Implementing call-by-name

Mike Spivey, December 2018

This document sets out sample solutions to the two problems contained in the Christmas Assignment for the Compilers course in 2018. The first problem, to implement an equivalent of the C `continue` statement, is quite simple to solve.

The second problem is to implement parameters passed by name; this is harder. The suggested technique is to transform the program after semantic analysis so as to replace name parameters with local procedures passed as procedural parameters. This does not really work, because the semantic analysis phase must not only identify name parameters, but also analyse them carefully, ensuring that any local variables mentioned in an actual parameter passed by name are addressable and will not become register variables. Once semantic analysis is over, it would still be possible to rewrite the tree as suggested, but this would require a separate scan of the entire AST to pull out name parameters, and it is more straight-forward just to generate code from the tree directly.

Two restrictions are suggested in the assignment: to allow only integer-valued parameters to be passed by name, and to forbid procedural parameters that themselves take a parameter passed by name. These are shown here to be unnecessary; we can allow name parameters of any scalar type, and procedural parameters create no difficulty at all.

1 Implementing continue

The implementation of the `continue` statement is very straightforward. We must add a new keyword the the lexer, and a new parameterless constructor to the type of abstract syntax trees for statements. There is a single production added to the parser.

```
stmt : ...
  | CONTINUE                    { Continue }
```

Both the semantic analyser and the intermediate code generator must be extended to accept the new statement. The `continue` statement will be implemented by an unconditional jump to an appropriately placed label, and it's simplest to delay allocation of the label until code generation.

All that needs to be done in the semantic analyser is to check that each

`continue` statement appears inside some kind of loop, and for that it's convenient to add a boolean parameter *inloop* to the recursive function *check_stmt*.

In the code generator, the function *gen_stmt* also acquires an additional parameter, in this case a label *contlab* to which a `continue` statement jumps. This label can have the dummy value *nolab* initially, and in each of the three kinds of loop statement, we make sure to place a label just after the loop body and pass that label to the recursive call of *gen_stmt* that generates code for the body.

All details of the changes needed are contained in the diff listings at the end of these answers. At least one test case for each kind of loop would be desirable, but they are omitted here for lack of time.

2 Name parameters: the idea

Name parameters are equivalent to parameterless local functions (sometimes called 'thunks'). For example, in this test case,

```
proc m();
  var g: integer;

  proc p(name x: integer): integer;
  begin
    g := g+1;
    return x+x
  end;

begin
  g := 0;
  println(p(2+3));
  println(p(g));
  println(p(p(7)));
  println(g)
end;
```

we can replace the parameters to each call of `p` with functions local to `m`:

```
proc m();
  var g: integer;

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

  proc thunk1(): integer; begin return 2+3 end;

  proc thunk2(): integer; begin return g end;

  proc thunk3a(): integer; begin return 7 end;

  proc thunk3b(): integer; begin return p(thunk3a) end;

begin
  g := 0;
```

```
  println(p(thunk1));
  println(p(thunk2));
  println(p(thunk3b))
end;
```

Note that nesting the thunks inside procedure `m` gives them access to its local variable `g`; also, though the two calls of `p` are nested in the third call of `println`, there is no need to nest the two resulting thunks, and it is less efficient to do so, since it makes the static chains longer.

It might be possible to implement this idea by introducing a transformation pass between semantic analysis and code generation, but that creates problems with the sequencing of calculations. We need to have carried out semantic analysis in order to identify which procedure calls involve name parameters; but on the other hand, we need to analyse the actual name parameters like `2+3` and `g` as if they were embedded in the bodies of local procedures like `thunk1` and `thunk2`. This particularly affects references to local variables, because the semantic analyser needs to ensure those variables do not live in registers but in the stack frame. So the semantic analyser does need to treat name parameters in a significantly different way.

In the solution that follows, a different approach is taken: instead of transforming the abstract syntax tree, both semantic analysis and intermediate code generation are changed to include name parameters. The code that is output does, however, closely resemble what would result from the transformation just described.

3 The easy bits

The new keyword `name` needs to be added to the lexer's table.

```
let symtable =
  Util.make_hash 100 [ ...; ("name", NAME); ... ]
```

A new kind of parameter declaration needs to be added to the parser.

```
%token NAME

...

formal_decl : ...
  | NAME ident_list COLON typexpr      { VarDecl (NParamDef, $2, $4) }
```

Also easy is to add a new *defkind* value to the dictionary module dict.mli.

```
type def_kind = ...
  | NParamDef                  (* Name parameter *)
```

with a similar change in dict.ml.

4 Semantic analysis

Changes to the semantic analyser are in three groups: we must make it possible to declare name parameters; we must enable them to be used in procedure bodies; and we must implement passing an expression as a name parameter.

For parameter declarations, we need to add *NParamDef* to the kinds of definition that can be accepted by *param_alloc*. Since the idea is that a name parameter will, like a procedural parameter, be represented by a closure, we can re-use an existing rule.

```
let param_alloc pcount d =
  let s = param_rep.r_size in
  match d.d_kind with ...
    | NParamDef | PParamDef →
      d.d_addr ← Local (param_base + s ∗ !pcount);
      pcount := !pcount + 2
```

Then, in *do_alloc*, declarations with *NParamDef* must be added to those that need to be passed to the storage allocation routine.

```
let do_alloc alloc ds =
  let h d =
    match d.d_kind with
        VarDef | CParamDef | VParamDef | FieldDef
          | NParamDef | PParamDef → alloc d
      | _ → () in
  List.iter h ds
```

Next, in *check_decl*, we implement the check that name parameters must have scalar types (generalising the requirement that integers be allowed).

```
let check_decl d env =
  (∗ All types of declaration are mixed together in the AST ∗)
  match d with ...
    | VarDecl (kind, xs, te) →
        let t = check_typexpr te env in
        let def x env =
          let d = make_def x kind t in
          add_def d env in
        if kind = NParamDef && not (scalar t) then
          sem_error "Only scalar name parameters are allowed" [ ];
        Util.accum def xs env
```

Now we can turn to using name parameters in procedure bodies. We add *NParamDef* to the list of kinds of definition which (unlike, say, a procedure name) denote values that can be used in expressions.

```
let has_value d =
  match d.d_kind with
      ConstDef _ | VarDef | CParamDef | VParamDef
        | NParamDef | StringDef → true
    | _ → false
```

That's all the change needed here for uses of these parameters, though more remains to be revealed when we come to code generation later.

