Compilers: practical assignment report

Candidate #: 1023505

January 25, 2019

1 Task

The task requires us to add `continue` statements and call by name semantics to the compiler given in lab 4. This report contains both a description of the changes made, as well as the tests added to test these features. Appendix A shows a full automatically generated summary of the changes made, and Appendix B contains the full text of the tests, including the output each test should generate when run, and the output generated by the compiler for these tests. A table of contents can be found at the end.

2 Continue statements

Adding `continue` statements to the language requires considering several aspects of the compiler: the lexer, the abstract syntax tree, the parser, the semantic analyser, the intermediate code generator and the machine code generator. The changes to these aspects, as well as the new tests added, are described below.

2.1 Lexical analysis

To modify the lexer for our purposes, add a corresponding symbol to `symtable`, in `lexer.mll`, as follows:

```
let symtable =
  Util.make_hash 100
    [ ("continue", CONTINUE); (* all old symbols *) ]
```

2.2 Abstract syntax

To modify the abstract syntax for our purposes, add a new constructor `ContinueStmt` to the type `stmt_guts`, in `tree.ml` and `tree.mli`, as follows:

```
and stmt_guts =
  (* all the old constructors *)
  | ContinueStmt
```

Also, modify `fStmt` to accept `ContinueStmt`'s:

```
and fStmt s =
  match s.s_guts with
    (* all old cases *)
    | ContinueStmt ->
        fMeta "(CONTINUE)" []
```

2.3 Parsing

To modify the parser in `parser.mly` to accept `continue` statements, add a new token `CONTINUE`, as follows:

```
%token                  CONTINUE
```

Also, add a new rule for `stmt1` that produces a `ContinueStmt`, as follows:

```
stmt1 :
  /* all the old rules */
  | CONTINUE                            { ContinueStmt } ;
```

2.4 Semantic analysis

Several modifications must be made to the semantic analyser from `check.ml`, to make the compiler produce a semantic error for `Continue` statements that are not within loops:

- Add a reference to a boolean called `in_loop`, in the scope of `check_stmt`, that will be made to contain `true` if and only if the statement we are currently checking is inside a loop:

```
let rec in_loop = ref false
and check_stmt s env alloc = (*...*)
```

- To check `while`, `repeat` and `for` statements, add code to `check_stmt` that modifies `in_loop` before and after the pre-existing code, to make sure that `in_loop` continues to have the property mentioned in the previous bullet point:

```
and check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with
    (* Other cases, such as Skip, Seq ss, etc. *)
    | (*While/Repeat/For*)Stmt (*Parameters*) ->
        let old_in_loop_value = !in_loop in
        in_loop := true;
        (* Old code *)
        in_loop := old_in_loop_value
```

- To check `continue` statements, if `in_loop` contains `false`, raise an appropriate semantic error:

```
and check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with
    (* Other cases *)
     | ContinueStmt ->
        if not !in_loop
        then sem_error "continue statement must be in a loop" []
        else ()
```

2.5 Intermediate code generation

To generate intermediate code for `continue` statements, we store the place in the intermediate code to which a `continue` statement should jump. The changes made, which follow, are in service to this design:

- First, we add `continue_lab`, a reference to the label to which a `continue` statement should jump, in the scope of `gen_stmt`:

```
let rec continue_lab = ref (label ())
and gen_stmt s =
```

- The code in `gen_stmt` that generates `while` statements must remember the previous value for `continue_lab`, re-assign `continue_lab` appropriately, generate code for the `while` statement (storing the result in `return_value`), restore the old value of `continue_lab`, then return the stored code:

```
and gen_stmt s =
    (* Other cases *)
      | WhileStmt (test, body) ->
          (* The test is at the top, improving the chances of finding
             common subexpressions between the test and loop body. *)
          let l1 = label () and l2 = label () and l3 = label()
          and old_continue_lab = !continue_lab in
          continue_lab := l1;
          let return_value =
              <SEQ,
                <LABEL l1>,
                gen_cond test l2 l3,
                <LABEL l2>,
                gen_stmt body,
                <JUMP l1>,
                <LABEL l3>> in
          continue_lab := old_continue_lab;
          return_value
```

- I add an extra label [1] in the intermediate code generated for `repeat` statements, immediately before the code that tests the loop condition; this is jumped to by `continue` statements in this loop. I also modify `continue_lab` before and after code generation as with `while` statements:

```
and gen_stmt s =
    (* Other cases *)
      | RepeatStmt (body, test) ->
          let l1 = label () and l2 = label () and l3 = label ()
          and old_continue_lab = !continue_lab in
          continue_lab := l2;
          let return_value =
              <SEQ,
                <LABEL l1>,
                gen_stmt body,
                <LABEL l2>,
                gen_cond test l3 l1,
                <LABEL l3>> in
          continue_lab := old_continue_lab;
          return_value
```

- `for` statements are modified very similarly to `repeat` statements:

```
and gen_stmt s =
    (* Other cases *)
      | ForStmt (var, lo, hi, body, upb) ->
          (* Use previously allocated temp variable to store upper bound.
             We could avoid this if the upper bound is constant. *)
          let tmp = match !upb with Some d -> d | _ -> failwith "for" in
          let l1 = label () and l2 = label () and l3 = label ()
          and old_continue_lab = !continue_lab in
          continue_lab := l2;
          let return_value =
              <SEQ,
                <STOREW, gen_expr lo, gen_addr var>,
                <STOREW, gen_expr hi, address tmp>,
                <LABEL l1>,
                <JUMPC (Gt, l3), gen_expr var, <LOADW, address tmp>>,
                gen_stmt body,
                <LABEL l2>,
                <STOREW,
                    <BINOP Plus, gen_expr var, <CONST 1>>, gen_addr var>,
                <JUMP l1>,
                <LABEL l3>> in
          continue_lab := old_continue_lab;
          return_value
```

- `continue` statements are translated directly into `JUMP`'s to `!continue_lab`:

```
and gen_stmt s =
    (* Other cases *)
      | ContinueStmt -> <JUMP !continue_lab>
```

2.6 Machine code generation

The pre-existing machine code generator is sufficient to generate correct code, and the code it generates is reasonably clean; the tests I created reveal no obvious inefficiencies, and most of the old tests now generate the same code, ignoring differences in label names, on optimisation level `-O2`.

