Departmental Coversheet
Michaelmas term 2019 Mini-project
Paper title: Compilers Assignment MT2019
Candidate Number: 1033781
Your degree: Computer Science

Contents

1 Task 1 : Array Loops

To make the compiler understand a new type of statement (i.e. For-in Statement) means we need to alter more stages of compilation. We start from updating the abstract syntax tree, the lexer and the parser to recognise the key words and the structure of the new statement, then the semantic analyser that ensures correctness and finish with revising the code generator.

1.1 Abstract Syntax

Update **stmt_guts** to accommodate a new type of statement. Note that a For-in Structure should have, as arguments, the name of the variable used as iterator, the expression representing the array we are iterating over, the statements contained in the body of the loop and the slot provided for allocating a hidden variable, useful in the later stages.

```
and stmt_guts = ...
    | ForInStmt of name * expr * stmt * def option ref
```

Both **tree.ml** and **tree.mli** should be updated this way.

1.2 Lexer

The lexer (**lexer.mll**) only needs to be extended to recognise the keyword **in**.

```
let symtable = Util.make_hash 100 [ ... ("in", IN) ...]
```

1.3 Parser

A new token has to be added to the parser (**parser.mly**). Also, add For-in case to stmt definition.

```
%token IN
...
stmt1 :
    ...
    | FOR name IN expr DO stmts END       { ForInStmt ($2, $4, $6, ref None) }
```

1.4 Semantic Analysis

Our statement has multiple parameters: var (the iterator), arr (the array we iterate over), body (the body of the loop) and tmp (the hidden variable declared before).

We need to check that arr is actually an array. We do not need to check var, because it is right now just a name. Instead, we need to make a definition for it, create a new environment from the current one, and only then check the body with this new environment. Var will have the same type as the base type

of the array, and we assume that even if it is declared local/global before with the same name, it should run without errors. Moreover, inside the loop, by changing the value of var we want to change the value of the array. Ergo, the kind we use for var is *VParamDef*. Finally, we allocate memory for the hidden variable.

```
let rec check_stmt s env alloc = ...
    | ForInStmt (var, arr, body, tmp) ->
       let arrt = check_expr arr env and is_array t = match t.t_guts with
ArrayType (n, t1) -> true | _-> false in
       if not (is_array arrt) then failwith "not array";

       let d = make_def var.x_name VParamDef (base_type arrt) in
       let new_env = add_def d (new_block env) in
       begin
        alloc d;
        var.x_def <- Some d;
        check_stmt body new_env alloc;
       end;
       let t = make_def (intern "*pos*") VarDef integer in
       alloc t; tmp := Some t
```

1.5 Code Generation

We are going to use the hidden variable to store the position var is pointing to in the array. This way, it is more intuitive to check that we start from 0 and finish when we reach the size of the array. At each iteration of the loop, increase position by one and update var accordingly. The rest of the code is similar to the code in ForStmt.

```
let rec gen_stmt s = ...
      | ForInStmt (var, arr, body, tmp) ->
          (* Use previously allocated temp variable to store current position*)
          let pos = match !tmp with Some d -> d | _-> failwith "for in" and
              array_size t = match t.t_guts with ArrayType (n, t1) -> n
                                                 | _-> failwith "not_array" in

          let lab1 = label() and lab2 = label() and
              arr_size = array_size ( arr.e_type ) and
              f x = match x with Some y -> y | _-> failwith "no def" in

          <SEQ,
            <STOREW, <CONST 0>, address pos>,
            <LABEL lab1>,
            <STOREW, gen_addr arr, address (f var.x_def)>,
```

```
<JUMPC (Gt, lab2), <LOADW, address pos>,
                   <BINOP Minus, <CONST arr_size>, <CONST 1> > >,

<STOREW, <OFFSET, gen_addr arr, <BINOP Times, <LOADW, address pos>,
         <CONST (size_of ( base_type arr.e_type ))> > >,
         address (f var.x_def)>,

gen_stmt body,

<STOREW, <BINOP Plus, <LOADW, address pos>, <CONST 1> >,
         address pos>,

<JUMP lab1>,
<LABEL lab2>>
```

1.6 Test Cases

We want to explore some edge cases, and that the compiler is doing what it is intended.

- check that the compiler works correctly when we iterate over an array which is not of simple integers and that it can successfully write over this memory (this test using a record is already supplied)
- check that the compiler can correctly interpret nested for-in statements

```
var i: integer;
var v: array 5 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);

    for iter in v do
      if iter = 5 then
        newline()
      end
    end
  end
end.
```

- check that we can use the same variable name defined in the for-in statement in more for-ins on the same level

```
var i: integer;
var v: array 5 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end;

  for it in v do
    print_num(it);
    newline()
  end
end.
```

- check that the variable defined in the for-in statement cannot be used outside (basically check that the global environment is not influenced)

```
var i: integer;
var v: array 5 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end;

  print_num(it);
end.
```

- check that we can declare multiple variables in nested for-in statements with the same name and that they do not influence each other

```
var i, it: integer;
var v, w: array 2 of integer;

begin
  for i := 0 to 1 do
    v[i] := 5
```

```
  end;
  for i := 0 to 1 do
    w[i] := i+1
  end;

  it := 10;
  for it in v do
    print_num(it);
    newline();
    for it in w do
      print_num(it);
      newline()
    end;
  end;
  print_num(it);
  newline();
end.
```

- check that a for-in statement called for an empty array does nothing and runs without any errors

```
var i: integer;
var v: array 0 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end
end.
```

2 Task 2 : Commuting Code

As a general idea, we will check that the variables accessed in the first body do NOT intersect with the variables changed in the second body and that the variables changed in the first body do NOT intersect with the variables accessed in the second body. The method used does not consider procedures.

2.1 Abstract Syntax

Add the structure of Commute Statement in the **stmt_guts** in **tree.ml** and **tree.mli**.

```
and stmt_guts = ...
    | CommuteStmt of stmt * stmt
```

2.2 Lexer

The symbols '[' and ']' are already added, so we do not need to change anything here.

2.3 Parser

We need to alter stmt definition with CommuteStmt.

```
stmt1 : ...
    | SUB stmts COLON stmts BUS          { CommuteStmt ($2, $4) } ;
```

2.4 Semantic Analysis

Most of the functionality of our special statement is implemented in the file **check.ml**. We want to keep four data structures in which we update all the variables changed (in set1_changed and set2_changed) and accessed (in set1 an set2) in body1 and body2 of the statement. Because we want our code to have reasonable time complexity, we will use the Set implemented in the ocaml library. This allows additions in O(log n) and intersection of 2 sets in O(log n), where n is the number of elements in the set.

We start by declaring the four sets as global variables. We will store ident types, so we create our own custom set. Given that sets are immutable, we have use references. This way, when a function such as Set.add is called, we can store the reference to this newly created data structure.

```
module SetIdent = Set.Make(
  struct
    let compare = Pervasives.compare
    type t = ident
  end
  )
```

```
let set1_changed = ref SetIdent.empty
let set1 = ref SetIdent.empty
let set2_changed = ref SetIdent.empty
let set2 = ref SetIdent.empty
```

The initial function **check_stmt** has to be modified with the **CommuteStmt** option.

```
let rec check_stmt s env alloc = ...
    | CommuteStmt (body1, body2) ->
        check_stmt body1 env alloc;
        check_stmt body2 env alloc;

        check_stmt_var_changed body1 env alloc 1;
        check_stmt_var body1 env alloc 1;
        check_stmt_var_changed body2 env alloc 2;
        check_stmt_var body2 env alloc 2;

        if SetIdent.is_empty (SetIdent.inter (!set1) (!set2_changed)) = false then
        (*use failwith to make compiler halt*)
            failwith "Possibly incorrect";
        if SetIdent.is_empty (SetIdent.inter (!set1_changed) (!set2)) = false then
            failwith "Possibly incorrect";

        set1 := SetIdent.empty;
        set2 := SetIdent.empty;
        set1_changed := SetIdent.empty;
        set2_changed := SetIdent.empty
```

Note that we still have to check recursively body1 and body2 first. Afterwards, call the functions **check_stmt_var** and **check_stmt_var_changed** to update the sets for each body. To make implementation more intuitive, we use 2 integers, 1 or 2, to specify to which set we have to add.

To make computation of Commute Statements inside Commute Statements and/or sequences of Commute Statements, we make the sets empty at the end. The way this works is the following: if we have a sequence of Commute Statements, each time we enter another one, the sets are empty and **check_stmt_var** and **check_stmt_var_changed** will fill them accordingly; if we have a Commute Statement inside another one, when we enter the first, **check_stmt** will be called and will deal with the Commute Statement inside, where sets will be empty at exit; now **check_stmt_var** and **check_stmt_var_changed** will add all inner changed variables to the sets, even if they are in another Commute Statement.

Also, we want our program to halt when we encounter our first "Possibly incorrect". Otherwise, if we do not have any Commute Statement or all of them are correct, we want it to halt at the end of **check.ml**.

```
let annotate (Prog (Block (globals, ss, fsize, nregv), glodefs)) =
    ...
    failwith "Correct"
```

The function **check_stmt_var** is implemented similarly to **check_stmt**, but simplified, as it only needs to check which variables are called.

```
let rec check_stmt_var s env alloc cnt =
    match s.s_guts with
      Skip -> ()

    | Seq ss ->
        List.iter (fun s1 -> check_stmt_var s1 env alloc cnt) ss

    | Assign (lhs, rhs) ->
        check_expr_var lhs env cnt;
        check_expr_var rhs env cnt

    | IfStmt (cond, thenpt, elsept) ->
        check_expr_var cond env cnt;
        check_stmt_var thenpt env alloc cnt;
        check_stmt_var elsept env alloc cnt

    | WhileStmt (cond, body) ->
        check_expr_var cond env cnt;
        check_stmt_var body env alloc cnt

    | RepeatStmt (body, test) ->
        check_expr_var test env cnt;
        check_stmt_var body env alloc cnt

    | ForStmt (var, lo, hi, body, upb) ->
        check_expr_var lo env cnt;
        check_expr_var hi env cnt;
        check_expr_var var env cnt;
        check_stmt_var body env alloc cnt

    | CaseStmt (sel, arms, deflt) ->
        check_expr_var sel env cnt;
        let check_arm_var (lab, body) = check_stmt_var body env alloc cnt in
        List.iter check_arm_var arms;
        check_stmt_var deflt env alloc cnt

    | CommuteStmt (body1, body2) ->
      check_stmt_var body1 env alloc cnt;
      check_stmt_var body2 env alloc cnt
```

```
    | _ -> ()
```

The function **check_stmt_var_changed** is almost the same as **check_stmt_var**, but only checks variables that are changed.

After analyzing the syntax of the compiled language, we deduce that the only way a statement can change the value of a variable is through an assign operation or within a for statement. (We assume that the code from task 1 is not used).

```
let rec check_stmt_var_changed s env alloc cnt =
  match s.s_guts with
    Skip -> ()

  | Seq ss ->
      List.iter (fun s1 -> check_stmt_var_changed s1 env alloc cnt) ss

(*We do not need to check rhs because the variable changed is only the one on the left *)
  | Assign (lhs, rhs) ->
      check_expr_var_changed lhs env cnt

  | IfStmt (cond, thenpt, elsept) ->
      check_stmt_var_changed thenpt env alloc cnt;
      check_stmt_var_changed elsept env alloc cnt

  | WhileStmt (cond, body) ->
      check_stmt_var_changed body env alloc cnt

  | RepeatStmt (body, test) ->
      check_stmt_var_changed body env alloc cnt

  | ForStmt (var, lo, hi, body, upb) ->
      check_expr_var_changed var env cnt;
      check_stmt_var_changed body env alloc cnt

  | CaseStmt (sel, arms, deflt) ->
      let check_arm_var (lab, body) = check_stmt_var_changed body env alloc cnt in
      List.iter check_arm_var arms;
      check_stmt_var_changed deflt env alloc cnt

  | CommuteStmt (body1, body2) ->
    check_stmt_var_changed body1 env alloc cnt;
    check_stmt_var_changed body2 env alloc cnt

  | _ -> ()
```

Both our functions call further **check_expr_var** and **check_expr_var_changed**

which actually add the variables in the sets accordingly. By looking at expression type in **tree.ml**, we see that for checking modified variables, it is enough to only check if the expression is a variable, a subscript or a select. These are the only ones that can appear in the left hand side of the assignment or in a for.

```
let rec check_expr_var e env cnt =
    match e.e_guts with
      Variable x ->
        if cnt = 1 then
          set1 := SetIdent.add (x.x_name) (!set1)
        else
          set2 := SetIdent.add (x.x_name) (!set2)

    | Sub (a, b) ->
      check_expr_var a env cnt;
      check_expr_var b env cnt

    | Select (a, b) ->
      check_expr_var a env cnt

    | Deref (x) ->
      check_expr_var x env cnt

    | Monop (sym, x) ->
      check_expr_var x env cnt

    | Binop (sym, x, y) ->
      check_expr_var x env cnt;
      check_expr_var y env cnt

    | _-> ()

let rec check_expr_var_changed e env cnt =
  match e.e_guts with
    Variable x ->
      if cnt = 1 then
        set1_changed := SetIdent.add (x.x_name) (!set1_changed)
      else
        set2_changed := SetIdent.add (x.x_name) (!set2_changed)

  | Sub (a, b) ->
    check_expr_var_changed a env cnt

  | Select (a, b) ->
    check_expr_var_changed a env cnt
```

```
  | _-> ()
```

2.5 Code Generation

Given that our code should halt every time it encounters a Commute Statement, Code Generation is not used, but we still need to augment our **gen_stmt** function, otherwise errors will appear.

```
let rec gen_stmt s = ...
    | CommuteStmt (body1, body2) ->
          <SEQ,
            gen_stmt body1,
            gen_stmt body2>
```

2.6 Test Cases

We will explore more test cases, with more focus on edge cases, to ensure the code has no bugs.