The remaining part is what to do with procedure calls that involve name parameters. For these, we need to add to the function *check_arg* that matches up formal and actual parameters.

```
let check_arg formal arg env =
```

```
match formal.d_kind with ...
  | NParamDef →
      incr level;
      let t1 = check_expr arg env in
      if not (same_type formal.d_type t1) then
        sem_error "argument has wrong type" [ ];
      decr level
```

There's a subtlety here concerning the use of local variables in name parameters (see the test case locname.p). If a local is going to be addressable from the thunk that is created for a name parameter, the storage allocator must be prevented from making it into a register variable, just as if it were used in a procedure nested inside the current one. The code that checks expressions achieves this by setting the *d_mem* flag on variables that must be addressable, and it does this when a local variable is used that is not at the same nesting level as the expression being analysed. The upshot is that we must increment the global variable *level* during the time the parameter expression is being analysed. Nested calls to procedures with name parameters will cause the level to be incremented multiple times; this is wrong because actually all the expressions are at the same level, one greater than the enclosing procedure, but the extra increments and subsequent decrements do no harm.

5 Code generation

To generate code for procedure calls with name parameters, we'll need to generate and emit little procedures for the thunks. It's convenient for this if we separate the process of feeding code for a procedure body to the back end from the process of generating operator trees from the abstract syntax tree. Here is a subroutine, extracted from *do_proc*, that implements the feeding of the back end.

```
let out_proc lab nargs fsize nregv code =
  let code0 = show "Initial code" (Optree.canon code) 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)
    else
      show "After sharing" (Share.traverse code1) in
  Tran.translate lab nargs fsize nregv (flatten code2)
```

This subroutine is called from a simplified version of *do_proc*, but it can also be called with code that is to be wrapped up as a thunk body. What remains of *do_proc* is much simpler than before.

```
let do_proc lab lev nargs (Block (_, body, fsize, nregv)) =
  proc_level := lev + 1;
  level := !proc_level;
  retlab := label ( );
  out_proc lab nargs !fsize !nregv
```

⟨*SEQ*, *gen_stmt body*, ⟨*LABEL* !*retlab*⟩⟩

We'll return in a moment to the change from a single *level* variable to the pair (*proc_level*, *level*).

Next, let's look at the process of calling a procedure with a name parameter. The idea is that the parameter expression becomes a little procedure nested inside the current procedure; these 'thunks' are created by the helper function *gen_thunk*, which calls the *out_proc* function to emit the procedure body.

```
let gen_thunk e =
  let lev = !level in
  level := !proc_level + 1;
  let lab = gensym ( ) in
  out_proc lab 0 0 0 ⟨RESULTW, gen_expr e⟩;
  level := lev;
  lab
```

Again, there is a subtlety about nesting levels here. We need to generate code for the parameter expression as if it were nested inside an internal procedure; since the code for local variable references depends on the difference in levels between the expression and the variable (giving the number of static links to traverse), we must make the apparent level one greater while translating the expression. Note that the value must in this case be exact, and multiple increments and decrements for nested procedure calls (within the recursive call *gen_expr e*) won't give us what we need. Accordingly, the old global variable *level* is split into two variables *proc_level* and *level*, with *level* always equal to *proc_level* or *proc_level* + 1. The value of *level* is carefully saved and restored by *gen_thunk*.

Here's the declaration of the new variables, replacing the single, old variable.

```
(* proc_level – nesting level of current procedure *)
let proc_level = ref 0

(* level – nesting level of expression *)
let level = ref 0
```

Now we can look at the code in *gen_arg* that generates code to pass a name parameter.

```
let gen_arg f a =
  match f.d_kind with ...
    | NParamDef →
        let thunk = gen_thunk a in
        [⟨GLOBAL thunk⟩; schain (!level − !proc_level)]
```

There are two words of parameter passed, making up a closure with a thunk body generated from the parameter expression, and the address of the current procedure's frame as a static link. Note that it's not correct to write just ⟨*LOCAL* 0⟩ for the static link, because that doesn't cater properly for nested calls, as in the jensmat.p test case (see Section 6).

When a name parameter is used, we must call the thunk as a parameterless function. Unusually, a reference to a name parameter is syntactically a variable, but it does not have an address to load from. So we must separate

out simple variables from other kinds of references, and for the sake of simplicity repeat a line or two of code. (We omit the preamble that deals with expressions that have been reduced to constants with *e.e_value* set.)

```
let gen_expr e = ...
  match e.e_guts with
      Variable x →
        let d = get_def x in
        if d.d_kind = NParamDef then begin
          let (fn, sl) = gen_closure d in
          ⟨CALL 0, fn, ⟨STATLINK, sl⟩⟩
        end else begin
          let ld =
            if size_of e.e_type = 1 then LOADC else LOADW in
          ⟨ld, gen_addr e⟩
        end

    | Sub _ | Select _ | Deref _ →
        let ld =
          if size_of e.e_type = 1 then LOADC else LOADW in
        ⟨ld, gen_addr e⟩
```

If a variable reference is to a name parameter, we fetch the closure for the parameter and call it, passing no parameters, and use the result as the value of the parameter.