[1] This does not increase code length if there are no `Continue` statements in the loop, since in that case the label added will have no `JUMP`'s to it, and thus can be optimised away during peephole optimisation

2.7 Testing

All previous tests continue to pass; additionally, I have added 6 new tests (which pass):

Test name	Test function
`while_continue.p`	Tests a `continue` statement inside a `while` loop.
`repeat_continue.p`	Tests a `continue` statement inside a `repeat` loop.
`for_continue.p`	Tests a `continue` statement inside a `for` loop.
`bare_continue.p`	Tests a `continue` statement not inside a loop [2].
`multiple_continues.p`	Tests loops that contain multiple `continue` statements.
`nested_continue.p`	Tests continue statements inside nested loops

3 Call by name semantics

First, a preliminary note; notice the peculiarities of the following code:

```
proc f(=> p : integer);
begin
    print_num(p);
    print_num(p);
    newline()
end;
proc g(var p : integer);
begin
    p := p + 1;
    return p
end;
proc h();
    var x : integer;
begin
    f(g(x) + 3)
end;
begin
    h()
end.
```

Interestingly, `f` has no way of accessing the variable `x` using only the information visible to it through it's static link or dynamic link, so it must somehow be passed this variable's address (the value is not sufficient, because `g` changes the value of `x`). Rather than exhaustively passing the address of each variable mentioned in an argument directly, which might take very much stack space, and is quite complicated, it makes more sense to simply wrap each call by name parameter in a thunk (i.e. a procedure created implicitly by the compiler), and pass that. In this design, the code in the example should compile to essentially the same intermediate code as:

```
proc f(proc p(): integer): integer;
begin
    print_num(p());
    print_num(p());
    newline()
end;
proc g(var p : integer): integer;
begin
    p := p + 1;
    return p
end;
proc h(): integer;
    var x : integer;
    proc thunk();
    begin
        return g(x) + 3;
    end;
begin
    f(thunk)
end;
begin
    h()
end.
```

[2] NB: this should produce an error message – to be able to test this, I have extended the testing script in the `Makefile` so that error messages can be tested

In essence, in this design a function that accepts a call by name parameter will treat that parameter precisely as though it were a function that takes no arguments and returns an `integer`; and code calling a function with such parameters will wrap the argument it passes in a thunk.
This deisgn will also not allow getting the address of a call by name parameter (i.e. not letting such parameters be assigned to, and not letting them be passed as call by reference parameters). This is in accordance with the specification given in the task, and does not substantially reduce functionality (as demonstrated by the test cases that implement Jensen's device and Knuth and Merner's general problem solver), since our language includes call by reference parameters.
I split the implementation of this feature into the same parts as before.

3.1 Lexical analysis

To modify the lexer for our purposes, add a new rule to `rule token`, in `lexer.mll`:

```
rule token =
    (* Other rules *)
    | "=>"              { RIGHTARROW }
```

3.2 Abstract Syntax

We must add a new constructor `NParamDef` to the type `def_kind`, defined in `dict.ml` and `dict.mli`:

```
type def_kind =
    (* Other constructors *)
  | NParamDef                   (* Named parameter *)
    (* Other constructors *)
```

Also, I extend `fKind` in `tree.ml` to accept `NParamDef`'s:

```
and fKind =
  function
    (* Other cases *)
    | NParamDef -> fStr "NPARAM"
    (* Other cases *)
```

3.3 Parsing

To modify the parser from `parser.mly` to accept call by name parameters, first add a new token `RIGHTARROW`, as follows:

```
%token                  RIGHTARROW
```

and then add a new rule to `formal_decl`, as follows:

```
formal_decl :
    /* Other rules */
   | RIGHTARROW ident_list COLON typexpr { VarDecl (NParamDef, $2, $4) } ;
```

3.4 Semantic Analysis

Several changes must be made to the semantic analyser found in `check.ml`, so as to comply with the specification given in the task (which specifies several cases that should lead to errors), and to comply with the design outlined earlier:

- I make `has_value` recognise that `NParamDef`'s have values:

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

- I make `do_alloc` recognise that `NParamDef`'s need to be allocated:

```
let do_alloc alloc ds =
  let h d =
    match d.d_kind with
        VarDef | CParamDef | VParamDef | FieldDef | PParamDef | NParamDef->
          alloc d
```

```
      | _ -> () in
  List.iter h ds
```

- I make `NParamDef` work precisely like `CParamDef` and `VParamDef` in `check_arg`, since the same semantic checks must be done on call by name arguments as with call by reference or call by value arguments:

```
and check_arg formal arg env =
  match formal.d_kind with
        CParamDef | VParamDef | NParamDef-> (* Old code for CParamDef and VParamDef *)
```

- I make `NParamDef` work precisely like `PParamDef` in `param_alloc`, since a call by name parameter is implemented as a function parameter, and thus should be allocated as one:

```
let param_alloc pcount d =
  let s = param_rep.r_size in
  match d.d_kind with
    (* Other cases *)
    | PParamDef | NParamDef ->
        d.d_addr <- Local (param_base + s * !pcount);
        pcount := !pcount + 2
    (* Other cases *)
```

- I modify `check_var` to output an appropriate error message for call by name parameters.

```
let rec check_var e addressible =
  match e.e_guts with
      Variable x ->
        let d = get_def x in
        begin
          match d.d_kind with
            (* Other cases *)
            | NParamDef ->
                sem_error "$ is a call by name parameter, and has no address" [fId x.
    x_name]
            (* Other cases *)
        end
    (* Other cases *)
$
```