- check that the code works correctly on some normal tests
- check that the code sees when we call a variable in a for statement

```
var x, y, i: integer;
begin
[ x:=1; x:=1:
  for x := 1 to 2 do
    i := i+1
  end
 ];
print_num(x);
end.
```

```
var x, y,z,i: integer;
begin
[ x := 3 :
  for i := 1 to x do
    y := 3
  end
 ];
print_num(x);
end.
```

  Both should return "Possibly incorrect"

- check that if variables are called in both bodies but not changed, it returns "Correct"

```
var x, y, z : integer;
begin
  x := 3;
  y := 4;
  z := 5;
  [ y := x : z := x];
  print_num(y);
end.
```

- check that if statement and while statement are interpreted as intended

```
var x, y, z, t, i, j : integer;
begin
  x:=1;
  y:=1;
  z:=1;
 [if x=3 then z:=2 else t:=3; end :
  x:=15;
  for i := 1 to 10 do
    for j := 1 to 12 do
      x:=x+1;
    end;
  end;
];
  x:=1;
  y:=1;
  z:=1;
  t:=1;
end.
```

and

```
var a, b, c, d : integer;
begin
  a := 1;
  b := 1;
  while (b <> 2) do
    b := b+1
  end;
  [a := 3 :
  b := 4;
  while (b > 2) do
  b := b +1;
  b := b -2
  end;
  ]
end.
```

- check that a sequence of correct Commute Statements is interpreted as intended

```
var x, y, z, t : integer;
begin
[x:=1; y:=1 :
 z:=1; t:=1];

[x:=1; z:=1 :
  t:=1; y:=1]
end.
```

- check that two correct nested Commute Statements are interpreted as intended

```
var x, y, z, t : integer;
begin
[x:=1; [x:=2 : z:=y] : t:=y ]
end.
```

- check that two nested Commute Statements (one correct, one not) give out "Possibly incorrect"

```
var x, y, z, t : integer;
begin
[x:=1; [x:=2 : y:=3] : t:=x ]
end.
```

- check that records are interpreted correctly when different

```
type point = record x,y :integer end;
var a, b: point;

begin
[a.x := 3; a.y := 5 :
 b.y := 4; b.x := 4]
end.
```

- see that the code does not make a distinction between parts of the same record - Possibly incorrect (note that the same happens with arrays and different entries)

```
type point = record x,y :integer end;
var a, b: point;

begin
[a.x := 3 :
 a.y := 4]
end.
```

- check that case statements work properly

```
var a, b, c : integer;
begin
a := 2;
[ case a of   1 : b:=3
            | 2 : c:=3
            end
: a:=3]
end.
```

- check that a variable changed in one body and called but not changed in the other is detected

```
var x, y, z : integer;
begin
[x := y + z : y:= 3]
end.
```