Fetching the closure for a name parameter is the same as for a procedural parameter.

```
let gen_closure d =
  match d.d_kind with ...
    | PParamDef | NParamDef →
        (⟨LOADW, address d⟩,
          ⟨LOADW, ⟨OFFSET, address d, ⟨CONST addr_size⟩⟩⟩)
```

6 Test cases

In addition to the supplied test case, it's worth including one (murawski2.p) where the variable accessed from a name parameter is local to the calling procedure.

```
proc println(x: integer);
begin
  print_num(x);
  newline()
end;

proc m();
  var g: integer;

  proc p(name x: integer): integer;
  begin
    g := g+1;
    return x+x
```

```
  end;

begin
  g := 0;
  println(p(2+3));
  println(p(g));
  println(p(p(7)));
  println(g)
end;

begin
  m()
end.
```

We can write a version of the recursive factorial function (namefac.p) where the conditional is wrapped up in a procedure, using name parameters to delay the evaluation of the then and else parts until their values are needed.

```
proc iff(test: boolean; name thenpt, elsept: integer): integer;
begin
  if test then return thenpt else return elsept end
end;

proc fac(n: integer): integer;
begin
  return iff(n = 0, 1, n * fac(n-1))
end;

begin
  print_num(fac(10));
  newline()
end.
```

Jensen's device of course works with our implementation (jensen.p).

```
proc sum(a, b: integer; var i: integer; name f: integer): integer;
  var s: integer;
begin
  s := 0;
  for i := a to b do s := s + f end;
  return s
end;

proc mysum(n: integer): integer;
  var j: integer;
begin
  return sum(1, n, j, j*j)
end;

begin
  print_num(mysum(10)); newline()
end.
```

It's challenging to support nested applications of Jensen's device, as in the this test case jensmat.p that sums a matrix. The challenge here is to ensure

that, of the two thunks generated from the expression sum(i, sum(j, a[i][j])), the outer one passes the correct static link for the inner one.

```
type matrix = array 3 of array 3 of integer;

proc sum(var i: integer; name f: integer): integer;
  var s: integer;
begin
  s := 0;
  for i := 0 to 2 do s := s + f end;
  return s
end;

proc matsum(var a: matrix): integer;
  var i, j: integer;
begin
    return sum(i, sum(j, a[i][j]))
end;

proc test();
    var a: matrix; i, j: integer;
begin
    for i := 0 to 2 do
        for j := 0 to 2 do
            a[i][j] := (i+1)*(j+1)
        end
    end;

    print_num(matsum(a)); newline()
end;

begin test() end.
```

And now, at last, Knuth's "Man or boy" test comes into its own (namemob.p).

```
proc A(k: integer; name x1, x2, x3, x4, x5 : integer): integer;
  proc B(): integer;
  begin
    k := k-1;
    return A(k, B(), x1, x2, x3, x4)
  end;
begin
  if k <= 0 then return x4 + x5 else return B() end
end;

begin
  print_num(A(10, 1, -1, -1, 1, 0)); newline()
end.
```

The specification states that procedures with name parameters can't be used as procedural parameters. There is no reason for this restriction, and it is not implemented in the solution above. Here is a simple test case nameproc.p that shows that the forbidden feature works without problems.

```
proc p(proc f(name x: integer): integer): integer;
  var u: integer;
```

```
  proc g(): integer;
  begin
    u := u+1;
    return u
  end;
begin
  u := 3;
  return f(g())
end;

proc q(name x: integer): integer;
begin
  return x + x
end;

begin
  print_num(p(q)); newline()
end.
```