- I add functions `contains_call_by_name` (which checks if a particular declaration contains a call by name parameter at any level) and `check_param` (which, for a given declaration, checks that all call by name parameters are integers, and that no parameters are functions that take call by name parameters):

```
(* This will check if it's argument, a formal parameter, contains
 * a call by name parameter at any level *)
and contains_call_by_name d = match d with
  VarDecl(NParamDef, _, _) -> true
  | PParamDecl(Heading(_, params, _)) -> List.exists contains_call_by_name params
  | _ -> false

(* This checks if a formal parameter is acceptable *)
and check_param env d = match d with
  (* If the formal parameter is a call by name parameter
   * we check if it is of intergral type *)
  VarDecl(NParamDef, _, te) ->
    let t = check_typexpr te env in
    (
    match t.t_guts with
      BasicType IntType -> ()
      | _ -> sem_error "Call by name parameter must be an integer" []
    )
  (* If the parameter is a procedure, then we check that
   * it contains no call by name parameters at any level *)
  | PParamDecl (Heading(_, params, _)) ->
      if List.exists contains_call_by_name params
```

```
      then sem_error "Functions that use call by name parameters cannot be parameters" []
      else ()
  | _ -> ()
```

3.5 Intermediate code generation

Several changes must be made to `tgen.ml`, in order to comply with the design specified previously. In general, I will modify the parts of the code that consume call by name parameters to make them treat these parameters like function parameters that take no arguments and return an integer, and I will modify the parts of the code that pass call by name arguments so as to have them pass a thunk instead, simultaneously adding the thunk that needs to be produced to a global list of such thunks. Near the end of intermediate code generation, I will then build all the thunks in this global list. As these thunks may also contain function calls that use call by name parameters, and thus may also spawn new thunks of their own, it is necessary to repeatedly build all the thunks in the global list until no more are spawned, removing thunks that are already built along the way so as not to build a thunk twice. The changes made to this part of the compiler now follow:

- I make `gen_closure` work on `NParamDef`'s the same as it does on `PParamDef`'s:

```
let gen_closure d =
  match d.d_kind with
      (* Other cases *)
    | PParamDef | NParamDef->
        (<LOADW, address d>,
          <LOADW, <OFFSET, address d, <CONST addr_size>>>)
      (* Other cases *)
```

- I make `gen_addr` generate an appropriate error (`Named parameters have no address`) when asked to generate an address for a call by name parameter.

```
(* |gen_addr| -- code for the address of a variable *)
let rec gen_addr v =
  match v.e_guts with
      Variable x ->
        let d = get_def x in
        begin
          match d.d_kind with
            (* Other cases *)
            | NParamDef -> failwith "Named parameters have no address"
            (* Other cases *)
        end
      (* Other cases *)
```

- I make `gen_expr` treat `NParamDef`'s like function calls that take no arguments:

```
and gen_expr e =
  match e.e_value with
      Some v ->
        <CONST v>
    | None ->
        begin
          match e.e_guts with
              Variable x when (get_def x).d_kind == NParamDef ->
                gen_call x []
              (* other cases *)
        end
```

- I make `gen_call` treat `NParamDef`'s like normal procedures, with no parameters:

```
and gen_call x args =
  let d = get_def x in
  match d.d_kind with
    (* Other cases *)
    | NParamDef ->
        let (fn, sl) = gen_closure d in
        <CALL 0, @(fn :: <STATLINK, sl> :: [])>
    (* Other cases *)
```

- I add a new counter `curr_thunk` and an associated function `get_thunk_label`, which generates names for thunks. The names are of the form `__thunk_id`, where `id` is the current value of `curr_thunk` (which shall be incremented after each call of `get_thunk_label`); this scheme insures that no two thunk names can coincide (due to `id`), and that a thunk's name will never coincide with a user defined function's name (since such names cannot begin with `__`):

```
and curr_thunk = ref 0
and get_thunk_label () =
    curr_thunk := 1 + !curr_thunk;
    sprintf "__thunk_$" [fNum !curr_thunk]
$
```

- I add a new global variable, `thunks_to_be_generated`, a reference to an initially empty list. This will hold 3-tuples `(label, level, block)`, each of which represent that we need to generate a thunk whose name is `label`, at level `level`, whose body consists of `block`.
- I make `gen_arg` treat `NParamDef`'s as follows:
  1. We extract the expression which is passed as a parameter.
  2. We create a block of code that corresponds to a body of a function that simply returns that expression
  3. We create an thunk name, using `get_thunk_label`.
  4. We push these values onto `thunks_to_be_generated`, to be generated later.
  5. We generate code that pushes a closure for the function we just asked to be created onto the stack.

  The code follows:

```
and gen_arg f a =
  match f.d_kind with
      (* Other cases *)
    | NParamDef ->
        (
        (* To generate a call by name argument we wrap the argument expression
         * in a block that simply returns it: *)
        let block = makeBlock ([], {s_guts = Return (Some a); s_line = -1}) in
        (* then create a label for a thunk *)
        let lab = get_thunk_label () in
        (* and add the thunk into the global list: *)
        thunks_to_be_generated :=
          (lab, !level, block) :: !thunks_to_be_generated;
        (* Now, the actual code is simply a closure for that
         * thunk *)
        [<GLOBAL lab>; <LOCAL 0>]
        )
    | _ -> failwith "bad arg"
```

- I add a new function `build_all_thunks` that will build code for all the thunks in `thunks_to_be_generated`, and then, if some new thunks have been added to the `thunks_to_be_generated`, will call itself recursively:

```
let rec build_all_thunks () =
  (* This reverses in order to build the thunks
   * in the order they are required *)
  let tmp = List.rev !thunks_to_be_generated in
  thunks_to_be_generated := [];
  (* I build all the thunks required *)
  List.map (fun (lab, lev, bl) -> do_proc lab lev 0 bl) tmp;
  (* And check if any more are now required *)
  if (List.length !thunks_to_be_generated > 0)
    then build_all_thunks ()
    else ()
```

- I modify `translate` to make it call `build_all_thunks`:

```
let translate (Prog (block, glodefs)) =
  Target.preamble ();
  gen_procs (get_decls block);
  do_proc "pmain" 0 0 block;
  build_all_thunks ();
  List.iter gen_global !glodefs;
  List.iter (fun (lab, s) -> Target.emit_string lab s) (string_table ());
  Target.postamble ()
```

3.6 Machine code generation

The pre-existing machine code generator is sufficient to generate correct code, and the code it generates is reasonably clean, as demonstrated by the tests that follow.