3 Diff For Task 1

```
diff -r 5e3fa9d71839 lab4/check.ml
--- a/lab4/check.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/check.ml        Thu Jan 30 03:56:01 2020 +0000
@@ -7,6 +7,7 @@
 open Print
 open Lexer
 open Mach
+open Set

 (* EXPRESSIONS *)

@@ -37,15 +38,15 @@
 (* |lookup_def| -- find definition of a name, give error if none *)
 let lookup_def x env =
   err_line := x.x_line;
-  try
-    let d = lookup x.x_name env in
+  try
+    let d = lookup x.x_name env in
     x.x_def <- Some d; d
-  with Not_found ->
+  with Not_found ->
     sem_error "$ is not declared" [fId x.x_name]

 (* |add_def| -- add definition to envmt, give error if already declared *)
 let add_def d env =
-  try define d env with
+  try define d env with
     Exit -> sem_error "$ is already declared" [fId d.d_tag]

 (* |check_monop| -- check application of unary operator *)
@@ -78,19 +79,19 @@
 (* |try_monop| -- propagate constant through unary operation *)
 let try_monop w =
   function
-      Some x -> Some (do_monop w x)
+      Some x -> Some (do_monop w x)
     | None -> None

 (* |try_binop| -- propagate constant through unary operation *)
 let try_binop w v1 v2 =
-  match (v1, v2) with
+  match (v1, v2) with
       (Some x1, Some x2) -> Some (do_binop w x1 x2)
```

```
     | _ -> None

 (* |has_value| -- check if object is suitable for use in expressions *)
-let has_value d =
+let has_value d =
   match d.d_kind with
-      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
+      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -99,10 +100,10 @@
       Variable x ->
         let d = get_def x in
         begin
-          match d.d_kind with
+          match d.d_kind with
               VarDef | VParamDef | CParamDef ->
                 d.d_mem <- d.d_mem || addressible
-            | _ ->
+            | _ ->
                 sem_error "$ is not a variable" [fId x.x_name]
         end
     | Sub (a, i) -> check_var a addressible
@@ -118,15 +119,15 @@
 (* |expr_type| -- return type of expression *)
 and expr_type e env =
   match e.e_guts with
-      Variable x ->
-        let d = lookup_def x env in
+      Variable x ->
+        let d = lookup_def x env in
+        if not (has_value d) then
+          sem_error "$ is not a variable" [fId x.x_name];
         if d.d_level < !level then d.d_mem <- true;
         begin
-          match d.d_kind with
+          match d.d_kind with
               ConstDef v ->
-                e.e_value <- Some v
+                e.e_value <- Some v
             | _ -> ()
         end;
         d.d_type
@@ -162,17 +163,17 @@
     | Constant (n, t) -> e.e_value <- Some n; t
```

```
     | String (lab, n) -> row n character
     | Nil -> e.e_value <- Some 0; addrtype
-    | FuncCall (p, args) ->
+    | FuncCall (p, args) ->
         let v = ref None in
         let t1 = check_funcall p args env v in
         if same_type t1 voidtype then
           sem_error "$ does not return a result" [fId p.x_name];
         e.e_value <- !v; t1
-    | Monop (w, e1) ->
+    | Monop (w, e1) ->
         let t = check_monop w (check_expr e1 env) in
         e.e_value <- try_monop w e1.e_value;
         t
-    | Binop (w, e1, e2) ->
+    | Binop (w, e1, e2) ->
         let t = check_binop w (check_expr e1 env) (check_expr e2 env) in
         e.e_value <- try_binop w e1.e_value e2.e_value;
         t
@@ -185,7 +186,7 @@
         check_libcall q args env v; d.d_type
     | ProcDef | PParamDef ->
         let p = get_proc d.d_type in
-        check_args p.p_fparams args env;
+        check_args p.p_fparams args env;
         p.p_result
     | _ -> sem_error "$ is not a procedure" [fId f.x_name]

@@ -202,22 +203,22 @@
         let t1 = check_expr arg env in
         if not (same_type formal.d_type t1) then
           sem_error "argument has wrong type" [];
-        if formal.d_kind = VParamDef then
+        if formal.d_kind = VParamDef then
           check_var arg true
     | PParamDef ->
         let pf = get_proc formal.d_type in
-        let x = (match arg.e_guts with Variable x -> x
+        let x = (match arg.e_guts with Variable x -> x
           | _ -> sem_error "procedure argument must be a proc name" []) in
         let actual = lookup_def x env in
         begin
-          match actual.d_kind with
+          match actual.d_kind with
             ProcDef | PParamDef  ->
               let pa = get_proc actual.d_type in
```

```
                if not (match_args pf.p_fparams pa.p_fparams) then
                  sem_error "argument lists don't match" [];
                if not (same_type pf.p_result pa.p_result) then
                  sem_error "result types don't match" []
-           | _ ->
+           | _ ->
                sem_error "argument $ is not a procedure" [fId x.x_name]
        end
    | _ -> failwith "bad formal"
@@ -235,7 +236,7 @@
        sem_error "argument of $ has wrong type" [fLibId q.q_id] in
    List.iter2 check q.q_argtypes args
  end;
-  match q.q_id with
+  match q.q_id with
      ChrFun ->
        let e1 = List.hd args in
        v := e1.e_value
@@ -253,7 +254,7 @@
        check_var (List.hd args) true
    | NewProc ->
        let t1 = check_expr (List.hd args) env in
-       if not (is_pointer t1) then
+       if not (is_pointer t1) then
          sem_error "parameter of new must be a pointer" [];
        check_var (List.hd args) false
    | ArgvProc ->
@@ -284,14 +285,23 @@

 (* check_dupcases -- check for duplicate case labels *)
 let check_dupcases vs =
-  let rec chk =
+  let rec chk =
    function
        [] | [_] -> ()
-     | x :: (y :: ys as rest) ->
+     | x :: (y :: ys as rest) ->
          if x = y then sem_error "duplicate case label" [];
          chk rest in
  chk (List.sort compare vs)

+(* Global sets for Commute checking -> they store the variables used in body1 and body2 of
+module SetIdent = Set.Make(
+  struct
+    let compare = Pervasives.compare
+    type t = ident
```

```
+  end
+  )
+
+
 (* |check_stmt| -- check and annotate a statement *)
 let rec check_stmt s env alloc =
   err_line := s.s_line;
@@ -332,7 +342,7 @@
     | IfStmt (cond, thenpt, elsept) ->
         let ct = check_expr cond env in
         if not (same_type ct boolean) then
-          sem_error "test in if statement must be a boolean" [];
+          sem_error "test in if statement must be a boolean" [];
         check_stmt thenpt env alloc;
         check_stmt elsept env alloc

@@ -363,6 +373,22 @@
         let d = make_def (intern "*upb*") VarDef integer in
         alloc d; upb := Some d

+    | ForInStmt (var, arr, body, tmp) ->
+       let arrt = check_expr arr env and is_array t = match t.t_guts with ArrayType (n, t1)
+       if not (is_array arrt) then failwith "not array";
+
+       let d = make_def var.x_name VParamDef (base_type arrt) in
+       let new_env = add_def d (new_block env) in
+       begin
+        alloc d;
+        var.x_def <- Some d;
+        check_stmt body new_env alloc;
+       end;
+       let t = make_def (intern "*pos*") VarDef integer in
+       alloc t; tmp := Some t
+
+
+
     | CaseStmt (sel, arms, deflt) ->
         let st = check_expr sel env in
         if not (scalar st) then
@@ -378,7 +404,6 @@
         check_dupcases vs;
         check_stmt deflt env alloc

-
 (* TYPES AND DECLARATIONS *)
```

```
 (* |lookup_typename| -- find a named type in the environment *)
@@ -403,7 +428,7 @@

 (* local_alloc -- allocate locals downward in memory *)
 let local_alloc size nreg d =
-  if !regvars && not d.d_mem && scalar (d.d_type)
+  if !regvars && not d.d_mem && scalar (d.d_type)
       && !nreg < Mach.nregvars then begin
     d.d_addr <- Register !nreg; incr nreg
   end
@@ -440,7 +465,7 @@
   List.iter h ds

 (* |check_typexpr| -- check a type expression, returning the ptype *)
-let rec check_typexpr te env =
+let rec check_typexpr te env =
   match te with
       TypeName x ->
         let d = lookup_typename x env in
@@ -461,7 +486,7 @@
         let r = { r_size = !size; r_align = max_align } in
         mk_type (RecordType defs) r
     | Pointer te ->
-        let t =
+        let t =
           match te with
               TypeName x ->
                 let d = lookup_typename x env in
@@ -475,7 +500,7 @@
         mk_type (PointerType t) addr_rep

 (* |check_decl| -- check a declaration and add it to the environment *)
-and check_decl d env =
+and check_decl d env =
   (* All types of declaration are mixed together in the AST *)
   match d with
       ConstDecl (x, e) ->
@@ -492,12 +517,12 @@
         end
     | VarDecl (kind, xs, te) ->
         let t = check_typexpr te env in
-        let def x env =
+        let def x env =
           let d = make_def x kind t in
           add_def d env in
         Util.accum def xs env
```

```
     | TypeDecl tds ->
-        let tds' =
+        let tds' =
           List.map (function (x, te) -> (x, te, ref voidtype)) tds in
         let add_hole (x, te, h) env1 =
           add_def (make_def x (HoleDef h) voidtype) env1 in
@@ -505,7 +530,7 @@
         let redefine (x, te, h) env1 =
           let t = check_typexpr te env1 in
           h := t; replace (make_def x TypeDef t) env1 in
-        Util.accum redefine tds' env'
+        Util.accum redefine tds' env'
     | ProcDecl (Heading (x, _, _) as heading, body) ->
         let t = check_heading env heading in
         let d = make_def x.x_name ProcDef t in
@@ -515,7 +540,7 @@
         let t = check_heading env heading in
         let d = make_def x.x_name PParamDef t in
         add_def d env
-
+
 (* |check_heading| -- process a procedure heading into a procedure type *)
 and check_heading env (Heading (x, fparams, result)) =
   err_line := x.x_line;
@@ -554,7 +579,7 @@
   align max_align fsize

 (* |check_bodies| -- check bodies of procedure declarations *)
-and check_bodies env ds =
+and check_bodies env ds =
   let check =
     function
         ProcDecl (Heading(x, _, _), body) ->
@@ -570,19 +595,19 @@

 (* INITIAL ENVIRONMENT *)

-let defn (x, k, t) env =
+let defn (x, k, t) env =
   let d = { d_tag = intern x; d_kind = k; d_type = t;
             d_level = 0; d_mem = false; d_addr = Nowhere } in
   define d env

-let libproc i n ts =
+let libproc i n ts =
   LibDef { q_id = i; q_nargs = n; q_argtypes = ts }
```

```
 let operator op ts =
   libproc (Operator op) (List.length ts) ts

 (* |init_env| -- environment for whole program *)
-let init_env =
+let init_env =
   Util.accum defn
     [ ("integer", TypeDef, integer);
       ("char", TypeDef, character);
diff -r 5e3fa9d71839 lab4/dict.ml
--- a/lab4/dict.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/dict.ml        Thu Jan 30 03:56:01 2020 +0000
@@ -43,14 +43,14 @@
     | Nowhere -> fStr "*nowhere*"

 (* |libid| -- type of picoPascal library procedures *)
-type libid = ChrFun | OrdFun | PrintNum | PrintChar | PrintString
+type libid = ChrFun | OrdFun | PrintNum | PrintChar | PrintString
   | NewLine | ReadChar | ExitProc | NewProc | ArgcFun | ArgvProc
   | OpenIn | CloseIn | Operator of Optree.op

 (* |lib_name| -- name of a library procedure *)
-let lib_name x =
+let lib_name x =
   match x with
-      PrintNum -> "print_num" | PrintChar -> "print_char"
+      PrintNum -> "print_num" | PrintChar -> "print_char"
     | PrintString -> "print_string" | NewLine -> "newline"
     | ReadChar -> "read_char" | ChrFun -> "chr" | OrdFun -> "ord"
     | ExitProc -> "exit" | NewProc -> "new"
@@ -73,7 +73,7 @@
 *)

 (* |def_kind| -- kinds of definition *)
-type def_kind =
+type def_kind =
     ConstDef of int             (* Constant (value) *)
   | StringDef                   (* String *)
   | TypeDef                     (* Type *)
@@ -88,7 +88,7 @@
   | DummyDef                    (* Dummy *)

 (* |def| -- definitions in environment *)
-and def =
+and def =
```

```
   { d_tag: ident;                (* Name *)
     d_kind: def_kind;            (* Kind of object *)
     d_type: ptype;               (* Type *)
@@ -99,7 +99,7 @@
 and basic_type = VoidType | IntType | CharType | BoolType | AddrType

 (* |ptype| -- picoPascal types *)
-and ptype =
+and ptype =
   { t_id: int;                   (* Unique identifier *)
     t_guts: type_guts;           (* Shape of the type *)
     t_rep: Mach.metrics }
@@ -123,10 +123,10 @@
     q_nargs: int;
     q_argtypes: ptype list }

-module IdMap =
-  Map.Make(struct
-    type t = ident
-    let compare = compare
+module IdMap =
+  Map.Make(struct
+    type t = ident
+    let compare = compare
   end)

 type environment = Env of (def list * def IdMap.t)
@@ -144,13 +144,13 @@
     function
         [] -> raise Not_found
-      | d::ds ->
+      | d::ds ->
           if x = d.d_tag then d else search ds in
   search ds

 let can f x = try f x; true with Not_found -> false

-let define d (Env (b, m)) =
+let define d (Env (b, m)) =
   if can (find_def d.d_tag) b then raise Exit;
   Env (d::b, add_def d m)

@@ -158,7 +158,7 @@
   let rec repl =
     function
```

```
          [] -> failwith "replace"
-       | d'::ds ->
+       | d'::ds ->
            if d.d_tag = d'.d_tag then d::ds else d' :: repl ds in
    Env (repl b, add_def d m)

@@ -179,7 +179,7 @@
 let addrtype =  mk_type (BasicType AddrType) addr_rep

 let row n t =
-  let r = t.t_rep in
+  let r = t.t_rep in
   mk_type (ArrayType (n, t)) { r_size = n * r.r_size; r_align = r.r_align }

 let discrete t =
@@ -209,15 +209,20 @@
     | ArrayType (n, t1) -> t1
     | _ -> failwith "base_type"

+let array_size t =
+  match t.t_guts with
+      ArrayType (n, t1) -> n
+    | _-> failwith "not_array"
+
 let get_proc t =
   match t.t_guts with
       ProcType p -> p
     | _ -> failwith "get_proc"

-let rec same_type t1 t2 =
+let rec same_type t1 t2 =
   match (t1.t_guts, t2.t_guts) with
       (ProcType p1, ProcType p2) ->
-        match_args p1.p_fparams p2.p_fparams
+        match_args p1.p_fparams p2.p_fparams
         && same_type p1.p_result p2.p_result
     | (ArrayType (n1, u1), ArrayType(n2, u2)) ->
         n1 = n2 && same_type u1 u2
@@ -225,7 +230,7 @@
     | (BasicType x, PointerType _) -> x = AddrType
     | (_, _) -> t1.t_id = t2.t_id

-and match_args fp1 fp2 =
+and match_args fp1 fp2 =
   match (fp1, fp2) with
       ([], []) -> true
```

```
     | (f1::fp1', f2::fp2') ->
diff -r 5e3fa9d71839 lab4/lexer.mll
--- a/lab4/lexer.mll        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/lexer.mll        Thu Jan 30 03:56:01 2020 +0000
@@ -11,10 +11,10 @@

 let lineno = ref 1                      (* Current line in input file *)

-let symtable =
+let symtable =
   Util.make_hash 100
     [ ("array", ARRAY); ("begin", BEGIN);
-      ("const", CONST); ("do", DO); ("if", IF ); ("else", ELSE);
+      ("const", CONST); ("do", DO); ("if", IF ); ("else", ELSE);
       ("end", END); ("of", OF); ("proc", PROC); ("record", RECORD);
       ("return", RETURN); ("then", THEN); ("to", TO);
       ("type", TYPE); ("var", VAR); ("while", WHILE);
@@ -22,7 +22,7 @@
       ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
       ("elsif", ELSIF); ("case", CASE);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
-      ("not", NOT); ("mod", MULOP Mod) ]
+      ("not", NOT); ("mod", MULOP Mod); ("in", IN) ]

 let lookup s =
   try Hashtbl.find symtable s with
@@ -68,14 +68,14 @@
 let notq = [^'\'']
 let notqq = [^'"']

-rule token =
+rule token =
   parse
       letter (letter | digit | '_')* as s
                       { lookup s }
     | digit+ as s     { NUMBER (int_of_string s) }
     | q (notq as c) q { CHAR c }
     | q q q q         { CHAR '\'' }
-    | qq ((notqq | qq qq)* as s) qq
+    | qq ((notqq | qq qq)* as s) qq
                       { get_string s }
     | ";"             { SEMI }
     | "."             { DOT }
@@ -102,14 +102,13 @@
     | "\r"            { token lexbuf }
     | "\n"            { next_line lexbuf; token lexbuf }
```

```
     | _                 { BADTOK }
-    | eof               { err_message "unexpected end of file" [] !lineno;
+    | eof               { err_message "unexpected end of file" [] !lineno;
                           exit 1 }

-and comment =
+and comment =
   parse
       "*)"              { () }
     | "\n"              { next_line lexbuf; comment lexbuf }
     | _                 { comment lexbuf }
-    | eof               { err_message "end of file in comment" [] !lineno;
+    | eof               { err_message "end of file in comment" [] !lineno;
                           exit 1 }
-
diff -r 5e3fa9d71839 lab4/parser.mly
--- a/lab4/parser.mly        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/parser.mly        Thu Jan 30 03:56:01 2020 +0000
@@ -9,7 +9,7 @@

 %token <Dict.ident>     IDENT
 %token <Optree.op>      MULOP ADDOP RELOP
-%token <int>            NUMBER
+%token <int>            NUMBER
 %token <char>           CHAR
 %token <Optree.symbol * int> STRING

@@ -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 IN

 %type <Tree.program>    program
 %start                  program
@@ -33,17 +33,17 @@

 %%

-program :
+program :
     block DOT                          { Prog ($1, ref []) } ;

-block :
+block :
```

```
     decl_list BEGIN stmts END          { makeBlock ($1, $3) } ;

-decl_list :
+decl_list :
     /* empty */                        { [] }
   | decl decl_list                     { $1 @ $2 } ;