Finally, there is a test case locname.p that catches a bug where a local variable used in an actual name parameter is wrongly allowed to be kept in a register. Note that i and j may try to share the first callee-save register, and its value would then be destroyed by the assignment to j before it is fetched as the value of i.

```
proc p(name x: integer);
  var j: integer;
begin
  j := 48;
  print_num(x); newline()
end;

proc q();
  var i: integer;
begin
  i := 37;
  p(i)
end;

begin
  q()
end.
```

```
--- ../../labs/lab4/check.ml    2023-10-08 15:18:06.767585400 +0100
+++ check.ml    2023-10-08 15:30:14.502954460 +0100
@@ -90,7 +90,8 @@
 (* |has_value| -- check if object is suitable for use in expressions *)
 let has_value d =
   match d.d_kind with
-      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
+      ConstDef _ | VarDef | CParamDef | VParamDef
+        | NParamDef | StringDef -> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -198,12 +199,21 @@
 (* |check_arg| -- check one (formal, actual) parameter pair *)
 and check_arg formal arg env =
   match formal.d_kind with
-      CParamDef | VParamDef ->
+      CParamDef ->
+       let t1 = check_expr arg env in
+       if not (same_type formal.d_type t1) then
+         sem_error "argument has wrong type" []
+    | VParamDef ->
+       let t1 = check_expr arg env in
+       if not (same_type formal.d_type t1) then
+         sem_error "argument has wrong type" [];
+        check_var arg true
+    | NParamDef ->
+        incr level;
         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
-          check_var arg true
+        decr level
     | PParamDef ->
         let pf = get_proc formal.d_type in
         let x = (match arg.e_guts with Variable x -> x
@@ -291,13 +301,13 @@
   chk (List.sort compare vs)

 (* |check_stmt| -- check and annotate a statement *)
-let rec check_stmt s env alloc =
+let rec check_stmt s env alloc inloop =
   err_line := s.s_line;
   match s.s_guts with
       Skip -> ()

     | Seq ss ->
-        List.iter (fun s1 -> check_stmt s1 env alloc) ss
+       List.iter (fun s1 -> check_stmt s1 env alloc inloop) ss

     | Assign (lhs, rhs) ->
         let lt = check_expr lhs env
@@ -327,21 +337,25 @@
                   sem_error "function must return a result" []
         end

+    | Continue ->
+        if not inloop then
+          sem_error "continue statement must be inside a loop" []
+
     | 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" [];
-        check_stmt thenpt env alloc;
-        check_stmt elsept env alloc
+       check_stmt thenpt env alloc inloop;
+       check_stmt elsept env alloc inloop

     | WhileStmt (cond, body) ->
         let ct = check_expr cond env in
         if not (same_type ct boolean) then
           sem_error "type mismatch in while statement" [];
-        check_stmt body env alloc
+       check_stmt body env alloc true

     | RepeatStmt (body, test) ->
-        check_stmt body env alloc;
+        check_stmt body env alloc true;
         let ct = check_expr test env in
         if not (same_type ct boolean) then
           sem_error "type mismatch in repeat statement" []
@@ -354,7 +368,7 @@
             || not (same_type hit integer) then
           sem_error "type mismatch in for statement" [];
         check_var var false;
-        check_stmt body env alloc;
+        check_stmt body env alloc true;

         (* Allocate space for hidden variable.  In the code, this will
            be used to save the upper bound.  *)
@@ -370,11 +384,11 @@
           let (t1, v) = check_const lab env in
           if not (same_type t1 st) then
             sem_error "case label has wrong type" [];
-          check_stmt body env alloc; v in
+          check_stmt body env alloc inloop; v in

         let vs = List.map check_arm arms in
         check_dupcases vs;
-        check_stmt deflt env alloc
+       check_stmt deflt env alloc inloop


 (* TYPES AND DECLARATIONS *)
@@ -416,7 +430,7 @@
 let param_size d =
   match d.d_kind with
       CParamDef | VParamDef -> 1
-    | PParamDef -> 2
+    | PParamDef | NParamDef -> 2
     | _ -> failwith "param_size"

 (* param_alloc -- allocate space for formal parameters *)
@@ -434,7 +448,7 @@
 let do_alloc alloc ds =
   let h d =
     match d.d_kind with
-        VarDef | CParamDef | VParamDef | FieldDef | PParamDef ->
+        VarDef | CParamDef | VParamDef | FieldDef | NParamDef | PParamDef ->
           alloc d
       | _ -> () in
   List.iter h ds
```

```
@@ -502,6 +516,8 @@
          let def x env =
            let d = make_def x kind t in
            add_def d env in
+         if kind = NParamDef && not (scalar t) then
+           sem_error "Only scalar name parameters are allowed" [];
          Util.accum def xs env
      | TypeDecl tds ->
          let tds' =
@@ -546,8 +562,7 @@
  (* Storage is allocated for local variables in a separate phase from
     processing the declarations.  That makes it possible to gather
     additional information about the usage of variables (i.e., whether
-    they must be addressible) before deciding which can live in
-    registers. *)
+    they must be addressible) before deciding which can live in registers. *)

  (* |check_block| -- check a local block *)
  let rec check_block level rt env (Block (ds, ss, fsize, nregv)) =
@@ -556,7 +571,7 @@
    let pre_alloc d = defs := !defs @ [d] in
    check_bodies env' ds;
    return_type := rt;
-   check_stmt ss env' pre_alloc;
+   check_stmt ss env' pre_alloc false;
    do_alloc (local_alloc fsize nregv) !defs;
    align max_align fsize

@@ -623,6 +638,6 @@
    return_type := voidtype;
    level := 1;
    let alloc = local_alloc fsize nregv in
-   check_stmt ss env alloc;
+   check_stmt ss env alloc false;
    align max_align fsize;
    glodefs := top_block env
--- ../../labs/lab4/dict.ml     2023-10-08 15:18:06.771585372 +0100
+++ dict.ml     2023-10-08 15:30:14.506954436 +0100
@@ -80,6 +80,7 @@
    | VarDef                     (* Variable *)
    | CParamDef                  (* Value parameter *)
    | VParamDef                  (* Var parameter *)
+   | NParamDef                  (* Name parameter *)
    | FieldDef                   (* Field of record *)
    | ProcDef                    (* Procedure *)
    | PParamDef                  (* Proc parameter *)
--- ../../labs/lab4/dict.mli    2023-10-08 15:18:06.767585400 +0100
+++ dict.mli    2023-10-08 15:30:14.506954436 +0100
@@ -44,6 +44,7 @@
    | VarDef                     (* Variable *)
    | CParamDef                  (* Value parameter *)
    | VParamDef                  (* Var parameter *)