3.7 Testing

All previous tests continue to pass; additionally, I have added several new tests (which pass):

Test name	Test function
`by_name_not_param.p`	Tests that using a function that takes a call by name parameter as a parameter generates an appropriate error.
`rightarrow_non_int_error.p`	Tests that using a non-integer call by name parameter leads to an appropriate error.
`given_named_param_test.p`	Contains the example given in the task description. This test also demonstrates that the compiler can optimise constant expressions passed by name.
`lazy_evaluate_by_name.p`	Tests that call by name parameters are not evaluated if they are not used.
`reevaluate_by_name.p`	Tests that call by name parameters are re-evaluated each time they are used.
`rightarrow_same_as_var.p`	Tests that call by name parameter declarations are syntactically similar to call by reference parameter declarations (i.e. they can be followed by a list of parameters).
`by_name_local.p`	Tests that a call by name parameter containing a local variable not in the scope of the called function works correctly.
`by_name_nested_function.p`	Tests that a call by name parameter containing implicit changes to state not in the scope of the called function works correctly.
`by_name_to_by_name.p`	Tests that a call by name parameter can be passed as a call by name parameter to another function.
`by_reference_to_by_name.p`	Tests that a call by reference parameter can be passed as a call by name parameter to another function.
`by_name_not_by_ref.p`	Tests that a call by name parameter cannot be passed as a call by reference parameter to another function.
`by_name_not_assigned.p`	Tests that a call by reference parameter cannot be assigned to.
`gps_primes.p`	Tests an application of Knuth and Merner's general problem solver that calculates prime numbers.
`jensen1.p`	Tests both a simple application of Jensen's device, and that local variables can be passed as call by name parameters.
`jensen2.p`	Tests both a more complicated application of Jensen's device, and that global variables can be passed as call by name parameters.
`jensen2d.p`	Tests using Jensen's device to calculate the sum of a matrix, and that locals can be passed by name.
`jensen3.p`	Tests complicated application of Jensen's device, and that global variables can be passed as call by name parameters.
`jensen_function.p`	Calculates $1^4 + ... + 9^4$ using Jensen's device.
`by_name_iterator.p`	Uses call by name parameters to implement an iterator-like construct