-decl :
+decl :
     CONST const_decls                  { $2 }
   | VAR var_decls                      { $2 }
   | proc_decl                          { [$1] }
@@ -60,7 +60,7 @@
     type_decl                          { [$1] }
   | type_decl type_decls               { $1 :: $2 } ;

-type_decl :
+type_decl :
     IDENT EQUAL typexpr SEMI           { ($1, $3) } ;

 var_decls :
@@ -73,18 +73,18 @@
 proc_decl :
     proc_heading SEMI block SEMI       { ProcDecl ($1, $3) } ;

-proc_heading :
+proc_heading :
     PROC name params return_type       { Heading ($2, $3, $4) } ;

 params :
     LPAR RPAR                          { [] }
   | LPAR formal_decls RPAR             { $2 } ;

-formal_decls :
+formal_decls :
     formal_decl                        { [$1] }
   | formal_decl SEMI formal_decls      { $1 :: $3 } ;

-formal_decl :
+formal_decl :
     ident_list COLON typexpr           { VarDecl (CParamDef, $1, $3) }
   | VAR ident_list COLON typexpr       { VarDecl (VParamDef, $2, $4) }
   | proc_heading                       { PParamDecl $1 } ;
@@ -93,14 +93,14 @@
     /* empty */                        { None }
   | COLON typexpr                      { Some $2 } ;
```

```
-stmts :
+stmts :
     stmt_list                          { seq $1 } ;

 stmt_list :
     stmt                               { [$1] }
   | stmt SEMI stmt_list                { $1 :: $3 } ;

-stmt :
+stmt :
     line stmt1                         { makeStmt ($2, $1) }
   | /* A trick to force the right line number */
     IMPOSSIBLE                         { failwith "impossible" } ;
@@ -114,12 +114,13 @@
   | name actuals                       { ProcCall ($1, $2) }
   | RETURN expr_opt                    { Return $2 }
   | IF expr THEN stmts elses END       { IfStmt ($2, $4, $5) }
-  | WHILE expr DO stmts END            { WhileStmt ($2, $4) }
+  | WHILE expr DO stmts END            { WhileStmt ($2, $4) }
   | REPEAT stmts UNTIL expr            { RepeatStmt ($2, $4) }
-  | FOR name ASSIGN expr TO expr DO stmts END
+  | FOR name ASSIGN expr TO expr DO stmts END
                                       { let v = makeExpr (Variable $2) in
-                                        ForStmt (v, $4, $6, $8, ref None) }
-  | CASE expr OF arms else_part END    { CaseStmt ($2, $4, $5) } ;
+                                        ForStmt (v, $4, $6, $8, ref None) }
+  | CASE expr OF arms else_part END    { CaseStmt ($2, $4, $5) }
+  | FOR name IN expr DO stmts END      { ForInStmt ($2, $4, $6, ref None) }

 elses :
     /* empty */                        { makeStmt (Skip, 0) }
@@ -137,11 +138,11 @@
     /* empty */                        { makeStmt (Skip, 0) }
   | ELSE stmts                         { $2 } ;

-ident_list :
+ident_list :
     IDENT                              { [$1] }
   | IDENT COMMA ident_list             { $1 :: $3 } ;

-expr_opt :
+expr_opt :
     /* empty */                        { None }
   | expr                               { Some $1 } ;
```

```
@@ -171,21 +172,21 @@
   | MINUS factor                          { makeExpr (Monop (Uminus, $2)) }
   | LPAR expr RPAR                        { $2 } ;

-actuals :
+actuals :
     LPAR RPAR                             { [] }
   | LPAR expr_list RPAR                   { $2 } ;

-expr_list :
+expr_list :
     expr                                  { [$1] }
   | expr COMMA expr_list                  { $1 :: $3 } ;

-variable :
+variable :
     name                                  { makeExpr (Variable $1) }
   | variable SUB expr BUS                 { makeExpr (Sub ($1, $3)) }
   | variable DOT name                     { makeExpr (Select ($1, $3)) }
   | variable ARROW                        { makeExpr (Deref $1) } ;

-typexpr :
+typexpr :
     name                                  { TypeName $1 }
   | ARRAY expr OF typexpr                 { Array ($2, $4) }
   | RECORD fields END                     { Record $2 }
@@ -195,12 +196,12 @@
     field_decl opt_semi                   { [$1] }
   | field_decl SEMI fields                { $1 :: $3 } ;

-field_decl :
+field_decl :
     ident_list COLON typexpr              { VarDecl (FieldDef, $1, $3) } ;

 opt_semi :
     SEMI                                  { () }
   | /* empty */                           { () } ;

-name :
+name :
     IDENT                                 { makeName ($1, !Lexer.lineno) } ;
diff -r 5e3fa9d71839 lab4/tgen.ml
--- a/lab4/tgen.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/tgen.ml        Thu Jan 30 03:56:01 2020 +0000
@@ -7,6 +7,7 @@
 open Optree
```

```
 open Lexer
 open Print
+open Check

 let boundchk = ref false
 let optlevel = ref 0
@@ -51,11 +52,11 @@
   match d.d_addr with
       Global g ->
         <GLOBAL g>
-    | Local off ->
+    | Local off ->
         <OFFSET, schain (!level - d.d_level), <CONST off>>
     | Register i ->
         <REGVAR i>
-    | Nowhere ->
+    | Nowhere ->
         failwith (sprintf "address $" [fId d.d_tag])

 (* |gen_closure| -- two trees for a (code, envt) pair *)
@@ -84,7 +85,7 @@
   libcall "memcpy" [dst; src; <CONST n>] voidtype

 (* |gen_addr| -- code for the address of a variable *)
-let rec gen_addr v =
+let rec gen_addr v =
   match v.e_guts with
       Variable x ->
         let d = get_def x in
@@ -95,19 +96,19 @@
             | VParamDef ->
                 <LOADW, address d>
             | CParamDef ->
-                if scalar d.d_type || is_pointer d.d_type then
+                if scalar d.d_type || is_pointer d.d_type then
                   address d
                 else
                   <LOADW, address d>
             | StringDef ->
                 address d
-            | _ ->
+            | _ ->
                 failwith "load_addr"
         end
     | Sub (a, i) ->
         let bound_check t =
```

```
          if not !boundchk then t else <BOUND, t, <CONST (bound a.e_type)>> in
-       <OFFSET,
+       <OFFSET,
          gen_addr a,
          <BINOP Times, bound_check (gen_expr i), <CONST (size_of v.e_type)>>>
     | Select (r, x) ->
@@ -123,9 +124,9 @@
 (* |gen_expr| -- tree for the value of an expression *)
 and gen_expr e =
   match e.e_value with
-      Some v ->
+      Some v ->
         <CONST v>
-    | None ->
+    | None ->
         begin
           match e.e_guts with
               Variable _ | Sub _ | Select _ | Deref _ ->
@@ -139,7 +140,7 @@
                 libcall "int_mod" [gen_expr e1; gen_expr e2] integer
             | Binop (w, e1, e2) ->
                 <BINOP w, gen_expr e1, gen_expr e2>
-            | FuncCall (p, args) ->
+            | FuncCall (p, args) ->
                 gen_call p args
             | _ -> failwith "gen_expr"
         end
@@ -157,21 +158,21 @@
         <CALL p.p_pcount, @(fn :: <STATLINK, sl> :: numargs 0 args)>

 (* |gen_arg| -- generate code for a procedure argument *)
-and gen_arg f a =
+and gen_arg f a =
   match f.d_kind with
       CParamDef ->
-        if scalar f.d_type || is_pointer f.d_type then
+        if scalar f.d_type || is_pointer f.d_type then
           [gen_expr a]
-        else
+        else
           [gen_addr a]
     | VParamDef ->
         [gen_addr a]
     | PParamDef ->
         begin
-          match a.e_guts with
```

```
-              Variable x ->
+          match a.e_guts with
+              Variable x ->
                let (fn, sl) = gen_closure (get_def x) in [fn; sl]
-            | _ ->
+            | _ ->
                failwith "bad funarg"
         end
     | _ -> failwith "bad arg"
@@ -243,14 +244,14 @@
     let rec tab u qs =
       match qs with
          [] -> []
-       | (v, l) :: rs ->
+       | (v, l) :: rs ->
           if u = v then l :: tab (v+1) rs else deflab :: tab (u+1) qs in
     <JCASE (tab lobound table, deflab),
      <BINOP Minus, sel, <CONST lobound>>>
   end

 (* |gen_stmt| -- generate code for a statement *)
-let rec gen_stmt s =
+let rec gen_stmt s =
   let code =
     match s.s_guts with
        Skip -> <NOP>
@@ -302,7 +303,7 @@
          let l1 = label () and l2 = label () in
          <SEQ,
           <LABEL l1>,
-          gen_stmt body,
+          gen_stmt body,
           gen_cond test l2 l1,
           <LABEL l2>>

@@ -321,6 +322,21 @@
          <JUMP l1>,
          <LABEL l2>>

+      | ForInStmt (var, arr, body, tmp) ->
+          (* Use previously allocated temp variable to store current position*)
+          let pos = match !tmp with Some d -> d | _-> failwith "for in" and array_size t =
+          let lab1 = label() and lab2 = label() and arr_size = array_size ( arr.e_type ) ar
+          <SEQ,
+           <STOREW, <CONST 0>, address pos>,
+           <LABEL lab1>,
```

```
+            <STOREW, gen_addr arr, address (f var.x_def)>,
+            <JUMPC (Gt, lab2), <LOADW, address pos>, <BINOP Minus, <CONST arr_size>, <CONST
+            <STOREW, <OFFSET, gen_addr arr, <BINOP Times, <LOADW, address pos>, <CONST (siz
+            gen_stmt body,
+            <STOREW, <BINOP Plus, <LOADW, address pos>, <CONST 1> >, address pos>,
+            <JUMP lab1>,
+            <LABEL lab2>>
+
       | CaseStmt (sel, arms, deflt) ->
           (* Use one jump table, and hope it is reasonably compact *)
           let deflab = label () and donelab = label () in
@@ -348,8 +364,8 @@
     function
         <CALL n, @args> ->
           let t = Regs.new_temp 1 in
-          <AFTER,
-            <DEFTEMP t, <CALL n, @(List.map do_tree args)>>,
+          <AFTER,
+            <DEFTEMP t, <CALL n, @(List.map do_tree args)>>,
             <TEMP t>>
       | <w, @args> ->
           <w, @(List.map do_tree args)> in
@@ -369,13 +385,13 @@
 let do_proc lab lev nargs (Block (_, body, fsize, nregv)) =
   level := lev+1;
   retlab := label ();
-  let code0 =
+  let code0 =
     show "Initial code" (Optree.canon <SEQ, gen_stmt body, <LABEL !retlab>>) in
   Regs.init ();
   let code1 = if !optlevel < 1 then code0 else
       show "After simplification" (Jumpopt.optimise (Simp.optimise code0)) in
-  let code2 = if !optlevel < 2 then
-      show "After unnesting" (unnest code1)
+  let code2 = if !optlevel < 2 then
+      show "After unnesting" (unnest code1)
     else
       show "After sharing" (Share.traverse code1) in
   Tran.translate lab nargs !fsize !nregv (flatten code2)
@@ -387,7 +403,7 @@
 let get_decls (Block (decls, _, _, _)) = decls

 (* |gen_proc| -- translate a procedure, ignore other declarations *)
-let rec gen_proc =
+let rec gen_proc =
   function
```

```
       ProcDecl (Heading (x, _, _), block) ->
         let d = get_def x in
@@ -416,4 +432,3 @@
   List.iter gen_global !globdefs;
   List.iter (fun (lab, s) -> Target.emit_string lab s) (string_table ());
   Target.postamble ()
-
diff -r 5e3fa9d71839 lab4/tree.ml
--- a/lab4/tree.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/tree.ml        Thu Jan 30 03:56:01 2020 +0000
@@ -5,7 +5,7 @@
 open Print

 (* |name| -- type for applied occurrences, with mutable annotations *)
-type name =
+type name =
   { x_name: ident;              (* Name of the reference *)
     x_line: int;                (* Line number *)
     mutable x_def: def option } (* Definition in scope *)
@@ -15,56 +15,57 @@

 and block = Block of decl list * stmt * int ref * int ref

-and decl =
+and decl =
     ConstDecl of ident * expr
   | VarDecl of def_kind * ident list * typexpr
   | TypeDecl of (ident * typexpr) list
   | ProcDecl of proc_heading * block
   | PParamDecl of proc_heading

-and proc_heading = Heading of name * decl list * typexpr option
+and proc_heading = Heading of name * decl list * typexpr option

-and stmt =
+and stmt =
   { s_guts: stmt_guts;
     s_line: int }

 and stmt_guts =
-    Skip
+    Skip
   | Seq of stmt list
   | Assign of expr * expr
-  | ProcCall of name * expr list
+  | ProcCall of name * expr list
```

```
    | 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
+   | ForInStmt of name * expr * stmt * def option ref

-and expr =
-  { e_guts: expr_guts;
-    mutable e_type: ptype;
+and expr =
+  { e_guts: expr_guts;
+    mutable e_type: ptype;
     mutable e_value: int option }

 and expr_guts =
     Constant of int * ptype
    | Variable of name
-   | Sub of expr * expr
+   | Sub of expr * expr
    | Select of expr * name
    | Deref of expr
    | String of Optree.symbol * int
    | Nil
    | FuncCall of name * expr list
-   | Monop of Optree.op * expr
+   | Monop of Optree.op * expr
    | Binop of Optree.op * expr * expr

-and typexpr =
-    TypeName of name
+and typexpr =
+    TypeName of name
    | Array of expr * typexpr
    | Record of decl list
    | Pointer of typexpr

 (* |makeExpr| -- construct an expression node with dummy annotations *)
-let makeExpr e =
+let makeExpr e =
   { e_guts = e; e_type = voidtype; e_value = None }

 (* |makeStmt| -- construct a stmt node *)
@@ -80,8 +81,8 @@
     | ss -> makeStmt (Seq ss, 0)
```

```
 let get_def x =
-  match x.x_def with
-      Some d -> d
+  match x.x_def with
+      Some d -> d
     | None -> failwith (sprintf "missing def of $" [fId x.x_name])

 (* |MakeBlock| -- construct a block node with dummy annotations *)
@@ -90,7 +91,7 @@

 (* Grinder *)

-let fTail f xs =
+let fTail f xs =
   let g prf = List.iter (fun x -> prf " $" [f x]) xs in fExt g

 let fList f =
@@ -105,11 +106,11 @@
       [] -> fMeta "(BLOCK $)" [fStmt stmts]
     | _ -> fMeta "(BLOCK (DECLS$) $)" [fTail(fDecl) decls; fStmt stmts]

-and fDecl =
+and fDecl =
   function
-      ConstDecl (x, e) ->
+      ConstDecl (x, e) ->
         fMeta "(CONST $ $)" [fId x; fExpr e]
-    | VarDecl (kind, xs, te) ->
+    | VarDecl (kind, xs, te) ->
         fMeta "($ $ $)" [fKind kind; fList(fId) xs; fType te]
     | TypeDecl tds ->
         let f (x, te) = fMeta "($ $)" [fId x; fType te] in
@@ -131,7 +132,7 @@
   let res = match te with Some t -> fType t | None -> fStr "VOID" in
   fMeta "($ $ $)" [fName p; fList(fDecl) fps; res]

-and fStmt s =
+and fStmt s =
   match s.s_guts with
       Skip -> fStr "(SKIP)"
     | Seq stmts -> fMeta "(SEQ$)" [fTail(fStmt) stmts]
@@ -139,7 +140,7 @@
     | ProcCall (p, aps) -> fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
     | Return (Some e) -> fMeta "(RETURN $)" [fExpr e]
     | Return None -> fStr "(RETURN)"
```

```
-    | IfStmt (test, thenpt, elsept) ->
+    | IfStmt (test, thenpt, elsept) ->
         fMeta "(IF $ $ $)" [fExpr test; fStmt thenpt; fStmt elsept]
     | WhileStmt (test, body) ->
         fMeta "(WHILE $ $)" [fExpr test; fStmt body]
@@ -150,6 +151,8 @@
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
         fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]
+    | ForInStmt (var, arr, body, _) ->
+        fMeta "(FORIN $ $ $ $)" [fName var; fExpr arr; fStmt body]

 and fExpr e =
   match e.e_guts with
@@ -162,9 +165,9 @@
     | Nil -> fStr "(NIL)"
     | FuncCall (p, aps) ->
         fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
-    | Monop (w, e1) ->
+    | Monop (w, e1) ->
         fMeta "($ $)" [Optree.fOp w; fExpr e1]
-    | Binop (w, e1, e2) ->
+    | Binop (w, e1, e2) ->
         fMeta "($ $ $)" [Optree.fOp w; fExpr e1; fExpr e2]

 and fType =
@@ -174,5 +177,5 @@
     | Record fields -> fMeta "(RECORD$)" [fTail(fDecl) fields]
     | Pointer t1 -> fMeta "(POINTER $)" [fType t1]

-let print_tree fp pfx (Prog (body, _)) =
+let print_tree fp pfx (Prog (body, _)) =
   fgrindf fp pfx "(PROGRAM $)" [fBlock body]
diff -r 5e3fa9d71839 lab4/tree.mli
--- a/lab4/tree.mli        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/tree.mli        Thu Jan 30 03:56:01 2020 +0000
@@ -18,7 +18,7 @@
 *)

 (* |name| -- type for applied occurrences, with mutable annotations *)
-type name =
+type name =
   { x_name: ident;              (* Name of the reference *)
     x_line: int;                (* Line number *)
     mutable x_def: def option } (* Definition in scope *)
@@ -30,50 +30,51 @@
```

```
 and block = Block of decl list * stmt * int ref * int ref

-and decl =
+and decl =
     ConstDecl of ident * expr
   | VarDecl of def_kind * ident list * typexpr
   | TypeDecl of (ident * typexpr) list
   | ProcDecl of proc_heading * block
   | PParamDecl of proc_heading

-and proc_heading = Heading of name * decl list * typexpr option
+and proc_heading = Heading of name * decl list * typexpr option

-and stmt =
+and stmt =
   { s_guts: stmt_guts;
     s_line: int }

 and stmt_guts =
-    Skip
+    Skip
   | Seq of stmt list
   | Assign of expr * expr
-  | ProcCall of name * expr list
+  | ProcCall of name * expr list
   | 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
+  | ForInStmt of name * expr * stmt * def option ref

-and expr =
-  { e_guts: expr_guts;
-    mutable e_type: ptype;
+and expr =
+  { e_guts: expr_guts;
+    mutable e_type: ptype;
     mutable e_value: int option }

 and expr_guts =
     Constant of int * ptype
   | Variable of name
-  | Sub of expr * expr
```

```
+   | Sub of expr * expr
    | Select of expr * name
    | Deref of expr
    | String of Optree.symbol * int
    | Nil
    | FuncCall of name * expr list
-   | Monop of Optree.op * expr
+   | Monop of Optree.op * expr
    | Binop of Optree.op * expr * expr

-and typexpr =
-    TypeName of name
+and typexpr =
+    TypeName of name
    | Array of expr * typexpr
    | Record of decl list
    | Pointer of typexpr
```