+   | NParamDef                  (* Name parameter *)
    | FieldDef                   (* Field of record *)
    | ProcDef                    (* Procedure *)
    | PParamDef                  (* Proc parameter *)
--- ../../labs/lab4/tgen.ml     2023-10-08 15:18:06.787585264 +0100
+++ tgen.ml     2023-10-08 15:33:37.865720780 +0100
@@ -12,9 +12,52 @@
  let optlevel = ref 0
  let debug = ref 0
```

```
+(* unnest -- move procedure calls to top level *)
+let unnest code =
+  let rec do_tree =
+    function
+        <CALL n, @args> ->
+          let t = Regs.new_temp 1 in
+          <AFTER,
+            <DEFTEMP t, <CALL n, @(List.map do_tree args)>>,
+            <TEMP t>>
+      | <w, @args> ->
+          <w, @(List.map do_tree args)> in
+  let do_root =
+    function <op, @args> -> <op, @(List.map do_tree args)> in
+  Optree.canonicalise <SEQ, @(List.map do_root code)>
+
+let show label code =
+  if !debug > 0 then begin
+    printf "$$:\n" [fStr Mach.comment; fStr label];
+    List.iter (Optree.print_optree Mach.comment) code;
+    printf "\n" []
+  end;
+  code
+
+let out_proc lab nargs fsize nregv code =
+  let code0 = show "Initial code" (Optree.canonicalise code) 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)
+    else
+      show "After sharing" (Share.traverse code1) in
+  Target.start_proc lab nargs fsize;
+  Regs.get_regvars nregv;
+  Tran.translate (flatten code2);
+  Target.end_proc ()
+
 (* |level| -- nesting level of current procedure *)
+let proc_level = ref 0
+
+(* |level| -- nesting level of expression *)
 let level = ref 0

+(* |nlambda| -- count of lambda labels *)
+let nlambda = ref 0
+
 (* |retlab| -- label to return from current procedure *)
 let retlab = ref nolab

@@ -61,7 +104,7 @@
       ProcDef ->
         (<GLOBAL (symbol_of d)>,
           if d.d_level = 0 then <CONST 0> else schain (!level - d.d_level))
-    | PParamDef ->
+    | PParamDef | NParamDef ->
         (<LOADW, address d>,
           <LOADW, <OFFSET, address d, <CONST addr_rep.r_size>>>)
     | _ -> failwith "missing closure"
@@ -125,7 +168,16 @@
     | None ->
         begin
           match e.e_guts with
```

```
-              Variable _ | Sub _ | Select _ | Deref _ ->
+              Variable x ->
+                let d = get_def x in
+                if d.d_kind = NParamDef then begin
+                  let (fn, sl) = gen_closure d in
+                  <CALL 0, fn, <STATLINK, sl>>
+                end else begin
+                  let ld = if size_of e.e_type = 1 then LOADC else LOADW in
+                  <ld, gen_addr e>
+                end
+            | Sub _ | Select _ | Deref _ ->
                 let ld = if size_of e.e_type = 1 then LOADC else LOADW in
                 <ld, gen_addr e>
             | Monop (w, e1) ->
@@ -163,6 +215,9 @@
           [gen_addr a]
     | VParamDef ->
         [gen_addr a]
+    | NParamDef ->
+        let thunk = gen_thunk a in
+        [<GLOBAL thunk>; schain (!level - !proc_level)]
     | PParamDef ->
         begin
           match a.e_guts with
@@ -202,6 +257,17 @@
         let proc = sprintf "$" [fLibId q.q_id] in
         libcall proc (List.map gen_expr args) voidtype

+and gen_thunk e =
+  let lev = !level in
+  level := !proc_level+1;
+  incr nlambda;
+  let lab = sprintf "__lambda$" [fNum !nlambda] in
+  let body = gen_expr e in
+  printf "$Thunk:\n" [fStr Mach.comment];
+  out_proc lab 0 0 0 <RESULTW, body>;
+  level := lev;
+  lab
+
 (* |gen_cond| -- generate code to branch on a condition *)
 let rec gen_cond test tlab flab =
   match test.e_value with
@@ -250,12 +316,12 @@
   end

 (* |gen_stmt| -- generate code for a statement *)
-let rec gen_stmt s =
+let rec gen_stmt s contlab =
   let code =
     match s.s_guts with
         Skip -> <NOP>

-      | Seq ss -> <SEQ, @(List.map gen_stmt ss)>
+      | Seq ss -> <SEQ, @(List.map (fun s -> gen_stmt s contlab) ss)>

       | Assign (v, e) ->
           if scalar v.e_type || is_pointer v.e_type then begin
@@ -275,15 +341,18 @@
               | None -> <JUMP !retlab>
           end

+      | Continue ->
```

```
+           <JUMP contlab>
+
        | IfStmt (test, thenpt, elsept) ->
            let l1 = label () and l2 = label () and l3 = label() in
            <SEQ,
              gen_cond test l1 l2,
              <LABEL l1>,
-             gen_stmt thenpt,
+             gen_stmt thenpt contlab,
              <JUMP l3>,
              <LABEL l2>,
-             gen_stmt elsept,
+             gen_stmt elsept contlab,
              <LABEL l3>>

        | WhileStmt (test, body) ->
@@ -294,15 +363,16 @@
              <LABEL l1>,
              gen_cond test l2 l3,
              <LABEL l2>,
-             gen_stmt body,
+             gen_stmt body l1,
              <JUMP l1>,
              <LABEL l3>>

        | RepeatStmt (body, test) ->
-           let l1 = label () and l2 = label () in
+           let l1 = label () and l2 = label () and l3 = label () in
            <SEQ,
              <LABEL l1>,
-             gen_stmt body,
+             gen_stmt body l3,
+             <LABEL l3>,
              gen_cond test l2 l1,
              <LABEL l2>>

@@ -310,13 +380,14 @@
            (* Use previously allocated temp variable to store upper bound.
               We could avoid this if the upper bound is constant. *)
            let tmp = match !upb with Some d -> d | _ -> failwith "for" in
-           let l1 = label () and l2 = label () in
+           let l1 = label () and l2 = label () and l3 = label () in
            <SEQ,
              <STOREW, gen_expr lo, gen_addr var>,
              <STOREW, gen_expr hi, address tmp>,
              <LABEL l1>,
              <JUMPC (Gt, l2), gen_expr var, <LOADW, address tmp>>,
-             gen_stmt body,
+             gen_stmt body l3,
+             <LABEL l3>,
              <STOREW, <BINOP Plus, gen_expr var, <CONST 1>>, gen_addr var>,
              <JUMP l1>,
              <LABEL l2>>
@@ -330,63 +401,26 @@
            let gen_case lab (v, body) =
              <SEQ,
                <LABEL lab>,
-               gen_stmt body,
+               gen_stmt body contlab,
                <JUMP donelab>> in
            <SEQ,
              gen_jtable (gen_expr sel) table deflab,
```

```
              <SEQ, @(List.map2 gen_case labs arms)>,
              <LABEL deflab>,
-             gen_stmt deflt,
+             gen_stmt deflt contlab,
              <LABEL donelab>> in

    (* Label the code with a line number *)
    <SEQ, <LINE s.s_line>, code>

-(* unnest -- move procedure calls to top level *)
-let unnest code =
-  let rec do_tree =