4 Appendix A

A full, automatically generated (with `hg diff -r initial_revision_number *.{mll,mli,mly,ml}`), description of the changes made to the compiler:

```
diff -r 9f5c8e19f204 lab4/check.ml
--- a/lab4/check.ml Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/check.ml Wed Jan 23 22:40:39 2019 +0000
@@ -90,7 +90,7 @@
  (* |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 | StringDef | NParamDef-> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -102,6 +102,8 @@
           match d.d_kind with
               VarDef | VParamDef | CParamDef ->
                 d.d_mem <- d.d_mem || addressible
+            | NParamDef ->
+                sem_error "$ is a by-name parameter, and has no address" [fId x.x_name]
             | _ ->
                 sem_error "$ is not a variable" [fId x.x_name]
         end
@@ -198,7 +200,7 @@
 (* |check_arg| -- check one (formal, actual) parameter pair *)
 and check_arg formal arg env =
   match formal.d_kind with
-      CParamDef | VParamDef ->
+      CParamDef | VParamDef | NParamDef->
         let t1 = check_expr arg env in
         if not (same_type formal.d_type t1) then
           sem_error "argument has wrong type" [];
@@ -293,7 +295,8 @@
   chk (List.sort compare vs)

 (* |check_stmt| -- check and annotate a statement *)
-let rec check_stmt s env alloc =
+let rec in_loop = ref false
+and check_stmt s env alloc =
   err_line := s.s_line;
   match s.s_guts with
       Skip -> ()
@@ -337,18 +340,26 @@
         check_stmt elsept env alloc

     | WhileStmt (cond, body) ->
+        let old_in_loop_value = !in_loop in
+        in_loop := true;
         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;
+        in_loop := old_in_loop_value

     | RepeatStmt (body, test) ->
+        let old_in_loop_value = !in_loop in
+        in_loop := true;
         check_stmt body env alloc;
         let ct = check_expr test env in
         if not (same_type ct boolean) then
-          sem_error "type mismatch in repeat statement" []
+            sem_error "type mismatch in repeat statement" [];
+        in_loop := old_in_loop_value

     | ForStmt (var, lo, hi, body, upb) ->
+        let old_in_loop_value = !in_loop in
+        in_loop := true;
         let vt = check_expr var env in
         let lot = check_expr lo env in
         let hit = check_expr hi env in
@@ -361,7 +372,8 @@
         (* Allocate space for hidden variable.  In the code, this will
            be used to save the upper bound.  *)
         let d = make_def (intern "*upb*") VarDef integer in
-        alloc d; upb := Some d
+        alloc d; upb := Some d;
```

```
+        in_loop := old_in_loop_value

     | CaseStmt (sel, arms, deflt) ->
         let st = check_expr sel env in
@@ -377,6 +389,10 @@
         let vs = List.map check_arm arms in
         check_dupcases vs;
         check_stmt deflt env alloc
+     | ContinueStmt ->
+        if not !in_loop
+        then sem_error "continue statement must be in a loop" []
+        else ()


 (* TYPES AND DECLARATIONS *)
@@ -421,7 +437,7 @@
       CParamDef | VParamDef ->
         d.d_addr <- Local (param_base + s * !pcount);
         incr pcount
-    | PParamDef ->
+    | PParamDef | NParamDef ->
         d.d_addr <- Local (param_base + s * !pcount);
         pcount := !pcount + 2
     | _ -> failwith "param_alloc"
@@ -434,7 +450,7 @@
 let do_alloc alloc ds =
   let h d =
     match d.d_kind with
-        VarDef | CParamDef | VParamDef | FieldDef | PParamDef ->
+        VarDef | CParamDef | VParamDef | FieldDef | PParamDef | NParamDef->
           alloc d
       | _ -> () in
   List.iter h ds
@@ -515,7 +531,33 @@
         let t = check_heading env heading in
         let d = make_def x.x_name PParamDef t in
         add_def d env
-
+
+(* This will check if it's argument, a formal parameter, contains
+ * a by-name parameter at any level *)
+and contains_call_by_name d = match d with
+  VarDecl(NParamDef, _, _) -> true
+  | PParamDecl(Heading(_, params, _)) -> List.exists contains_call_by_name params
+  | _ -> false
+
+(* This checks if a formal parameter is acceptable *)
+and check_param env d = match d with
+  (* If the formal parameter is a by-name parameter
+   * we check if it is of intergral type *)
+  VarDecl(NParamDef, _, te) ->
+    let t = check_typexpr te env in
+    (
+    match t.t_guts with
+      BasicType IntType -> ()
+      | _ -> sem_error "Call by name parameter must be an integer" []
+    )
+  (* If the parameter is a procedure, then we check that
+   * it contains no by-name parameters at any level *)
+  | PParamDecl (Heading(_, params, _)) ->
+      if List.exists contains_call_by_name params
+      then sem_error "Functions that use call-by-name parameters cannot be parameters" []
+      else ()
+  | _ -> ()
+
 (* |check_heading| -- process a procedure heading into a procedure type *)
 and check_heading env (Heading (x, fparams, result)) =
   err_line := x.x_line;
@@ -529,6 +571,7 @@
               Some te -> check_typexpr te env | None -> voidtype) in
```

```
   if not (same_type rt voidtype) && not (scalar rt) then
     sem_error "return type must be scalar" [];
+  List.map (check_param env) fparams;
   let p = { p_fparams = defs; p_pcount = !pcount; p_result = rt } in
   mk_type (ProcType p) proc_rep

diff -r 9f5c8e19f204 lab4/dict.ml
--- a/lab4/dict.ml  Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/dict.ml  Wed Jan 23 22:40:39 2019 +0000
@@ -80,6 +80,7 @@
   | VarDef                      (* Variable *)
   | CParamDef                   (* Value parameter *)
   | VParamDef                   (* Var parameter *)
+  | NParamDef                   (* Named parameter *)
   | FieldDef                    (* Field of record *)
   | ProcDef                     (* Procedure *)
   | PParamDef                   (* Proc parameter *)
diff -r 9f5c8e19f204 lab4/dict.mli
--- a/lab4/dict.mli Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/dict.mli Wed Jan 23 22:40:39 2019 +0000
@@ -43,6 +43,7 @@
   | VarDef                      (* Variable *)
   | CParamDef                   (* Value parameter *)
   | VParamDef                   (* Var parameter *)
+  | NParamDef                   (* Named parameter *)
   | FieldDef                    (* Field of record *)
   | ProcDef                     (* Procedure *)
   | PParamDef                   (* Proc parameter *)
diff -r 9f5c8e19f204 lab4/lexer.mll
--- a/lab4/lexer.mll  Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/lexer.mll  Wed Jan 23 22:40:39 2019 +0000
@@ -13,7 +13,8 @@

 let symtable =
   Util.make_hash 100
-    [ ("array", ARRAY); ("begin", BEGIN);
+    [ ("continue", CONTINUE);
+      ("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);
@@ -96,6 +97,7 @@
     | "<>"              { RELOP Neq }
     | "<="              { RELOP Leq }
     | ">="              { RELOP Geq }
+    | "=>"              { RIGHTARROW }
     | ":="              { ASSIGN }
     | [' ''\t']+        { token lexbuf }
     | "(*"              { comment lexbuf; token lexbuf }
diff -r 9f5c8e19f204 lab4/parser.mly
--- a/lab4/parser.mly Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/parser.mly Wed Jan 23 22:40:39 2019 +0000
@@ -23,6 +23,8 @@
 %token                  PROC RECORD RETURN THEN TO TYPE
 %token                  VAR WHILE NOT POINTER NIL
 %token                  REPEAT UNTIL FOR ELSIF CASE
+%token                  CONTINUE
+%token                  RIGHTARROW

 %type <Tree.program>    program
 %start                  program
@@ -87,6 +89,7 @@
 formal_decl :
     ident_list COLON typexpr            { VarDecl (CParamDef, $1, $3) }
   | VAR ident_list COLON typexpr        { VarDecl (VParamDef, $2, $4) }
+  | RIGHTARROW ident_list COLON typexpr { VarDecl (NParamDef, $2, $4) }
   | proc_heading                        { PParamDecl $1 } ;

 return_type :
@@ -119,7 +122,8 @@
```

```
   | 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) } ;
+  | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) }
+  | CONTINUE                            { ContinueStmt } ;

 elses :
     /* empty */                        { makeStmt (Skip, 0) }