4 Tests For Task 1

4.1

```
var i: integer;
var v: array 5 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end
end.

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

(*@ picoPascal compiler output
```

```
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   for i := 0 to 4 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
        mov r4, #4
.L2:
        set r0, _i
        ldr r0, [r0]
        cmp r0, r4
        bgt .L3
@     v[i] := i+1
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L2
.L3:
@   for it in v do
        mov r6, #0
.L4:
        set r5, _v
        mov r0, #5
        sub r0, r0, #1
        cmp r6, r0
        bgt .L5
        set r0, _v
        mov r1, #4
```

```
        mul r1, r6, r1
        add r5, r0, r1
@     print_num(it);
        ldr r0, [r5]
        bl print_num
@     newline()
        bl newline
        add r6, r6, #1
        b .L4
.L5:
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
        .comm _v, 20, 4
@ End*)
```

4.2

```
type point = record x,y:integer end;
var a: array 10 of point;
var i, sum: integer;
begin
i := 0;
for r in a do
r.x := i mod 3;
r.y := i mod 5;
i := i+1
end;
(* x values: 0, 1, 2, 0, 1, 2, 0, 1, 2, 0 *)
(* y values: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4 *)
sum := 0;
for r in a do
sum := sum + r.x * r.y
end;
(* sum = 0*0 + 1*1 + 2*2 + 0*3 + 1*4 + 2*0 + 0*1 + 1*2 + 2*3 + 0*4 = 17 *)
print_num(sum); newline()
end.

(*<<
17
>>*)

(*@ picoPascal compiler output
        .include "fixup.s"
```

```
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #16
@ i := 0;
        mov r0, #0
        set r1, _i
        str r0, [r1]
@ for r in a do
        mov r4, #0
.L2:
        set r0, _a
        str r0, [fp, #-8]
        mov r0, #10
        sub r0, r0, #1
        cmp r4, r0
        bgt .L3
        set r0, _a
        mov r1, #8
        mul r1, r4, r1
        add r0, r0, r1
        str r0, [fp, #-8]
@ r.x := i mod 3;
        mov r1, #3
        set r0, _i
        ldr r0, [r0]
        bl int_mod
        ldr r1, [fp, #-8]
        str r0, [r1]
@ r.y := i mod 5;
        mov r1, #5
        set r0, _i
        ldr r0, [r0]
        bl int_mod
        ldr r1, [fp, #-8]
        str r0, [r1, #4]
@ i := i+1
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
```

```
        add r4, r4, #1
        b .L2
.L3:
@ sum := 0;
        mov r0, #0
        set r1, _sum
        str r0, [r1]
@ for r in a do
        mov r5, #0
.L4:
        set r0, _a
        str r0, [fp, #-16]
        mov r0, #10
        sub r0, r0, #1
        cmp r5, r0
        bgt .L5
        set r0, _a
        mov r1, #8
        mul r1, r5, r1
        add r0, r0, r1
        str r0, [fp, #-16]
@ sum := sum + r.x * r.y
        set r0, _sum
        ldr r0, [r0]
        ldr r1, [fp, #-16]
        ldr r1, [r1]
        ldr r2, [fp, #-16]
        ldr r2, [r2, #4]
        mul r1, r1, r2
        add r0, r0, r1
        set r1, _sum
        str r0, [r1]
        add r5, r5, #1
        b .L4
.L5:
@ print_num(sum); newline()
        set r0, _sum
        ldr r0, [r0]
        bl print_num
        bl newline
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _a, 80, 4
        .comm _i, 4, 4
```

```
        .comm _sum, 4, 4
@ End*)
```

4.3

```
var i: integer;
var v: array 5 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);

    for iter in v do
      if iter = 5 then
        newline()
      end
    end
  end
end.

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

(*@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@   for i := 0 to 4 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
```

```
        mov r4, #4
.L2:
        set r0, _i
        ldr r0, [r0]
        cmp r0, r4
        bgt .L3
@     v[i] := i+1
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L2
.L3:
@   for it in v do
        mov r0, #0
        str r0, [fp, #-8]
.L4:
        set r5, _v
        ldr r0, [fp, #-8]
        mov r1, #5
        sub r1, r1, #1
        cmp r0, r1
        bgt .L5
        set r0, _v
        ldr r1, [fp, #-8]
        mov r2, #4
        mul r1, r1, r2
        add r5, r0, r1
@     print_num(it);
        ldr r0, [r5]
        bl print_num
@     for iter in v do
        mov r0, #0
        str r0, [fp, #-4]
.L6:
```

```
        set r6, _v
        ldr r0, [fp, #-4]
        mov r1, #5
        sub r1, r1, #1
        cmp r0, r1
        bgt .L7
        set r0, _v
        ldr r1, [fp, #-4]
        mov r2, #4
        mul r1, r1, r2
        add r6, r0, r1
@       if iter = 5 then
        ldr r0, [r6]
        cmp r0, #5
        beq .L8
        b .L9
.L8:
@         newline()
        bl newline
        b .L10
.L9:
.L10:
        ldr r0, [fp, #-4]
        add r0, r0, #1
        str r0, [fp, #-4]
        b .L6
.L7:
        ldr r0, [fp, #-8]
        add r0, r0, #1
        str r0, [fp, #-8]
        b .L4
.L5:
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
        .comm _v, 20, 4
@ End*)
```

4.4

```
var i: integer;
var v: array 5 of integer;

begin
```

```
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end;

  for it in v do
    print_num(it);
    newline()
  end

end.

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

(*@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@   for i := 0 to 4 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
        mov r4, #4
.L2:
        set r0, _i
```

```
        ldr r0, [r0]
        cmp r0, r4
        bgt .L3
@     v[i] := i+1
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L2
.L3:
@   for it in v do
        mov r6, #0
.L4:
        set r5, _v
        mov r0, #5
        sub r0, r0, #1
        cmp r6, r0
        bgt .L5
        set r0, _v
        mov r1, #4
        mul r1, r6, r1
        add r5, r0, r1
@     print_num(it);
        ldr r0, [r5]
        bl print_num
@     newline()
        bl newline
        add r6, r6, #1
        b .L4
.L5:
@   for it in v do
        mov r0, #0
        str r0, [fp, #-8]
.L6:
        set r0, _v
```

```
        str r0, [fp, #-4]
        ldr r0, [fp, #-8]
        mov r1, #5
        sub r1, r1, #1
        cmp r0, r1
        bgt .L7
        set r0, _v
        ldr r1, [fp, #-8]
        mov r2, #4
        mul r1, r1, r2
        add r0, r0, r1
        str r0, [fp, #-4]
@     print_num(it);
        ldr r0, [fp, #-4]
        ldr r0, [r0]
        bl print_num
@     newline()
        bl newline
        ldr r0, [fp, #-8]
        add r0, r0, #1
        str r0, [fp, #-8]
        b .L6
.L7:
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
        .comm _v, 20, 4
@ End*)
```

4.5

```
var i: integer;
var v: array 5 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end;
```

```
  print_num(it);
end.

(*<<
error "it is not declared"
>>*)

(**)
```

4.6

```
var i, it: integer;
var v, w: array 2 of integer;

begin
  for i := 0 to 1 do
    v[i] := 5
  end;
  for i := 0 to 1 do
    w[i] := i+1
  end;

  it := 10;
  for it in v do
    print_num(it);
    newline();
    for it in w do
      print_num(it);
      newline()
    end;
  end;
  print_num(it);
  newline();
end.

(*<<
5
1
2
5
1
2
10
>>*)

(*@ picoPascal compiler output
```

```
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #16
@   for i := 0 to 1 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
        mov r4, #1
.L2:
        set r0, _i
        ldr r0, [r0]
        cmp r0, r4
        bgt .L3
@     v[i] := 5
        mov r0, #5
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L2
.L3:
@   for i := 0 to 1 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
        mov r5, #1
.L4:
        set r0, _i
        ldr r0, [r0]
        cmp r0, r5
        bgt .L5
@     w[i] := i+1
```

```
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _w
        set r2, _i
        ldr r2, [r2]
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L4
.L5:
@   it := 10;
        mov r0, #10
        set r1, _it
        str r0, [r1]
@   for it in v do
        mov r0, #0
        str r0, [fp, #-12]
.L6:
        set r6, _v
        ldr r0, [fp, #-12]
        mov r1, #2
        sub r1, r1, #1
        cmp r0, r1
        bgt .L7
        set r0, _v
        ldr r1, [fp, #-12]
        mov r2, #4
        mul r1, r1, r2
        add r6, r0, r1
@     print_num(it);
        ldr r0, [r6]
        bl print_num
@     newline();
        bl newline
@     for it in w do
        mov r0, #0
        str r0, [fp, #-8]
.L8:
        set r0, _w
```

```
        str r0, [fp, #-4]
        ldr r0, [fp, #-8]
        mov r1, #2
        sub r1, r1, #1
        cmp r0, r1
        bgt .L9
        set r0, _w
        ldr r1, [fp, #-8]
        mov r2, #4
        mul r1, r1, r2
        add r0, r0, r1
        str r0, [fp, #-4]
@       print_num(it);
        ldr r0, [fp, #-4]
        ldr r0, [r0]
        bl print_num
@       newline()
        bl newline
        ldr r0, [fp, #-8]
        add r0, r0, #1
        str r0, [fp, #-8]
        b .L8
.L9:
@   end;
        ldr r0, [fp, #-12]
        add r0, r0, #1
        str r0, [fp, #-12]
        b .L6
.L7:
@   print_num(it);
        set r0, _it
        ldr r0, [r0]
        bl print_num
@   newline();
        bl newline
@ end.
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
        .comm _it, 4, 4
        .comm _v, 8, 4
        .comm _w, 8, 4
@ End*)
```

4.7

```
var i: integer;
var v: array 0 of integer;

begin
  for i := 0 to 4 do
    v[i] := i+1
  end;

  for it in v do
    print_num(it);
    newline()
  end
end.

(*<<
>>*)

(*@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   for i := 0 to 4 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
        mov r4, #4
.L2:
        set r0, _i
        ldr r0, [r0]
        cmp r0, r4
        bgt .L3
@     v[i] := i+1
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #4
```

```
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L2
.L3:
@   for it in v do
        mov r6, #0
.L4:
        set r5, _v
        mov r0, #0
        sub r0, r0, #1
        cmp r6, r0
        bgt .L5
        set r0, _v
        mov r1, #4
        mul r1, r6, r1
        add r5, r0, r1
@     print_num(it);
        ldr r0, [r5]
        bl print_num
@     newline()
        bl newline
        add r6, r6, #1
        b .L4
.L5:
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
        .comm _v, 0, 4
@ End*)
```

4.8

```
var i : integer;
var v : array 7 of boolean;
begin
for i:=0 to 6 do
  if i mod 2 = 1 then
    v[i] := true
```

```
  else
    v[i] := false
    end;
end;
for i in v do
  if i = true then
    print_num (1)
  else
    print_num (2)
  end;
  newline()
end;
end.

(*<<
2
1
2
1
2
1
2
>>*)

(*@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ for i:=0 to 6 do
        mov r0, #0
        set r1, _i
        str r0, [r1]
        mov r4, #6
.L2:
        set r0, _i
        ldr r0, [r0]
        cmp r0, r4
        bgt .L3
@   if i mod 2 = 1 then
        mov r1, #2
        set r0, _i
```

```
        ldr r0, [r0]
        bl int_mod
        cmp r0, #1
        beq .L4
        b .L5
.L4:
@     v[i] := true
        mov r0, #1
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #1
        mul r2, r2, r3
        add r1, r1, r2
        strb r0, [r1]
        b .L6
.L5:
@     v[i] := false
        mov r0, #0
        set r1, _v
        set r2, _i
        ldr r2, [r2]
        mov r3, #1
        mul r2, r2, r3
        add r1, r1, r2
        strb r0, [r1]
.L6:
@ end;
        set r0, _i
        ldr r0, [r0]
        add r0, r0, #1
        set r1, _i
        str r0, [r1]
        b .L2
.L3:
@ for i in v do
        mov r6, #0
.L7:
        set r5, _v
        mov r0, #7
        sub r0, r0, #1
        cmp r6, r0
        bgt .L8
        set r0, _v
        mov r1, #1
        mul r1, r6, r1
```

```
        add r5, r0, r1
@   if i = true then
        ldrb r0, [r5]
        cmp r0, #1
        beq .L9
        b .L10
.L9:
@     print_num (1)
        mov r0, #1
        bl print_num
        b .L11
.L10:
@     print_num (2)
        mov r0, #2
        bl print_num
.L11:
@   newline()
        bl newline
        add r6, r6, #1
        b .L7
.L8:
@ end.
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
        .comm _v, 7, 4
@ End*)
```