-    function
-        <CALL n, @args> ->
-          let t = Regs.new_temp 1 in
-          <AFTER,
-            <DEFTEMP t, <CALL n, @(List.map do_tree args)>>,
-            <TEMP t>>
-      | <w, @args> ->
-          <w, @(List.map do_tree args)> in
-  let do_root =
-    function <op, @args> -> <op, @(List.map do_tree args)> in
-  Optree.canonicalise <SEQ, @(List.map do_root code)>
-
-let show label code =
-  if !debug > 0 then begin
-    printf "$$:\n" [fStr Mach.comment; fStr label];
-    List.iter (Optree.print_optree Mach.comment) code;
-    printf "\n" []
-  end;
-  code
-
 (* |do_proc| -- generate code for a procedure and pass to the back end *)
 let do_proc lab lev nargs (Block (_, body, fsize, nregv)) =
   try
-    level := lev+1;
+    proc_level := lev+1; level := !proc_level;
     retlab := label ();
     Regs.init ();
-    let code0 =
-      show "Initial code"
-        (Optree.canonicalise <SEQ, gen_stmt body, <LABEL !retlab>>) in
-    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)
-      else
-        show "After sharing" (Share.traverse code1) in
-
-    Target.start_proc lab nargs !fsize;
-    Regs.get_regvars !nregv;
-    Tran.translate (flatten code2);
-    Target.end_proc ()
+    out_proc lab nargs !fsize !nregv
+      <SEQ, gen_stmt body nolab, <LABEL !retlab>>
   with
     Failure msg ->
       let bt = Printexc.get_raw_backtrace () in
--- ../../labs/lab4/lexer.mll   2023-10-08 15:18:06.767585400 +0100
+++ lexer.mll   2023-10-08 15:30:14.506954436 +0100
```

```
@@ -13,12 +13,12 @@
   Util.make_hash 100
     [ ("array", ARRAY); ("begin", BEGIN);
       ("const", CONST); ("do", DO); ("if", IF ); ("else", ELSE);
-      ("end", END); ("of", OF); ("proc", PROC); ("record", RECORD);
-      ("return", RETURN); ("then", THEN); ("to", TO);
+      ("end", END); ("name", NAME); ("of", OF); ("proc", PROC);
+      ("record", RECORD); ("return", RETURN); ("then", THEN); ("to", TO);
       ("type", TYPE); ("var", VAR); ("while", WHILE);
       ("pointer", POINTER); ("nil", NIL);
       ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
-      ("elsif", ELSIF); ("case", CASE);
+      ("elsif", ELSIF); ("case", CASE); ("continue", CONTINUE);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
       ("not", NOT); ("mod", MULOP Mod) ]

--- ../../labs/lab4/parser.mly  2023-10-08 15:18:06.763585427 +0100
+++ parser.mly  2023-10-08 15:30:14.506954436 +0100
@@ -22,7 +22,7 @@
 %token                  ARRAY BEGIN CONST DO ELSE END IF OF
 %token                  PROC RECORD RETURN THEN TO TYPE
 %token                  VAR WHILE NOT POINTER NIL
-%token                  REPEAT UNTIL FOR ELSIF CASE
+%token                 REPEAT UNTIL FOR ELSIF CASE NAME CONTINUE

 %type <Tree.program>    program
 %start                  program
@@ -87,6 +87,7 @@
 formal_decl :
     ident_list COLON typexpr            { VarDecl (CParamDef, $1, $3) }
   | VAR ident_list COLON typexpr        { VarDecl (VParamDef, $2, $4) }
+  | NAME ident_list COLON typexpr      { VarDecl (NParamDef, $2, $4) }
   | proc_heading                        { PParamDecl $1 } ;

 return_type :
@@ -113,6 +114,7 @@
   | variable ASSIGN expr                { Assign ($1, $3) }
   | name actuals                        { ProcCall ($1, $2) }
   | RETURN expr_opt                     { Return $2 }
+  | CONTINUE                           { Continue }
   | IF expr THEN stmts elses END        { IfStmt ($2, $4, $5) }
   | WHILE expr DO stmts END             { WhileStmt ($2, $4) }
   | REPEAT stmts UNTIL expr             { RepeatStmt ($2, $4) }
--- ../../labs/lab4/tree.ml     2023-10-08 15:18:06.791585236 +0100
+++ tree.ml     2023-10-08 15:30:14.506954436 +0100
@@ -35,6 +35,7 @@
   | Assign of expr * expr
   | ProcCall of name * expr list
   | Return of expr option
+  | Continue
   | IfStmt of expr * stmt * stmt
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
@@ -140,6 +141,7 @@
     | ProcCall (p, aps) -> fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
     | Return (Some e) -> fMeta "(RETURN $)" [fExpr e]
     | Return None -> fStr "(RETURN)"
+    | Continue -> fStr "(CONTINUE)"
     | IfStmt (test, thenpt, elsept) ->
         fMeta "(IF $ $ $)" [fExpr test; fStmt thenpt; fStmt elsept]
     | WhileStmt (test, body) ->
--- ../../labs/lab4/tree.mli    2023-10-08 15:18:06.803585155 +0100
```

```
+++ tree.mli    2023-10-08 15:30:14.506954436 +0100
@@ -50,6 +50,7 @@
    | Assign of expr * expr
    | ProcCall of name * expr list
    | Return of expr option
+   | Continue
    | IfStmt of expr * stmt * stmt
    | WhileStmt of expr * stmt
    | RepeatStmt of stmt * expr
```

```
var g: integer;

proc println(x: integer);
begin
  print_num(x);
  newline()
end;

proc p(name x: integer): integer;
begin
  g := g+1;
  return x+x
end;

begin
  g := 0;
  println(p(2+3));
  println(p(g));
  println(p(p(7)));
  println(g)
end.

(*<<
10
4
28
5
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc println(x: integer);
        .text
_println:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_num(x);
        ldr r0, [fp, #40]
        bl print_num
@ newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc p(name x: integer): integer;
_p:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ g := g+1;
        ldr r4, =_g
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
@ return x+x
        ldr r10, [fp, #44]
        ldr r0, [fp, #40]
```

```
        blx r0
        ldr r10, [fp, #44]
        mov r4, r0
        ldr r0, [fp, #40]
        blx r0
        add r0, r4, r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #5
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda2:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, =_g
        ldr r0, [r0]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda4:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #7
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda3:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r1, [fp, #24]
        ldr r0, =__lambda4
        bl _p
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ g := 0;
        ldr r4, =_g
        mov r0, #0
        str r0, [r4]
@ println(p(2+3));
        mov r1, fp
        ldr r0, =__lambda1
        bl _p
        bl _println
@ println(p(g));
```

```
        mov r1, fp
        ldr r0, =__lambda2
        bl _p
        bl _println
@ println(p(p(7)));
        mov r1, fp
        ldr r0, =__lambda3
        bl _p
        bl _println
@ println(g)
        ldr r0, [r4]
        bl _println
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _g, 4, 4

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