diff -r 9f5c8e19f204 lab4/tgen.ml
--- a/lab4/tgen.ml  Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/tgen.ml  Wed Jan 23 22:40:39 2019 +0000
@@ -8,6 +8,7 @@
 open Lexer
 open Print

+let thunks_to_be_generated = ref []
 let boundchk = ref false
 let optlevel = ref 0
 let debug = ref 0
@@ -64,7 +65,7 @@
       ProcDef ->
         (address 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_size>>>)
     | _ -> failwith "missing closure"
@@ -99,6 +100,7 @@
                   address d
                 else
                   <LOADW, address d>
+            | NParamDef -> failwith "Named parameters have no address"
             | StringDef ->
                 address d
             | _ ->
@@ -128,7 +130,9 @@
     | None ->
         begin
           match e.e_guts with
-              Variable _ | Sub _ | Select _ | Deref _ ->
+              Variable x when (get_def x).d_kind == NParamDef ->
+                gen_call x []
+            | Variable _ | Sub _ | Select _ | Deref _ ->
                 let ld = if size_of e.e_type = 1 then LOADC else LOADW in
                 <ld, gen_addr e>
             | Monop (w, e1) ->
@@ -150,12 +154,20 @@
   match d.d_kind with
       LibDef q ->
         gen_libcall q args
+    | NParamDef ->
+        let (fn, sl) = gen_closure d in
+        <CALL 0, @(fn :: <STATLINK, sl> :: [])>
     | _ ->
         let p = get_proc d.d_type in
         let (fn, sl) = gen_closure d in
         let args = List.concat (List.map2 gen_arg p.p_fparams args) in
         <CALL p.p_pcount, @(fn :: <STATLINK, sl> :: numargs 0 args)>

+and curr_thunk = ref 0
+and get_thunk_label () =
+    curr_thunk := 1 + !curr_thunk;
+    sprintf "__thunk_$" [fNum !curr_thunk]
+
 (* |gen_arg| -- generate code for a procedure argument *)
 and gen_arg f a =
   match f.d_kind with
@@ -174,6 +186,20 @@
```

```
             | _ ->
                 failwith "bad funarg"
         end
+    | NParamDef ->
+        (
+        (* To generate a by-name argument we wrap the argument expression
+         * in a block that simply returns it: *)
+        let block = makeBlock ([], {s_guts = Return (Some a); s_line = -1}) in
+        (* then create a label for a thunk *)
+        let lab = get_thunk_label () in
+        (* and add the thunk into the global list: *)
+        thunks_to_be_generated :=
+          (lab, !level, block) :: !thunks_to_be_generated;
+        (* Now, the actual code is simply a closure for that
+         * thunk *)
+        [<GLOBAL lab>; <LOCAL 0>]
+        )
     | _ -> failwith "bad arg"

 (* |gen_libcall| -- generate code to call a built-in procedure *)
@@ -250,7 +276,8 @@
   end

 (* |gen_stmt| -- generate code for a statement *)
-let rec gen_stmt s =
+let rec continue_lab = ref (label ())
+and gen_stmt s =
   let code =
     match s.s_guts with
         Skip -> <NOP>
@@ -289,37 +316,55 @@
       | WhileStmt (test, body) ->
           (* The test is at the top, improving the chances of finding
              common subexpressions between the test and loop body. *)
-          let l1 = label () and l2 = label () and l3 = label() in
-          <SEQ,
-            <LABEL l1>,
-            gen_cond test l2 l3,
-            <LABEL l2>,
-            gen_stmt body,
-            <JUMP l1>,
-            <LABEL l3>>
+          let l1 = label () and l2 = label () and l3 = label()
+          and old_continue_lab = !continue_lab in
+          continue_lab := l1;
+          let return_value =
+              <SEQ,
+                <LABEL l1>,
+                gen_cond test l2 l3,
+                <LABEL l2>,
+                gen_stmt body,
+                <JUMP l1>,
+                <LABEL l3>> in
+          continue_lab := old_continue_lab;
+          return_value

       | RepeatStmt (body, test) ->
-          let l1 = label () and l2 = label () in
-          <SEQ,
-            <LABEL l1>,
-            gen_stmt body,
-            gen_cond test l2 l1,
-            <LABEL l2>>
+          let l1 = label () and l2 = label () and l3 = label ()
+          and old_continue_lab = !continue_lab in
+          continue_lab := l3;
+          let return_value =
+              <SEQ,
+                <LABEL l1>,
+                gen_stmt body,
```

```
+                <LABEL l3>,
+                gen_cond test l2 l1,
+                <LABEL l2>> in
+           continue_lab := old_continue_lab;
+           return_value

       | ForStmt (var, lo, hi, body, upb) ->
           (* Use previously allocated temp variable to store upper bound.
              We could avoid this if the upper bound is constant. *)
           let tmp = match !upb with Some d -> d | _ -> failwith "for" in
-           let l1 = label () and l2 = label () in
-           <SEQ,
-             <STOREW, gen_expr lo, gen_addr var>,
-             <STOREW, gen_expr hi, address tmp>,
-             <LABEL l1>,
-             <JUMPC (Gt, l2), gen_expr var, <LOADW, address tmp>>,
-             gen_stmt body,
-             <STOREW, <BINOP Plus, gen_expr var, <CONST 1>>, gen_addr var>,
-             <JUMP l1>,
-             <LABEL l2>>
+           let l1 = label () and l2 = label () and l3 = label ()
+           and old_continue_lab = !continue_lab in
+           continue_lab := l3;
+           let return_value =
+               <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,
+                 <LABEL l3>,
+                 <STOREW,
+                     <BINOP Plus, gen_expr var, <CONST 1>>, gen_addr var>,
+                 <JUMP l1>,
+                 <LABEL l2>> in
+           continue_lab := old_continue_lab;
+           return_value

       | CaseStmt (sel, arms, deflt) ->
           (* Use one jump table, and hope it is reasonably compact *)
@@ -337,7 +382,9 @@
             <SEQ, @(List.map2 gen_case labs arms)>,
             <LABEL deflab>,
             gen_stmt deflt,
-             <LABEL donelab>> in
+             <LABEL donelab>>
+
+      | ContinueStmt -> <JUMP !continue_lab> in

    (* Label the code with a line number *)
    <SEQ, <LINE s.s_line>, code>
@@ -408,12 +455,24 @@
         Target.emit_global (get_label d) (size_of d.d_type)
     | _ -> ()

+let rec build_all_thunks () =
+  (* This reverses in order to build the thunks
+   * in the order they are required *)
+  let tmp = List.rev !thunks_to_be_generated in
+  thunks_to_be_generated := [];
+  (* I build all the thunks required *)
+  List.map (fun (lab, lev, bl) -> do_proc lab lev 0 bl) tmp;
+  (* And check if any more are now required *)
+  if (List.length !thunks_to_be_generated > 0)
+    then build_all_thunks ()
+    else ()
+
 (* |translate| -- generate code for the whole program *)
 let translate (Prog (block, glodefs)) =
   Target.preamble ();
```

```
   gen_procs (get_decls block);
   do_proc "pmain" 0 0 block;
+  build_all_thunks ();
   List.iter gen_global !glodefs;
   List.iter (fun (lab, s) -> Target.emit_string lab s) (string_table ());
   Target.postamble ()
-
diff -r 9f5c8e19f204 lab4/tree.ml
--- a/lab4/tree.ml  Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/tree.ml  Wed Jan 23 22:40:39 2019 +0000
@@ -39,6 +39,7 @@
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt * def option ref
   | CaseStmt of expr * (expr * stmt) list * stmt
+  | ContinueStmt

 and expr =
   { e_guts: expr_guts;
@@ -124,6 +125,7 @@
       VarDef -> fStr "VAR"
     | CParamDef -> fStr "PARAM"
     | VParamDef -> fStr "VPARAM"
+    | NParamDef -> fStr "NPARAM"
     | FieldDef -> fStr "FIELD"
     | _ -> fStr "???"

@@ -150,6 +152,8 @@
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
         fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]
+    | ContinueStmt ->
+        fMeta "(CONTINUE)" []

 and fExpr e =
   match e.e_guts with
diff -r 9f5c8e19f204 lab4/tree.mli
--- a/lab4/tree.mli Mon Oct 15 21:58:03 2018 +0100
+++ b/lab4/tree.mli Wed Jan 23 22:40:39 2019 +0000
@@ -54,6 +54,7 @@
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt * def option ref
   | CaseStmt of expr * (expr * stmt) list * stmt
+  | ContinueStmt

 and expr =
   { e_guts: expr_guts;
```