5 Diff For Task 2

```
diff -r 5e3fa9d71839 lab4/check.ml
--- a/lab4/check.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/check.ml        Thu Jan 30 03:59:11 2020 +0000
@@ -7,6 +7,7 @@
 open Print
 open Lexer
 open Mach
+open Set

 (* EXPRESSIONS *)

@@ -37,15 +38,15 @@
 (* |lookup_def| -- find definition of a name, give error if none *)
```

```
 let lookup_def x env =
   err_line := x.x_line;
-  try
-    let d = lookup x.x_name env in
+  try
+    let d = lookup x.x_name env in
     x.x_def <- Some d; d
-  with Not_found ->
+  with Not_found ->
     sem_error "$ is not declared" [fId x.x_name]

 (* |add_def| -- add definition to envmt, give error if already declared *)
 let add_def d env =
-  try define d env with
+  try define d env with
     Exit -> sem_error "$ is already declared" [fId d.d_tag]

 (* |check_monop| -- check application of unary operator *)
@@ -78,19 +79,19 @@
 (* |try_monop| -- propagate constant through unary operation *)
 let try_monop w =
   function
-      Some x -> Some (do_monop w x)
+      Some x -> Some (do_monop w x)
     | None -> None

 (* |try_binop| -- propagate constant through unary operation *)
 let try_binop w v1 v2 =
-  match (v1, v2) with
+  match (v1, v2) with
       (Some x1, Some x2) -> Some (do_binop w x1 x2)
     | _ -> None

 (* |has_value| -- check if object is suitable for use in expressions *)
-let has_value d =
+let has_value d =
   match d.d_kind with
-      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
+      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -99,10 +100,10 @@
       Variable x ->
         let d = get_def x in
         begin
```

```
-          match d.d_kind with
+          match d.d_kind with
               VarDef | VParamDef | CParamDef ->
                 d.d_mem <- d.d_mem || addressible
-           | _ ->
+           | _ ->
                 sem_error "$ is not a variable" [fId x.x_name]
         end
     | Sub (a, i) -> check_var a addressible
@@ -118,15 +119,15 @@
 (* |expr_type| -- return type of expression *)
 and expr_type e env =
   match e.e_guts with
-      Variable x ->
-        let d = lookup_def x env in
+      Variable x ->
+        let d = lookup_def x env in
+        if not (has_value d) then
+          sem_error "$ is not a variable" [fId x.x_name];
         if d.d_level < !level then d.d_mem <- true;
         begin
-          match d.d_kind with
+          match d.d_kind with
               ConstDef v ->
-                e.e_value <- Some v
+                e.e_value <- Some v
            | _ -> ()
         end;
         d.d_type
@@ -162,17 +163,17 @@
     | Constant (n, t) -> e.e_value <- Some n; t
     | String (lab, n) -> row n character
     | Nil -> e.e_value <- Some 0; addrtype
-    | FuncCall (p, args) ->
+    | FuncCall (p, args) ->
         let v = ref None in
         let t1 = check_funcall p args env v in
         if same_type t1 voidtype then
           sem_error "$ does not return a result" [fId p.x_name];
         e.e_value <- !v; t1
-    | Monop (w, e1) ->
+    | Monop (w, e1) ->
         let t = check_monop w (check_expr e1 env) in
         e.e_value <- try_monop w e1.e_value;
         t
-    | Binop (w, e1, e2) ->
```

```
+    | Binop (w, e1, e2) ->
         let t = check_binop w (check_expr e1 env) (check_expr e2 env) in
         e.e_value <- try_binop w e1.e_value e2.e_value;
         t
@@ -185,7 +186,7 @@
         check_libcall q args env v; d.d_type
     | ProcDef | PParamDef ->
         let p = get_proc d.d_type in
-        check_args p.p_fparams args env;
+        check_args p.p_fparams args env;
         p.p_result
     | _ -> sem_error "$ is not a procedure" [fId f.x_name]

@@ -202,22 +203,22 @@
         let t1 = check_expr arg env in
         if not (same_type formal.d_type t1) then
           sem_error "argument has wrong type" [];
-        if formal.d_kind = VParamDef then
+        if formal.d_kind = VParamDef then
           check_var arg true
     | PParamDef ->
         let pf = get_proc formal.d_type in
-        let x = (match arg.e_guts with Variable x -> x
+        let x = (match arg.e_guts with Variable x -> x
           | _ -> sem_error "procedure argument must be a proc name" []) in
         let actual = lookup_def x env in
         begin
-          match actual.d_kind with
+          match actual.d_kind with
               ProcDef | PParamDef  ->
                 let pa = get_proc actual.d_type in
                 if not (match_args pf.p_fparams pa.p_fparams) then
                   sem_error "argument lists don't match" [];
                 if not (same_type pf.p_result pa.p_result) then
                   sem_error "result types don't match" []
-            | _ ->
+            | _ ->
                 sem_error "argument $ is not a procedure" [fId x.x_name]
         end
     | _ -> failwith "bad formal"
@@ -235,7 +236,7 @@
         sem_error "argument of $ has wrong type" [fLibId q.q_id] in
     List.iter2 check q.q_argtypes args
   end;
-  match q.q_id with
+  match q.q_id with
```

```
       ChrFun ->
         let e1 = List.hd args in
         v := e1.e_value
@@ -253,7 +254,7 @@
         check_var (List.hd args) true
     | NewProc ->
         let t1 = check_expr (List.hd args) env in
-        if not (is_pointer t1) then
+        if not (is_pointer t1) then
           sem_error "parameter of new must be a pointer" [];
         check_var (List.hd args) false
     | ArgvProc ->
@@ -284,14 +285,151 @@

 (* check_dupcases -- check for duplicate case labels *)
 let check_dupcases vs =
-  let rec chk =
+  let rec chk =
     function
         [] | [_] -> ()
-      | x :: (y :: ys as rest) ->
+      | x :: (y :: ys as rest) ->
           if x = y then sem_error "duplicate case label" [];
           chk rest in
   chk (List.sort compare vs)

+(* Global sets for Commute checking -> they store the variables used in body1 and body2 of
+module SetIdent = Set.Make(
+  struct
+    let compare = Pervasives.compare
+    type t = ident
+  end
+  )
+
+let set1_changed = ref SetIdent.empty
+let set1 = ref SetIdent.empty
+let set2_changed = ref SetIdent.empty
+let set2 = ref SetIdent.empty
+
+
+(* Task 2 functions *)
+
+let rec check_expr_var_changed e env cnt =
+  match e.e_guts with
+    Variable x ->
+      if cnt = 1 then
```

```
+        set1_changed := SetIdent.add (x.x_name) (!set1_changed)
+      else
+        set2_changed := SetIdent.add (x.x_name) (!set2_changed)
+
+  | Sub (a, b) ->
+    check_expr_var_changed a env cnt
+
+  | Select (a, b) ->
+    check_expr_var_changed a env cnt
+
+  | _-> ()
+
+let rec check_expr_var e env cnt =
+    match e.e_guts with
+      Variable x ->
+        if cnt = 1 then
+          set1 := SetIdent.add (x.x_name) (!set1)
+        else
+          set2 := SetIdent.add (x.x_name) (!set2)
+
+    | Sub (a, b) ->
+      check_expr_var a env cnt;
+      check_expr_var b env cnt
+
+    | Select (a, b) ->
+      check_expr_var a env cnt
+
+    | Deref (x) ->
+      check_expr_var x env cnt
+
+    | Monop (sym, x) ->
+      check_expr_var x env cnt
+
+    | Binop (sym, x, y) ->
+      check_expr_var x env cnt;
+      check_expr_var y env cnt
+
+    | _-> ()
+
+
+let rec check_stmt_var_changed s env alloc cnt =
+  match s.s_guts with
+    Skip -> ()
+
+  | Seq ss ->
+      List.iter (fun s1 -> check_stmt_var_changed s1 env alloc cnt) ss
```

```
+
+(*We do not need to check rhs because the variable changed is only the one on the left *)
+  | Assign (lhs, rhs) ->
+      check_expr_var_changed lhs env cnt
+
+  | IfStmt (cond, thenpt, elsept) ->
+      check_stmt_var_changed thenpt env alloc cnt;
+      check_stmt_var_changed elsept env alloc cnt
+
+  | WhileStmt (cond, body) ->
+      check_stmt_var_changed body env alloc cnt
+
+  | RepeatStmt (body, test) ->
+      check_stmt_var_changed body env alloc cnt
+
+  | ForStmt (var, lo, hi, body, upb) ->
+      check_expr_var_changed var env cnt;
+      check_stmt_var_changed body env alloc cnt
+
+  | CaseStmt (sel, arms, deflt) ->
+      let check_arm_var (lab, body) = check_stmt_var_changed body env alloc cnt in
+      List.iter check_arm_var arms;
+      check_stmt_var_changed deflt env alloc cnt
+
+  | CommuteStmt (body1, body2) ->
+    check_stmt_var_changed body1 env alloc cnt;
+    check_stmt_var_changed body2 env alloc cnt
+
+  | _ -> ()
+
+let rec check_stmt_var s env alloc cnt =
+    match s.s_guts with
+      Skip -> ()
+
+    | Seq ss ->
+        List.iter (fun s1 -> check_stmt_var s1 env alloc cnt) ss
+
+    | Assign (lhs, rhs) ->
+        check_expr_var lhs env cnt;
+        check_expr_var rhs env cnt
+
+    | IfStmt (cond, thenpt, elsept) ->
+        check_expr_var cond env cnt;
+        check_stmt_var thenpt env alloc cnt;
+        check_stmt_var elsept env alloc cnt
+
```

```
+    | WhileStmt (cond, body) ->
+        check_expr_var cond env cnt;
+        check_stmt_var body env alloc cnt
+
+    | RepeatStmt (body, test) ->
+        check_expr_var test env cnt;
+        check_stmt_var body env alloc cnt
+
+    | ForStmt (var, lo, hi, body, upb) ->
+        check_expr_var lo env cnt;
+        check_expr_var hi env cnt;
+        check_expr_var var env cnt;
+        check_stmt_var body env alloc cnt
+
+    | CaseStmt (sel, arms, deflt) ->
+        let check_arm_var (lab, body) = check_stmt_var body env alloc cnt in
+        List.iter check_arm_var arms;
+        check_stmt_var deflt env alloc cnt
+
+    | CommuteStmt (body1, body2) ->
+      check_stmt_var body1 env alloc cnt;
+      check_stmt_var body2 env alloc cnt
+
+    | _ -> ()
+
 (* |check_stmt| -- check and annotate a statement *)
 let rec check_stmt s env alloc =
   err_line := s.s_line;
@@ -332,7 +470,7 @@
     | IfStmt (cond, thenpt, elsept) ->
         let ct = check_expr cond env in
         if not (same_type ct boolean) then
-          sem_error "test in if statement must be a boolean" [];
+          sem_error "test in if statement must be a boolean" [];
         check_stmt thenpt env alloc;
         check_stmt elsept env alloc

@@ -378,6 +516,26 @@
         check_dupcases vs;
         check_stmt deflt env alloc

+    | CommuteStmt (body1, body2) ->
+        check_stmt body1 env alloc;
+        check_stmt body2 env alloc;
+
+        check_stmt_var_changed body1 env alloc 1;
```

```
+        check_stmt_var body1 env alloc 1;
+        check_stmt_var_changed body2 env alloc 2;
+        check_stmt_var body2 env alloc 2;
+
+        if SetIdent.is_empty (SetIdent.inter (!set1) (!set2_changed)) = false then
+        (*use failwith to make compiler halt*)
+            failwith "Possibly incorrect";
+        if SetIdent.is_empty (SetIdent.inter (!set1_changed) (!set2)) = false then
+            failwith "Possibly incorrect";
+
+        set1 := SetIdent.empty;
+        set2 := SetIdent.empty;
+        set1_changed := SetIdent.empty;
+        set2_changed := SetIdent.empty
+

 (* TYPES AND DECLARATIONS *)

@@ -403,7 +561,7 @@

 (* local_alloc -- allocate locals downward in memory *)
 let local_alloc size nreg d =
-  if !regvars && not d.d_mem && scalar (d.d_type)
+  if !regvars && not d.d_mem && scalar (d.d_type)
       && !nreg < Mach.nregvars then begin
     d.d_addr <- Register !nreg; incr nreg
   end
@@ -440,7 +598,7 @@
   List.iter h ds

 (* |check_typexpr| -- check a type expression, returning the ptype *)
-let rec check_typexpr te env =
+let rec check_typexpr te env =
   match te with
       TypeName x ->
         let d = lookup_typename x env in
@@ -461,7 +619,7 @@
         let r = { r_size = !size; r_align = max_align } in
         mk_type (RecordType defs) r
     | Pointer te ->
-        let t =
+        let t =
           match te with
               TypeName x ->
                 let d = lookup_typename x env in
@@ -475,7 +633,7 @@
```

```
         mk_type (PointerType t) addr_rep

 (* |check_decl| -- check a declaration and add it to the environment *)
-and check_decl d env =
+and check_decl d env =
   (* All types of declaration are mixed together in the AST *)
   match d with
       ConstDecl (x, e) ->
@@ -492,12 +650,12 @@
         end
     | VarDecl (kind, xs, te) ->
         let t = check_typexpr te env in
-        let def x env =
+        let def x env =
           let d = make_def x kind t in
           add_def d env in
         Util.accum def xs env
     | TypeDecl tds ->
-        let tds' =
+        let tds' =
           List.map (function (x, te) -> (x, te, ref voidtype)) tds in
         let add_hole (x, te, h) env1 =
           add_def (make_def x (HoleDef h) voidtype) env1 in
@@ -505,7 +663,7 @@
         let redefine (x, te, h) env1 =
           let t = check_typexpr te env1 in
           h := t; replace (make_def x TypeDef t) env1 in
-        Util.accum redefine tds' env'
+        Util.accum redefine tds' env'
     | ProcDecl (Heading (x, _, _) as heading, body) ->
         let t = check_heading env heading in
         let d = make_def x.x_name ProcDef t in
@@ -515,7 +673,7 @@
         let t = check_heading env heading in
         let d = make_def x.x_name PParamDef t in
         add_def d env
-
+
 (* |check_heading| -- process a procedure heading into a procedure type *)
 and check_heading env (Heading (x, fparams, result)) =
   err_line := x.x_line;
@@ -554,7 +712,7 @@
   align max_align fsize

 (* |check_bodies| -- check bodies of procedure declarations *)
-and check_bodies env ds =
```

```
+and check_bodies env ds =
   let check =
     function
         ProcDecl (Heading(x, _, _), body) ->
@@ -570,19 +728,19 @@

 (* INITIAL ENVIRONMENT *)

-let defn (x, k, t) env =
+let defn (x, k, t) env =
   let d = { d_tag = intern x; d_kind = k; d_type = t;
             d_level = 0; d_mem = false; d_addr = Nowhere } in
   define d env

-let libproc i n ts =
+let libproc i n ts =
   LibDef { q_id = i; q_nargs = n; q_argtypes = ts }

 let operator op ts =
   libproc (Operator op) (List.length ts) ts

 (* |init_env| -- environment for whole program *)
-let init_env =
+let init_env =
   Util.accum defn
     [ ("integer", TypeDef, integer);
       ("char", TypeDef, character);
@@ -620,4 +778,5 @@
   let alloc = local_alloc fsize nregv in
   check_stmt ss env alloc;
   align max_align fsize;
-  glodefs := top_block env
+  glodefs := top_block env;
+  failwith "Correct"
diff -r 5e3fa9d71839 lab4/dict.ml
--- a/lab4/dict.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/dict.ml        Thu Jan 30 03:59:11 2020 +0000
@@ -43,14 +43,14 @@
     | Nowhere -> fStr "*nowhere*"

 (* |libid| -- type of picoPascal library procedures *)
-type libid = ChrFun | OrdFun | PrintNum | PrintChar | PrintString
+type libid = ChrFun | OrdFun | PrintNum | PrintChar | PrintString
   | NewLine | ReadChar | ExitProc | NewProc | ArgcFun | ArgvProc
   | OpenIn | CloseIn | Operator of Optree.op
```

```
 (* |lib_name| -- name of a library procedure *)
-let lib_name x =
+let lib_name x =
   match x with
-      PrintNum -> "print_num" | PrintChar -> "print_char"
+      PrintNum -> "print_num" | PrintChar -> "print_char"
     | PrintString -> "print_string" | NewLine -> "newline"
     | ReadChar -> "read_char" | ChrFun -> "chr" | OrdFun -> "ord"
     | ExitProc -> "exit" | NewProc -> "new"
@@ -73,7 +73,7 @@
 *)

 (* |def_kind| -- kinds of definition *)
-type def_kind =
+type def_kind =
     ConstDef of int             (* Constant (value) *)
   | StringDef                   (* String *)
   | TypeDef                     (* Type *)
@@ -88,7 +88,7 @@
   | DummyDef                    (* Dummy *)

 (* |def| -- definitions in environment *)
-and def =
+and def =
   { d_tag: ident;               (* Name *)
     d_kind: def_kind;           (* Kind of object *)
     d_type: ptype;              (* Type *)
@@ -99,7 +99,7 @@
 and basic_type = VoidType | IntType | CharType | BoolType | AddrType

 (* |ptype| -- picoPascal types *)
-and ptype =
+and ptype =
   { t_id: int;                  (* Unique identifier *)
     t_guts: type_guts;          (* Shape of the type *)
     t_rep: Mach.metrics }
@@ -123,10 +123,10 @@
     q_nargs: int;
     q_argtypes: ptype list }

-module IdMap =
-  Map.Make(struct
-    type t = ident
-    let compare = compare
+module IdMap =
+  Map.Make(struct
```

```
+    type t = ident
+    let compare = compare
   end)

 type environment = Env of (def list * def IdMap.t)
@@ -144,13 +144,13 @@
   let rec search =
     function
         [] -> raise Not_found
-      | d::ds ->
+      | d::ds ->
           if x = d.d_tag then d else search ds in
   search ds

 let can f x = try f x; true with Not_found -> false

-let define d (Env (b, m)) =
+let define d (Env (b, m)) =
   if can (find_def d.d_tag) b then raise Exit;
   Env (d::b, add_def d m)

@@ -158,7 +158,7 @@
   let rec repl =
     function
         [] -> failwith "replace"
-      | d'::ds ->
+      | d'::ds ->
           if d.d_tag = d'.d_tag then d::ds else d' :: repl ds in
   Env (repl b, add_def d m)

@@ -179,7 +179,7 @@
 let addrtype =  mk_type (BasicType AddrType) addr_rep

 let row n t =
-  let r = t.t_rep in
+  let r = t.t_rep in
   mk_type (ArrayType (n, t)) { r_size = n * r.r_size; r_align = r.r_align }

 let discrete t =
@@ -209,15 +209,20 @@
     | ArrayType (n, t1) -> t1
     | _ -> failwith "base_type"

+let array_size t =
+  match t.t_guts with
+      ArrayType (n, t1) -> n
```

```
+    | _-> failwith "not_array"
+
 let get_proc t =
   match t.t_guts with
       ProcType p -> p
     | _ -> failwith "get_proc"

-let rec same_type t1 t2 =
+let rec same_type t1 t2 =
   match (t1.t_guts, t2.t_guts) with
       (ProcType p1, ProcType p2) ->
-        match_args p1.p_fparams p2.p_fparams
+        match_args p1.p_fparams p2.p_fparams
         && same_type p1.p_result p2.p_result
     | (ArrayType (n1, u1), ArrayType(n2, u2)) ->
         n1 = n2 && same_type u1 u2
@@ -225,7 +230,7 @@
     | (BasicType x, PointerType _) -> x = AddrType
     | (_, _) -> t1.t_id = t2.t_id

-and match_args fp1 fp2 =
+and match_args fp1 fp2 =
   match (fp1, fp2) with
       ([], []) -> true
     | (f1::fp1', f2::fp2') ->
diff -r 5e3fa9d71839 lab4/lexer.mll
--- a/lab4/lexer.mll        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/lexer.mll        Thu Jan 30 03:59:11 2020 +0000
@@ -11,10 +11,10 @@

 let lineno = ref 1                      (* Current line in input file *)

-let symtable =
+let symtable =
   Util.make_hash 100
     [ ("array", ARRAY); ("begin", BEGIN);
-      ("const", CONST); ("do", DO); ("if", IF ); ("else", ELSE);
+      ("const", CONST); ("do", DO); ("if", IF ); ("else", ELSE);