```
proc println(x: integer);
begin
  print_num(x);
  newline()
end;

proc m();
  var g: integer;

  proc p(name x: integer): integer;
  begin
    g := g+1;
    return x+x
  end;

begin
  g := 0;
  println(p(2+3));
  println(p(g));
  println(p(p(7)));
  println(g)
end;

begin
  m()
end.

(*<<
10
4
28
5
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc println(x: integer);
        .text
_println:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_num(x);
        ldr r0, [fp, #40]
        bl print_num
@ newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc m();
@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #5
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool
```

```
@ Thunk:
__lambda2:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r0, [r0, #-4]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda4:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #7
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda3:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r4, [fp, #24]
        mov r1, r4
        ldr r0, =__lambda4
        mov r10, r4
        bl _m.p
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_m:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@ g := 0;
        mov r0, #0
        str r0, [fp, #-4]
@ println(p(2+3));
        mov r1, fp
        ldr r0, =__lambda1
        mov r10, fp
        bl _m.p
        bl _println
@ println(p(g));
        mov r1, fp
        ldr r0, =__lambda2
        mov r10, fp
        bl _m.p
        bl _println
@ println(p(p(7)));
        mov r1, fp
        ldr r0, =__lambda3
        mov r10, fp
        bl _m.p
        bl _println
@ println(g)
        ldr r0, [fp, #-4]
        bl _println
```

```
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@   proc p(name x: integer): integer;
_m.p:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ g := g+1;
        ldr r0, [fp, #24]
        add r4, r0, #-4
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
@ return x+x
        ldr r10, [fp, #44]
        ldr r0, [fp, #40]
        blx r0
        ldr r10, [fp, #44]
        mov r4, r0
        ldr r0, [fp, #40]
        blx r0
        add r0, r4, r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ m()
        bl _m
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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

```
proc iff(test: boolean; name thenpt, elsept: integer): integer;
begin
  if test then return thenpt else return elsept end
end;

proc fac(n: integer): integer;
begin
  return iff(n = 0, 1, n * fac(n-1))
end;

begin
  print_num(fac(10));
  newline()
end.

(*<<
3628800
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc iff(test: boolean; name thenpt, elsept: integer): integer;
        .text
_iff:
        mov ip, sp
        stmfd sp!, {r0-r3}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ if test then return thenpt else return elsept end
        ldrb r0, [fp, #40]
        cmp r0, #0
        beq .L3
        ldr r10, [fp, #48]
        ldr r0, [fp, #44]
        blx r0
        b .L1
.L3:
        ldr r10, [fp, #56]
        ldr r0, [fp, #52]
        blx r0
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc fac(n: integer): integer;
@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #1
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda2:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
```

```
        ldr r4, [r0, #40]
        sub r0, r4, #1
        bl _fac
        mul r0, r4, r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_fac:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@ return iff(n = 0, 1, n * fac(n-1))
        str fp, [sp, #0]
        ldr r3, =__lambda2
        mov r2, fp
        ldr r1, =__lambda1
        ldr r0, [fp, #40]
        cmp r0, #0
        mov r0, #0
        moveq r0, #1
        bl _iff
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_num(fac(10));
        mov r0, #10
        bl _fac
        bl print_num
@ newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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

```
proc sum(a, b: integer; var i: integer; name f: integer): integer;
  var s: integer;
begin
  s := 0;
  for i := a to b do s := s + f end;
  return s
end;

proc mysum(n: integer): integer;
  var j: integer;
begin
  return sum(1, n, j, j*j)
end;

begin
  print_num(mysum(10));
  newline()
end.

(*<<
385
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc sum(a, b: integer; var i: integer; name f: integer): integer;
        .text
_sum:
        mov ip, sp
        stmfd sp!, {r0-r3}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r4, #0
@ for i := a to b do s := s + f end;
        ldr r0, [fp, #40]
        ldr r1, [fp, #48]
        str r0, [r1]
        ldr r5, [fp, #44]
.L2:
        ldr r0, [fp, #48]
        ldr r0, [r0]
        cmp r0, r5
        bgt .L3
        ldr r10, [fp, #56]
        ldr r0, [fp, #52]
        blx r0
        add r4, r4, r0
        ldr r6, [fp, #48]
        ldr r0, [r6]
        add r0, r0, #1
        str r0, [r6]
        b .L2
.L3:
@ return s
        mov r0, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc mysum(n: integer): integer;
```

```
@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r4, [r0, #-4]
        mul r0, r4, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_mysum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@ return sum(1, n, j, j*j)
        str fp, [sp, #0]
        ldr r3, =__lambda1
        add r2, fp, #-4
        ldr r1, [fp, #40]
        mov r0, #1
        bl _sum
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_num(mysum(10));
        mov r0, #10
        bl _mysum
        bl print_num
@ newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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

```
type matrix = array 3 of array 3 of integer;

proc sum(var i: integer; name f: integer): integer;
  var s: integer;
begin
  s := 0;
  for i := 0 to 2 do s := s + f end;
  return s
end;

proc matsum(var a: matrix): integer;
  var i, j: integer;
begin
    return sum(i, sum(j, a[i][j]))
end;

proc test();
    var a: matrix; i, j: integer;
begin
    for i := 0 to 2 do
        for j := 0 to 2 do
            a[i][j] := (i+1)*(j+1)
        end
    end;

    print_num(matsum(a)); newline()
end;

begin test() end.

(*<<
36
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc sum(var i: integer; name f: integer): integer;
        .text
_sum:
        mov ip, sp
        stmfd sp!, {r0-r3}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r4, #0
@ for i := 0 to 2 do s := s + f end;
        mov r0, #0
        ldr r1, [fp, #40]
        str r0, [r1]
        mov r5, #2
.L2:
        ldr r0, [fp, #40]
        ldr r0, [r0]
        cmp r0, r5
        bgt .L3
        ldr r10, [fp, #48]
        ldr r0, [fp, #44]
        blx r0
        add r4, r4, r0
        ldr r6, [fp, #40]
```

```
        ldr r0, [r6]
        add r0, r0, #1
        str r0, [r6]
        b .L2
.L3:
@ return s
        mov r0, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc matsum(var a: matrix): integer;
@ Thunk:
__lambda2:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r4, [fp, #24]
        ldr r0, [r4, #40]
        ldr r1, [r4, #-4]
        mov r2, #12
        mul r1, r1, r2
        add r0, r0, r1
        ldr r1, [r4, #-8]
        lsl r1, r1, #2
        add r0, r0, r1
        ldr r0, [r0]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r4, [fp, #24]
        mov r2, r4
        ldr r1, =__lambda2
        add r0, r4, #-8
        bl _sum
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_matsum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@ return sum(i, sum(j, a[i][j]))
        mov r2, fp
        ldr r1, =__lambda1
        add r0, fp, #-4
        bl _sum
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc test();
_test:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #40
```

```
@ for i := 0 to 2 do
        mov r4, #0
        mov r0, #2
        str r0, [fp, #-40]
.L7:
        ldr r0, [fp, #-40]
        cmp r4, r0
        bgt .L8
@ for j := 0 to 2 do
        mov r5, #0
        mov r6, #2
.L10:
        cmp r5, r6
        bgt .L9
@ a[i][j] := (i+1)*(j+1)
        add r7, r5, #1
        add r0, r4, #1
        mul r0, r0, r7
        add r1, fp, #-36
        mov r2, #12
        mul r2, r4, r2
        add r1, r1, r2
        lsl r2, r5, #2
        add r1, r1, r2
        str r0, [r1]
        mov r5, r7
        b .L10
.L9:
        add r4, r4, #1
        b .L7
.L8:
@ print_num(matsum(a)); newline()
        add r0, fp, #-36
        bl _matsum
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ begin test() end.
        bl _test
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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

```
(* Knuth's "Man or Boy" test with proper name parameters *)

proc A(k: integer; name x1, x2, x3, x4, x5 : integer): integer;
  proc B(): integer;
  begin
    k := k-1;
    return A(k, B(), x1, x2, x3, x4)
  end;
begin
  if k <= 0 then return x4 + x5 else return B() end
end;

begin
  print_num(A(10, 1, -1, -1, 1, 0)); newline()
end.