5 Appendix B

The full text of all tests, in alphabetical order, follows:

5.1 bare_continue.p

```
(* A continue statement not inside a loop *)
begin
    continue;
end.

(*<<
"test/bare_continue.p", line 3: continue
    statement must be in a loop
>>*)
(*[[
]]*)
```

5.2 by_name_iterator.p

```
(* This tests an "iterator-like" construct
    that can be made using call by name *)

(* This will continuously print iterator,
    until done is non-zero *)
proc print(=> iterator, done: integer);
begin
    while done = 0 do
        print_num(iterator);
        newline()
    end;
end;

(* An example that uses arrays *)
proc array_iterator(a: array 20 of integer;
    var i: integer):integer ;
    var ret: integer;
begin
```

```
    ret := a[i];
    i := i+1;
    return ret
end;

proc array_done(n: integer; i: integer):
    integer;
begin
    if i >= n then
        return 1
    else
        return 0
    end;
end;

proc do_array_test();
    var arr: array 20 of integer;
    var i: integer;
begin
    i := 0;
    while i < 20 do
        arr[i] := i;
        i := i+1
    end;
    i := 0;
    print(array_iterator(arr, i), array_done
    (20, i))
end;

(* An example that uses linked lists *)
type
    list_ptr = pointer to list;
    list = record data: integer; next:
    list_ptr end;

proc list_iterator(var p: list_ptr): integer
    ;
    var ret: integer;
begin
    ret := p^.data;
    p := p^.next;
    return ret
end;

proc list_done(p: list_ptr): integer;
begin
    if p = nil then
        return 1
    else
        return 0
    end;
end;

proc do_list_test();
    var i: integer;
    var p: list_ptr;
    var q: list_ptr;
begin
    i := 0;
    p := nil;

    while i < 20 do
        q := p;
        new(p);
        p^.next := q;
        p^.data := i;
        i := i+1
    end;

    q := p;
    print(list_iterator(q), list_done(q));
end;

begin
    do_list_test();
    do_array_test()

end.
(*<<
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc print(=> iterator, done: integer);
  .text
_print:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
.L3:
@     while done = 0 do
  ldr r10, [fp, #52]
  ldr r0, [fp, #48]
  blx r0
  cmp r0, #0
  bne .L5
@         print_num(iterator);
```

```
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  bl print_num
@         newline()
  bl newline
  b .L3
.L5:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc array_iterator(a: array 20 of integer
    ; var i: integer):integer ;
_array_iterator:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     ret := a[i];
  ldr r5, [fp, #44]
  ldr r6, [r5]
  ldr r0, [fp, #40]
  lsl r1, r6, #2
  add r0, r0, r1
  ldr r4, [r0]
@     i := i+1;
  add r0, r6, #1
  str r0, [r5]
@     return ret
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc array_done(n: integer; i: integer):
    integer;
_array_done:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     if i >= n then
  ldr r0, [fp, #44]
  ldr r1, [fp, #40]
  cmp r0, r1
  blt .L9
@         return 1
  mov r0, #1
  b .L7
.L9:
@         return 0
  mov r0, #0
.L7:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc do_array_test();
_do_array_test:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #88
@     i := 0;
  mov r0, #0
  str r0, [fp, #-84]
.L12:
@     while i < 20 do
  ldr r4, [fp, #-84]
  cmp r4, #20
  bge .L14
@         arr[i] := i;
  add r0, fp, #-80
  lsl r1, r4, #2
  add r0, r0, r1
  str r4, [r0]
@         i := i+1
  ldr r0, [fp, #-84]
  add r0, r0, #1
  str r0, [fp, #-84]
  b .L12
.L14:
@     i := 0;
  mov r0, #0
  str r0, [fp, #-84]
@     print(array_iterator(arr, i),
    array_done(20, i))
  mov r3, fp
  set r2, __thunk_2
  mov r1, fp
  set r0, __thunk_1
  bl _print
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc list_iterator(var p: list_ptr):
    integer;
_list_iterator:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     ret := p^.data;
  ldr r5, [fp, #40]
  ldr r6, [r5]
  ldr r4, [r6]
@     p := p^.next;
  ldr r0, [r6, #4]
  str r0, [r5]
@     return ret
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc list_done(p: list_ptr): integer;
_list_done:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     if p = nil then
  ldr r0, [fp, #40]
  cmp r0, #0
  bne .L18
@         return 1
  mov r0, #1
  b .L16
.L18:
@         return 0
  mov r0, #0
.L16:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc do_list_test();
_do_list_test:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     i := 0;
  mov r4, #0
```

```
@     p := nil;
  mov r5, #0
.L21:
@     while i < 20 do
  cmp r4, #20
  bge .L23
@         q := p;
  str r5, [fp, #-4]
@         new(p);
  mov r0, #8
  bl new
  mov r5, r0
@         p^.next := q;
  ldr r0, [fp, #-4]
  str r0, [r5, #4]
@         p^.data := i;
  str r4, [r5]
@         i := i+1
  add r4, r4, #1
  b .L21
.L23:
@     q := p;
  str r5, [fp, #-4]
@     print(list_iterator(q), list_done(q));
  mov r3, fp
  set r2, __thunk_4
  mov r1, fp
  set r0, __thunk_3
  bl _print
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     do_list_test();
  bl _do_list_test
@     do_array_test()
  bl _do_array_test
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  ldr r4, [fp, #24]
  add r1, r4, #-84
  add r0, r4, #-80
  bl _array_iterator
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_2:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  ldr r1, [r0, #-84]
  mov r0, #20
  bl _array_done
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_3:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  add r0, r0, #-4
  bl _list_iterator
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_4:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  ldr r0, [r0, #-4]
  bl _list_done
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ End
]]*)
```