       ("end", END); ("of", OF); ("proc", PROC); ("record", RECORD);
       ("return", RETURN); ("then", THEN); ("to", TO);
       ("type", TYPE); ("var", VAR); ("while", WHILE);
@@ -68,14 +68,14 @@
 let notq = [^'\'']
 let notqq = [^'"']

-rule token =
```

```
+rule token =
   parse
       letter (letter | digit | '_')* as s
                          { lookup s }
     | digit+ as s        { NUMBER (int_of_string s) }
     | q (notq as c) q    { CHAR c }
     | q q q q            { CHAR '\'' }
-    | qq ((notqq | qq qq)* as s) qq
+    | qq ((notqq | qq qq)* as s) qq
                          { get_string s }
     | ";"                { SEMI }
     | "."                { DOT }
@@ -102,14 +102,13 @@
     | "\r"               { token lexbuf }
     | "\n"               { next_line lexbuf; token lexbuf }
     | _                  { BADTOK }
-    | eof                { err_message "unexpected end of file" [] !lineno;
+    | eof                { err_message "unexpected end of file" [] !lineno;
                            exit 1 }

-and comment =
+and comment =
   parse
       "*)"               { () }
     | "\n"               { next_line lexbuf; comment lexbuf }
     | _                  { comment lexbuf }
-    | eof                { err_message "end of file in comment" [] !lineno;
+    | eof                { err_message "end of file in comment" [] !lineno;
                            exit 1 }
-
diff -r 5e3fa9d71839 lab4/parser.mly
--- a/lab4/parser.mly        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/parser.mly        Thu Jan 30 03:59:11 2020 +0000
@@ -9,7 +9,7 @@

 %token <Dict.ident>     IDENT
 %token <Optree.op>      MULOP ADDOP RELOP
-%token <int>            NUMBER
+%token <int>            NUMBER
 %token <char>           CHAR
 %token <Optree.symbol * int> STRING

@@ -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 IN

 %type <Tree.program>    program
 %start                  program
@@ -33,17 +33,17 @@

 %%

-program :
+program :
     block DOT                          { Prog ($1, ref []) } ;

-block :
+block :
     decl_list BEGIN stmts END          { makeBlock ($1, $3) } ;

-decl_list :
+decl_list :
     /* empty */                        { [] }
   | decl decl_list                     { $1 @ $2 } ;

-decl :
+decl :
     CONST const_decls                  { $2 }
   | VAR var_decls                      { $2 }
   | proc_decl                          { [$1] }
@@ -60,7 +60,7 @@
     type_decl                          { [$1] }
   | type_decl type_decls               { $1 :: $2 } ;

-type_decl :
+type_decl :
     IDENT EQUAL typexpr SEMI           { ($1, $3) } ;

 var_decls :
@@ -73,18 +73,18 @@
 proc_decl :
     proc_heading SEMI block SEMI       { ProcDecl ($1, $3) } ;

-proc_heading :
+proc_heading :
     PROC name params return_type       { Heading ($2, $3, $4) } ;

 params :
     LPAR RPAR                          { [] }
```

```
   | LPAR formal_decls RPAR              { $2 } ;

-formal_decls :
+formal_decls :
     formal_decl                         { [$1] }
   | formal_decl SEMI formal_decls       { $1 :: $3 } ;

-formal_decl :
+formal_decl :
     ident_list COLON typexpr            { VarDecl (CParamDef, $1, $3) }
   | VAR ident_list COLON typexpr        { VarDecl (VParamDef, $2, $4) }
   | proc_heading                        { PParamDecl $1 } ;
@@ -93,14 +93,14 @@
     /* empty */                         { None }
   | COLON typexpr                       { Some $2 } ;

-stmts :
+stmts :
     stmt_list                           { seq $1 } ;

 stmt_list :
     stmt                                { [$1] }
   | stmt SEMI stmt_list                 { $1 :: $3 } ;

-stmt :
+stmt :
     line stmt1                          { makeStmt ($2, $1) }
   | /* A trick to force the right line number */
     IMPOSSIBLE                          { failwith "impossible" } ;
@@ -114,12 +114,13 @@
   | name actuals                        { ProcCall ($1, $2) }
   | RETURN expr_opt                     { Return $2 }
   | IF expr THEN stmts elses END        { IfStmt ($2, $4, $5) }
-  | WHILE expr DO stmts END             { WhileStmt ($2, $4) }
+  | WHILE expr DO stmts END             { WhileStmt ($2, $4) }
   | REPEAT stmts UNTIL expr             { RepeatStmt ($2, $4) }
-  | FOR name ASSIGN expr TO expr DO stmts END
+  | FOR name ASSIGN expr TO expr DO stmts END
                                         { let v = makeExpr (Variable $2) in
-                                          ForStmt (v, $4, $6, $8, ref None) }
-  | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) } ;
+                                          ForStmt (v, $4, $6, $8, ref None) }
+  | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) }
+  | SUB stmts COLON stmts BUS           { CommuteStmt ($2, $4) } ;

 elses :
```

```
     /* empty */                        { makeStmt (Skip, 0) }
@@ -137,11 +138,11 @@
     /* empty */                        { makeStmt (Skip, 0) }
   | ELSE stmts                         { $2 } ;

-ident_list :
+ident_list :
     IDENT                              { [$1] }
   | IDENT COMMA ident_list             { $1 :: $3 } ;

-expr_opt :
+expr_opt :
     /* empty */                        { None }
   | expr                               { Some $1 } ;

@@ -171,21 +172,21 @@
   | MINUS factor                       { makeExpr (Monop (Uminus, $2)) }
   | LPAR expr RPAR                     { $2 } ;

-actuals :
+actuals :
     LPAR RPAR                          { [] }
   | LPAR expr_list RPAR                { $2 } ;

-expr_list :
+expr_list :
     expr                               { [$1] }
   | expr COMMA expr_list               { $1 :: $3 } ;

-variable :
+variable :
     name                               { makeExpr (Variable $1) }
   | variable SUB expr BUS              { makeExpr (Sub ($1, $3)) }
   | variable DOT name                  { makeExpr (Select ($1, $3)) }
   | variable ARROW                     { makeExpr (Deref $1) } ;

-typexpr :
+typexpr :
     name                               { TypeName $1 }
   | ARRAY expr OF typexpr              { Array ($2, $4) }
   | RECORD fields END                  { Record $2 }
@@ -195,12 +196,12 @@
     field_decl opt_semi                { [$1] }
   | field_decl SEMI fields             { $1 :: $3 } ;

-field_decl :
```

```
+field_decl :
     ident_list COLON typexpr            { VarDecl (FieldDef, $1, $3) } ;

 opt_semi :
     SEMI                                { () }
   | /* empty */                         { () } ;

-name :
+name :
     IDENT                               { makeName ($1, !Lexer.lineno) } ;
diff -r 5e3fa9d71839 lab4/tgen.ml
--- a/lab4/tgen.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/tgen.ml        Thu Jan 30 03:59:11 2020 +0000
@@ -7,6 +7,7 @@
 open Optree
 open Lexer
 open Print
+open Check

 let boundchk = ref false
 let optlevel = ref 0
@@ -51,11 +52,11 @@
   match d.d_addr with
       Global g ->
         <GLOBAL g>
-    | Local off ->
+    | Local off ->
         <OFFSET, schain (!level - d.d_level), <CONST off>>
     | Register i ->
         <REGVAR i>
-    | Nowhere ->
+    | Nowhere ->
         failwith (sprintf "address $" [fId d.d_tag])

 (* |gen_closure| -- two trees for a (code, envt) pair *)
@@ -84,7 +85,7 @@
   libcall "memcpy" [dst; src; <CONST n>] voidtype

 (* |gen_addr| -- code for the address of a variable *)
-let rec gen_addr v =
+let rec gen_addr v =
   match v.e_guts with
       Variable x ->
         let d = get_def x in
@@ -95,19 +96,19 @@
             | VParamDef ->
```

```
                <LOADW, address d>
            | CParamDef ->
-               if scalar d.d_type || is_pointer d.d_type then
+               if scalar d.d_type || is_pointer d.d_type then
                  address d
                else
                  <LOADW, address d>
            | StringDef ->
                address d
-           | _ ->
+           | _ ->
                failwith "load_addr"
        end
    | Sub (a, i) ->
        let bound_check t =
          if not !boundchk then t else <BOUND, t, <CONST (bound a.e_type)>> in
-       <OFFSET,
+       <OFFSET,
          gen_addr a,
          <BINOP Times, bound_check (gen_expr i), <CONST (size_of v.e_type)>>>
    | Select (r, x) ->
@@ -123,9 +124,9 @@
 (* |gen_expr| -- tree for the value of an expression *)
 and gen_expr e =
   match e.e_value with
-      Some v ->
+      Some v ->
         <CONST v>
-    | None ->
+    | None ->
         begin
           match e.e_guts with
               Variable _ | Sub _ | Select _ | Deref _ ->
@@ -139,7 +140,7 @@
                 libcall "int_mod" [gen_expr e1; gen_expr e2] integer
             | Binop (w, e1, e2) ->
                 <BINOP w, gen_expr e1, gen_expr e2>
-           | FuncCall (p, args) ->
+           | FuncCall (p, args) ->
                gen_call p args
            | _ -> failwith "gen_expr"
        end
@@ -157,21 +158,21 @@
        <CALL p.p_pcount, @(fn :: <STATLINK, sl> :: numargs 0 args)>

 (* |gen_arg| -- generate code for a procedure argument *)
```

```
-and gen_arg f a =
+and gen_arg f a =
   match f.d_kind with
       CParamDef ->
-        if scalar f.d_type || is_pointer f.d_type then
+        if scalar f.d_type || is_pointer f.d_type then
           [gen_expr a]
-        else
+        else
           [gen_addr a]
     | VParamDef ->
         [gen_addr a]
     | PParamDef ->
         begin
-          match a.e_guts with
-              Variable x ->
+          match a.e_guts with
+              Variable x ->
                 let (fn, sl) = gen_closure (get_def x) in [fn; sl]
-            | _ ->
+            | _ ->
                 failwith "bad funarg"
         end
     | _ -> failwith "bad arg"
@@ -243,14 +244,14 @@
     let rec tab u qs =
       match qs with
           [] -> []
-        | (v, l) :: rs ->
+        | (v, l) :: rs ->
             if u = v then l :: tab (v+1) rs else deflab :: tab (u+1) qs in
     <JCASE (tab lobound table, deflab),
       <BINOP Minus, sel, <CONST lobound>>>
   end

 (* |gen_stmt| -- generate code for a statement *)
-let rec gen_stmt s =
+let rec gen_stmt s =
   let code =
     match s.s_guts with
         Skip -> <NOP>
@@ -302,7 +303,7 @@
           let l1 = label () and l2 = label () in
           <SEQ,
             <LABEL l1>,
-            gen_stmt body,
```

```
+            gen_stmt body,
             gen_cond test l2 l1,
             <LABEL l2>>

@@ -321,6 +322,11 @@
             <JUMP l1>,
             <LABEL l2>>

+      | CommuteStmt (body1, body2) ->
+          <SEQ,
+            gen_stmt body1,
+            gen_stmt body2>
+
       | CaseStmt (sel, arms, deflt) ->
           (* Use one jump table, and hope it is reasonably compact *)
           let deflab = label () and donelab = label () in
@@ -348,8 +354,8 @@
     function
         <CALL n, @args> ->
           let t = Regs.new_temp 1 in
-          <AFTER,
-            <DEFTEMP t, <CALL n, @(List.map do_tree args)>>,
+          <AFTER,
+            <DEFTEMP t, <CALL n, @(List.map do_tree args)>>,
             <TEMP t>>
       | <w, @args> ->
           <w, @(List.map do_tree args)> in
@@ -369,13 +375,13 @@
 let do_proc lab lev nargs (Block (_, body, fsize, nregv)) =
   level := lev+1;
   retlab := label ();
-  let code0 =
+  let code0 =
     show "Initial code" (Optree.canon <SEQ, gen_stmt body, <LABEL !retlab>>) in
   Regs.init ();
   let code1 = if !optlevel < 1 then code0 else
       show "After simplification" (Jumpopt.optimise (Simp.optimise code0)) in
-  let code2 = if !optlevel < 2 then
-      show "After unnesting" (unnest code1)
+  let code2 = if !optlevel < 2 then
+      show "After unnesting" (unnest code1)
     else
       show "After sharing" (Share.traverse code1) in
   Tran.translate lab nargs !fsize !nregv (flatten code2)
@@ -387,7 +393,7 @@
 let get_decls (Block (decls, _, _, _)) = decls
```

```
 (* |gen_proc| -- translate a procedure, ignore other declarations *)
-let rec gen_proc =
+let rec gen_proc =
   function
       ProcDecl (Heading (x, _, _), block) ->
         let d = get_def x in
@@ -416,4 +422,3 @@
   List.iter gen_global !glodefs;
   List.iter (fun (lab, s) -> Target.emit_string lab s) (string_table ());
   Target.postamble ()
-
diff -r 5e3fa9d71839 lab4/tree.ml
--- a/lab4/tree.ml        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/tree.ml        Thu Jan 30 03:59:11 2020 +0000
@@ -5,7 +5,7 @@
 open Print

 (* |name| -- type for applied occurrences, with mutable annotations *)
-type name =
+type name =
   { x_name: ident;              (* Name of the reference *)
     x_line: int;                (* Line number *)
     mutable x_def: def option } (* Definition in scope *)
@@ -15,56 +15,57 @@

 and block = Block of decl list * stmt * int ref * int ref

-and decl =
+and decl =
     ConstDecl of ident * expr
   | VarDecl of def_kind * ident list * typexpr
   | TypeDecl of (ident * typexpr) list
   | ProcDecl of proc_heading * block
   | PParamDecl of proc_heading

-and proc_heading = Heading of name * decl list * typexpr option
+and proc_heading = Heading of name * decl list * typexpr option

-and stmt =
+and stmt =
   { s_guts: stmt_guts;
     s_line: int }

 and stmt_guts =
-    Skip
```

```
+    Skip
   | Seq of stmt list
   | Assign of expr * expr
-  | ProcCall of name * expr list
+  | ProcCall of name * expr list
   | 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
+  | CommuteStmt of stmt * stmt

-and expr =
-  { e_guts: expr_guts;
-    mutable e_type: ptype;
+and expr =
+  { e_guts: expr_guts;
+    mutable e_type: ptype;
     mutable e_value: int option }

 and expr_guts =
     Constant of int * ptype
   | Variable of name
-  | Sub of expr * expr
+  | Sub of expr * expr
   | Select of expr * name
   | Deref of expr
   | String of Optree.symbol * int
   | Nil
   | FuncCall of name * expr list
-  | Monop of Optree.op * expr
+  | Monop of Optree.op * expr
   | Binop of Optree.op * expr * expr

-and typexpr =
-    TypeName of name
+and typexpr =
+    TypeName of name
   | Array of expr * typexpr
   | Record of decl list
   | Pointer of typexpr

 (* |makeExpr| -- construct an expression node with dummy annotations *)
-let makeExpr e =
+let makeExpr e =
```

```
   { e_guts = e; e_type = voidtype; e_value = None }

 (* |makeStmt| -- construct a stmt node *)
@@ -80,8 +81,8 @@
     | ss -> makeStmt (Seq ss, 0)

 let get_def x =
-  match x.x_def with
-      Some d -> d
+  match x.x_def with
+      Some d -> d
     | None -> failwith (sprintf "missing def of $" [fId x.x_name])

 (* |MakeBlock| -- construct a block node with dummy annotations *)
@@ -90,7 +91,7 @@

 (* Grinder *)

-let fTail f xs =
+let fTail f xs =
   let g prf = List.iter (fun x -> prf " $" [f x]) xs in fExt g

 let fList f =
@@ -105,11 +106,11 @@
       [] -> fMeta "(BLOCK $)" [fStmt stmts]
     | _ -> fMeta "(BLOCK (DECLS$) $)" [fTail(fDecl) decls; fStmt stmts]

-and fDecl =
+and fDecl =
   function
-      ConstDecl (x, e) ->
+      ConstDecl (x, e) ->
         fMeta "(CONST $ $)" [fId x; fExpr e]
-    | VarDecl (kind, xs, te) ->
+    | VarDecl (kind, xs, te) ->
         fMeta "($ $ $)" [fKind kind; fList(fId) xs; fType te]
     | TypeDecl tds ->
         let f (x, te) = fMeta "($ $)" [fId x; fType te] in
@@ -131,7 +132,7 @@
   let res = match te with Some t -> fType t | None -> fStr "VOID" in
   fMeta "($ $ $)" [fName p; fList(fDecl) fps; res]

-and fStmt s =
+and fStmt s =
   match s.s_guts with
       Skip -> fStr "(SKIP)"
```

```
     | Seq stmts -> fMeta "(SEQ$)" [fTail(fStmt) stmts]
@@ -139,7 +140,7 @@
     | ProcCall (p, aps) -> fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
     | Return (Some e) -> fMeta "(RETURN $)" [fExpr e]
     | Return None -> fStr "(RETURN)"
-    | IfStmt (test, thenpt, elsept) ->
+    | IfStmt (test, thenpt, elsept) ->
         fMeta "(IF $ $ $)" [fExpr test; fStmt thenpt; fStmt elsept]
     | WhileStmt (test, body) ->
         fMeta "(WHILE $ $)" [fExpr test; fStmt body]
@@ -162,9 +163,9 @@
     | Nil -> fStr "(NIL)"
     | FuncCall (p, aps) ->
         fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
-    | Monop (w, e1) ->
+    | Monop (w, e1) ->
         fMeta "($ $)" [Optree.fOp w; fExpr e1]
-    | Binop (w, e1, e2) ->
+    | Binop (w, e1, e2) ->
         fMeta "($ $ $)" [Optree.fOp w; fExpr e1; fExpr e2]

 and fType =
@@ -174,5 +175,5 @@
     | Record fields -> fMeta "(RECORD$)" [fTail(fDecl) fields]
     | Pointer t1 -> fMeta "(POINTER $)" [fType t1]

-let print_tree fp pfx (Prog (body, _)) =
+let print_tree fp pfx (Prog (body, _)) =
   fgrindf fp pfx "(PROGRAM $)" [fBlock body]
diff -r 5e3fa9d71839 lab4/tree.mli
--- a/lab4/tree.mli        Wed Nov 27 13:33:20 2019 +0000
+++ b/lab4/tree.mli        Thu Jan 30 03:59:11 2020 +0000
@@ -18,7 +18,7 @@
 *)

 (* |name| -- type for applied occurrences, with mutable annotations *)
-type name =
+type name =
   { x_name: ident;              (* Name of the reference *)
     x_line: int;                (* Line number *)
     mutable x_def: def option } (* Definition in scope *)
@@ -30,50 +30,51 @@

 and block = Block of decl list * stmt * int ref * int ref

-and decl =
```

```
+and decl =
      ConstDecl of ident * expr
    | VarDecl of def_kind * ident list * typexpr
    | TypeDecl of (ident * typexpr) list
    | ProcDecl of proc_heading * block
    | PParamDecl of proc_heading

-and proc_heading = Heading of name * decl list * typexpr option
+and proc_heading = Heading of name * decl list * typexpr option

-and stmt =
+and stmt =
    { s_guts: stmt_guts;
      s_line: int }

 and stmt_guts =
-    Skip
+    Skip
    | Seq of stmt list
    | Assign of expr * expr
-   | ProcCall of name * expr list
+   | ProcCall of name * expr list
    | 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
+   | CommuteStmt of stmt * stmt

-and expr =
-   { e_guts: expr_guts;
-     mutable e_type: ptype;
+and expr =
+   { e_guts: expr_guts;
+     mutable e_type: ptype;
      mutable e_value: int option }

 and expr_guts =
      Constant of int * ptype
    | Variable of name
-   | Sub of expr * expr
+   | Sub of expr * expr
    | Select of expr * name
    | Deref of expr
    | String of Optree.symbol * int
```

```
    | Nil
    | FuncCall of name * expr list
-   | Monop of Optree.op * expr
+   | Monop of Optree.op * expr
    | Binop of Optree.op * expr * expr

-and typexpr =
-    TypeName of name
+and typexpr =
+    TypeName of name
    | Array of expr * typexpr
    | Record of decl list
    | Pointer of typexpr
```