(*<<
-67
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc A(k: integer; name x1, x2, x3, x4, x5 : integer): integer;
        .text
_A:
        mov ip, sp
        stmfd sp!, {r0-r3}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ if k <= 0 then return x4 + x5 else return B() end
        ldr r0, [fp, #40]
        cmp r0, #0
        bgt .L3
        ldr r10, [fp, #72]
        ldr r0, [fp, #68]
        blx r0
        ldr r10, [fp, #80]
        mov r4, r0
        ldr r0, [fp, #76]
        blx r0
        add r0, r4, r0
        b .L1
.L3:
        mov r10, fp
        bl _A.B
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@   proc B(): integer;
@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r10, [r0, #24]
        bl _A.B
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool
```

```
@ Thunk:
__lambda2:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r4, [r0, #24]
        ldr r10, [r4, #48]
        ldr r0, [r4, #44]
        blx r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda3:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r4, [r0, #24]
        ldr r10, [r4, #56]
        ldr r0, [r4, #52]
        blx r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda4:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r4, [r0, #24]
        ldr r10, [r4, #64]
        ldr r0, [r4, #60]
        blx r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda5:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
        ldr r4, [r0, #24]
        ldr r10, [r4, #72]
        ldr r0, [r4, #68]
        blx r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_A.B:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #32
@ k := k-1;
        ldr r0, [fp, #24]
        add r4, r0, #40
        ldr r0, [r4]
```

```
        sub r0, r0, #1
        str r0, [r4]
@ return A(k, B(), x1, x2, x3, x4)
        str fp, [sp, #24]
        ldr r0, =__lambda5
        str r0, [sp, #20]
        str fp, [sp, #16]
        ldr r0, =__lambda4
        str r0, [sp, #12]
        str fp, [sp, #8]
        ldr r0, =__lambda3
        str r0, [sp, #4]
        str fp, [sp, #0]
        ldr r3, =__lambda2
        mov r2, fp
        ldr r1, =__lambda1
        ldr r0, [fp, #24]
        ldr r0, [r0, #40]
        bl _A
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda6:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #1
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda7:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #-1
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda8:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #-1
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda9:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        mov r0, #1
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ Thunk:
__lambda10:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
```

```
        mov fp, sp
        mov r0, #0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #32
@ print_num(A(10, 1, -1, -1, 1, 0)); newline()
        str fp, [sp, #24]
        ldr r0, =__lambda10
        str r0, [sp, #20]
        str fp, [sp, #16]
        ldr r0, =__lambda9
        str r0, [sp, #12]
        str fp, [sp, #8]
        ldr r0, =__lambda8
        str r0, [sp, #4]
        str fp, [sp, #0]
        ldr r3, =__lambda7
        mov r2, fp
        ldr r1, =__lambda6
        mov r0, #10
        bl _A
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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

```
proc p(proc f(name x: integer): integer): integer;
  var u: integer;

  proc g(): integer;
  begin
    u := u+1;
    return u
  end;
begin
  u := 3;
  return f(g())
end;

proc q(name x: integer): integer;
begin
  return x + x
end;

begin
  print_num(p(q)); newline()
end.

(*<<
9
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc p(proc f(name x: integer): integer): integer;
@ Thunk:
        .text
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r10, [fp, #24]
        bl _p.g
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_p:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@ u := 3;
        mov r0, #3
        str r0, [fp, #-4]
@ return f(g())
        mov r1, fp
        ldr r0, =__lambda1
        ldr r10, [fp, #44]
        ldr r2, [fp, #40]
        blx r2
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@   proc g(): integer;
_p.g:
```

```
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ u := u+1;
        ldr r0, [fp, #24]
        add r4, r0, #-4
        ldr r0, [r4]
        add r0, r0, #1
        str r0, [r4]
@ return u
        ldr r0, [fp, #24]
        ldr r0, [r0, #-4]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc q(name x: integer): integer;
_q:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ return x + x
        ldr r10, [fp, #44]
        ldr r0, [fp, #40]
        blx r0
        ldr r10, [fp, #44]
        mov r4, r0
        ldr r0, [fp, #40]
        blx r0
        add r0, r4, r0
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ print_num(p(q)); newline()
        mov r1, #0
        ldr r0, =_q
        bl _p
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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

```
(* Variables used in name parameters must not live in registers *)

(* In p, local j will be made a register variable and live in r4 *)

proc p(name x: integer);
  var j: integer;
begin
  j := 48;
  print_num(x); newline()
end;

(* If we're not careful in semantic analysis, local i in q will also
   be made a register variable in r4, and the thunk for its use in
   p(i) will try to access r4 after it has been overwritten in p. *)

proc q();
  var i: integer;
begin
  i := 37;
  p(i)
end;

(* The solution is to ensure local variables used in actual name
   parameters live in the stack frame. *)

begin
  q()
end.

(*<<
37
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc p(name x: integer);
        .text
_p:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ j := 48;
        mov r4, #48
@ print_num(x); newline()
        ldr r10, [fp, #44]
        ldr r0, [fp, #40]
        blx r0
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc q();
@ Thunk:
__lambda1:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        ldr r0, [fp, #24]
```

```
        ldr r0, [r0, #-4]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

_q:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@ i := 37;
        mov r0, #37
        str r0, [fp, #-4]
@ p(i)
        mov r1, fp
        ldr r0, =__lambda1
        bl _p
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ q()
        bl _q
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

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