assume evaluation of |t1| writes the destination in one hit: *)
+        let v1 = eval_reg t1 R_any in
+        gen "bx" [v1]
+
     | <JUMPC (Eq, lab), t1, t2> -> condj "eq" lab t1 t2
     | <JUMPC (Lt, lab), t1, t2> -> condj "lt" lab t1 t2
     | <JUMPC (Gt, lab), t1, t2> -> condj "gt" lab t1 t2
--- ../../labs/lab4/tree.ml     2023-10-08 15:18:06.791585236 +0100
+++ tree.ml     2023-10-08 15:24:34.377061155 +0100
@@ -38,9 +38,14 @@
   | IfStmt of expr * stmt * stmt
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
-  | ForStmt of expr * expr * expr * stmt * def option ref
+  | ForStmt of expr * forelem list * stmt * def option ref
   | CaseStmt of expr * (expr * stmt) list * stmt

+and forelem =
+    OneElem of expr
+  | StepElem of expr * expr * expr
+  | WhileElem of expr * expr
+
 and expr =
```

```
   { e_guts: expr_guts;
     mutable e_type: ptype;
@@ -146,12 +151,22 @@
         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]
+    | ForStmt (var, elems, body, _) ->
+        fMeta "(FOR $ (PARTS$) $)"
+          [fExpr var; fTail(fForElem) elems; fStmt body]
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
         fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]

+and fForElem =
+  function
+      OneElem e ->
+        fExpr e
+    | StepElem (e1, e2, e3) ->
+        fMeta "(STEP $ $ $)" [fExpr e1; fExpr e2; fExpr e3]
+    | WhileElem (e1, e2) ->
+        fMeta "(WHILE $ $)" [fExpr e1; fExpr e2]
+
 and fExpr e =
   match e.e_guts with
       Constant (n, t) -> fMeta "(CONST $)" [fNum n]
--- ../../labs/lab4/tree.mli    2023-10-08 15:18:06.803585155 +0100
+++ tree.mli    2023-10-08 15:24:34.377061155 +0100
@@ -53,9 +53,14 @@
   | IfStmt of expr * stmt * stmt
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
-  | ForStmt of expr * expr * expr * stmt * def option ref
+  | ForStmt of expr * forelem list * stmt * def option ref
   | CaseStmt of expr * (expr * stmt) list * stmt

+and forelem =
+    OneElem of expr
+  | StepElem of expr * expr * expr
+  | WhileElem of expr * expr
+
 and expr =
   { e_guts: expr_guts;
     mutable e_type: ptype;
```

```
var i: integer;

begin
  for i := 1 to 5 do
    print_num(i);
    newline()
  end
end.

(*<<
1
2
3
4
5
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ for i := 1 to 5 do
        mov r0, #1
        ldr r1, =_i
        str r0, [r1]
        b .L7
.L4:
        ldr r4, =_i
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
.L7:
        ldr r0, =_i
        ldr r4, [r0]
        cmp r4, #5
        bgt .L1
@ print_num(i);
        mov r0, r4
        bl print_num
@ newline()
        bl newline
        b .L4
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _i, 4, 4

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

```
var i: integer;

begin
  for i := 1, 2, 5 step 2 until 10, i+3 while i * i < 400 do
    print_num(i);
    newline()
  end
end.

(*<<
1
2
5
7
9
14
17
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ for i := 1, 2, 5 step 2 until 10, i+3 while i * i < 400 do
        ldr r4, =.L4
        mov r0, #1
        ldr r1, =_i
        str r0, [r1]
        b .L2
.L4:
        ldr r4, =.L5
        mov r0, #2
        ldr r1, =_i
        str r0, [r1]
        b .L2
.L5:
        ldr r4, =.L6
        mov r0, #5
        ldr r1, =_i
        str r0, [r1]
        b .L9
.L6:
        ldr r5, =_i
        ldr r0, [r5]
        add r0, r0, #2
        str r0, [r5]
.L9:
        ldr r0, =_i
        ldr r0, [r0]
        cmp r0, #10
        ble .L2
        ldr r4, =.L10
.L10:
        ldr r5, =_i
        ldr r0, [r5]
        add r6, r0, #3
        str r6, [r5]
```

```
        mul r0, r6, r6
        cmp r0, #400
        bge .L1
.L2:
@ print_num(i);
        ldr r0, =_i
        ldr r0, [r0]
        bl print_num
@ newline()
        bl newline
        bx r4
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _i, 4, 4

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

```
(* Step in for loop is evaluated twice per iteration *)

var x, y, z: integer;

proc g(): integer;
begin
  y := y+1; return 1
end;

begin
  for x := 1 step g() until 10 do z := z+1 end;
  print_num(y); newline();
  print_num(z); newline()
end.

(*<<
21
10
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc g(): integer;
        .text
_g:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ y := y+1; return 1
        ldr r4, =_y
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
        mov r0, #1
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ for x := 1 step g() until 10 do z := z+1 end;
        mov r0, #1
        ldr r1, =_x
        str r0, [r1]
        b .L8
.L5:
        ldr r4, =_x
        bl _g
        ldr r1, [r4]
        add r0, r1, r0
        str r0, [r4]
.L8:
        bl _g
        cmp r0, #0
        ble .L6
        ldr r0, =_x
        ldr r0, [r0]
        cmp r0, #10
        ble .L3
```

```
        b .L4
.L6:
        ldr r0, =_x
        ldr r0, [r0]
        cmp r0, #10
        blt .L4
.L3:
        ldr r4, =_z
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
        b .L5
.L4:
@ print_num(y); newline();
        ldr r0, =_y
        ldr r0, [r0]
        bl print_num
        bl newline
@ print_num(z); newline()
        ldr r0, =_z
        ldr r0, [r0]
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _x, 4, 4

        .comm _y, 4, 4

        .comm _z, 4, 4

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

```
proc square(x: integer): integer;
begin return x*x end;

proc sqrt(x: integer): integer;
  var a, d: integer;
begin
  a := 0;
  for d := 1, 2*d while square(a+d) <= x, d div 2 while d >= 1 do
    if square(a+d) <= x then a := a+d end
  end;
  return a
end;

begin
  print_num(sqrt(200000000)); newline()
end.

(*<<
14142
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc square(x: integer): integer;
        .text
_square:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ begin return x*x end;
        ldr r4, [fp, #40]
        mul r0, r4, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc sqrt(x: integer): integer;
_sqrt:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ a := 0;
        mov r4, #0
@ for d := 1, 2*d while square(a+d) <= x, d div 2 while d >= 1 do
        ldr r6, =.L8
        mov r5, #1
        b .L3
.L8:
        ldr r6, =.L9
.L9:
        lsl r5, r5, #1
        add r0, r4, r5
        bl _square
        ldr r1, [fp, #40]
        cmp r0, r1
        ble .L3
        ldr r6, =.L11
.L11:
        mov r1, #2
```

```
        mov r0, r5
        bl int_div
        mov r5, r0
        cmp r5, #1
        blt .L4
.L3:
@ if square(a+d) <= x then a := a+d end
        add r0, r4, r5
        bl _square
        ldr r1, [fp, #40]
        cmp r0, r1
        bgt .L7
        add r4, r4, r5
.L7:
        bx r6
.L4:
@ return a
        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(sqrt(200000000)); newline()
        ldr r0, =200000000
        bl _sqrt
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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