6 Tests For Task 2

6.1

```
var x,y: integer;
begin
[ x:=1 : y:=2 ];
print_num(x); print_num(y)
end.

(*<<
Correct
>>*)
```

6.2

```
var x: integer;
begin
[ x:=1 : x:=2 ];
(* swapping would lead to a different answer *)
print_num(x);
end.


(*<<
Possibly incorrect
>>*)
```

6.3

```
var x, y: integer;
begin
```

```
[ x:=1; x:=1: x:=2 ];
(* swapping would lead to a different answer *)
print_num(x);
end.


(*<<
Possibly incorrect
>>*)
```

6.4

```
var x, y, i: integer;
begin
[ x:=1; x:=1:
  for x := 1 to 2 do
    i := i+1
  end
 ];
print_num(x);
end.


(*<<
Possibly incorrect
>>*)
```

6.5

```
var x, y,z,i: integer;
begin
[ x := 3 :
  for i := 1 to x do
    y := 3
  end
 ];
print_num(x);
end.


(*<<
Possibly incorrect
>>*)
```

6.6

```
var x, y,z,i: integer;
begin
[ x := 3 :
  for x := 1 to x do
    y := 3
  end
 ];
print_num(x);
end.


(*<<
Possibly incorrect
>>*)
```

6.7

```
var x, y, z : integer;
begin
  x := 3;
  y := 4;
  z := 5;
  [ y := x : z := x];
  print_num(y);
end.

(*<<
Correct
>>*)
```

6.8

```
var x, y, z, t, i, j : integer;
begin
  x:=1;
  y:=1;
  z:=1;
 [if x=3 then z:=2 else t:=3; end :
  x:=15;
  for i := 1 to 10 do
    for j := 1 to 12 do
      x:=x+1;
    end;
  end;
```

```
];
  x:=1;
  y:=1;
  z:=1;
  t:=1;
end.

(*<<
Possibly incorrect
>>*)
```

6.9

```
var a, b, c, d : integer;
begin
  a := 1;
  b := 1;
  while (b <> 2) do
    b := b+1
  end;
  [a := 3 :
  b := 4;
  while (b > 2) do
  b := b +1;
  b := b -2
  end;
  ]
end.


(*<<
Correct
>>*)
```

6.10

```
var x, y, z, t : integer;
begin
[x:=1; y:=1 :
 z:=1; t:=1];

[x:=1; z:=1 :
  t:=1; y:=1]
end.
```

```
(*<<
Correct
>>*)
```

6.11

```
var x, y, z, t : integer;
begin
[x:=1; [x:=2 : y:=3] : t:=x ]
end.


(*<<
Possibly incorrect
>>*)
```

6.12

```
var x, y, z, t : integer;
begin
[x:=1; [x:=2 : z:=y] : t:=y ]
end.


(*<<
Correct
>>*)
```

6.13

```
type point = record x,y :integer end;
var a, b: point;

begin
[a.x := 3; a.y := 5 :
 b.y := 4; b.x := 4]
end.

(*<<
Correct
>>*)
```

6.14

```
type point = record x,y :integer end;
var a, b: point;
```

```
begin
[a.x := 3 :
 a.y := 4]
end.

(*<<
Possibly incorrect
>>*)
```

6.15

```
var a : array 10 of integer;

begin
[a[1] := 3 :
 a[2] := 4]
end.

(*<<
Possibly incorrect
>>*)
```

6.16

```
var a, b, c : integer;
begin
a := 2;
[ case a of  1 : b:=3
            |2 : c:=3
            end
: a:=3]
end.


(*<<
Correct
>>*)
```

6.17

```
var a, b, c : integer;
begin
a := 2;
[ case a of  1 : b:=3
            |2 : c:=3
```

```
            end
: b:=3]
end.


(*<<
Possibly incorrect
>>*)
```

6.18

```
var x, y, z : integer;
begin
[x := y : z:= y]
end.


(*<<
Correct
>>*)
```

6.19

```
var x, y, z : integer;
begin
[x := y + z : y:= 3]
end.


(*<<
Possibly incorrect
>>*)
```