5.3 by_name_local.p

```
proc f(=> x : integer);
begin
    print_num(x);
    newline ();
    print_num(x);
    newline ()
end;
proc g(var x : integer) : integer;
begin
    x := x + 1;
    return x
end;

proc h();
    var x : integer;
begin
    x := 0;
    f(g(x) + 3)
end;
begin
    h()
end.

(*<<
4
5
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x : integer);
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     print_num(x);
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  bl print_num
@     newline ();
  bl newline
@     print_num(x);
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
```

```
  blx r0
  bl print_num
@     newline ()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

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

@ proc h();
_h:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     x := 0;
  mov r0, #0
  str r0, [fp, #-4]
@     f(g(x) + 3)
  mov r1, fp
  set r0, __thunk_1
  bl _f
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     h()
  bl _h
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  ldr r0, [fp, #24]
  add r0, r0, #-4
  bl _g
  add r0, r0, #3
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ End
]]*)
```

5.4 by_name_nested_function.p

```
proc f(=> x : integer);
begin
    print_num(x);
    newline ();
    print_num(x);
    newline ()
end;

proc h();
    var x : integer;
    proc g() : integer;
    begin
        x := x + 1;
        return x
    end;
begin
    x := 0;
    f(g() + 3)
end;
begin
    h()
end.

(*<<
4
5
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x : integer);
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     print_num(x);
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  bl print_num
@     newline ();
  bl newline
@     print_num(x);
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  bl print_num
@     newline ()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc h();
_h:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     x := 0;
  mov r0, #0
  str r0, [fp, #-4]
@     f(g() + 3)
  mov r1, fp
  set r0, __thunk_1
  bl _f
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@     proc g() : integer;
_g:
```

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

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     h()
  bl _h
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  ldr r10, [fp, #24]
  bl _g
  add r0, r0, #3
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ End
]]*)
```

5.5 by_name_not_assigned.p

```
(* a test to check that call by name
    parameters are not assigned to *)

proc f(=>x : integer):integer;
begin
    x := 1;
end;
begin
    f(2);
    newline()
end.

(*<<
"test/by_name_not_assigned.p", line 5: x is
    a call by name parameter, and has no
    address
>>*)
(*[[
]]*)
```

5.6 by_name_not_by_ref.p

```
(* a test to check that call by name
    parameters are not passed by reference
    *)

proc f(var x : integer):integer;
begin
    x := 1;
end;
proc g(=> x : integer):integer;
begin
    f(x)
end;
begin
    g(2);
    newline()
end.

(*<<
"test/by_name_not_by_ref.p", line 9: x is a
    call by name parameter, and has no
    address
>>*)
(*[[
]]*)
```

5.7 by_name_not_param.p

```
(* a test to check if functions with call by
     name parameters used as parameters
    leads to errors *)

proc f(proc g(=> y:integer):integer):integer
    ;
begin
    return 0;
end;
begin
    println("hello");
    newline()
end.

(*<<
"test/by_name_not_param.p", line 3:
    Functions that use call by name
    parameters cannot be parameters
>>*)
(*[[
]]*)
```

5.8 by_name_to_by_name.p

```
(* Tests that call by name parameters can be
     passed as call by name parameters *)

proc f(=> x : integer) : integer;
begin
    return x;
end;
proc g(=> x : integer) : integer;
begin
    return f(x);
end;
var x : integer;
begin
    x := 49;
    print_num(g(x));
    newline();
end.

(*<<
49
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x : integer) : integer;
```

```
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     return x;
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc g(=> x : integer) : integer;
_g:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     return f(x);
  mov r1, fp
  set r0, __thunk_1
  bl _f
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     x := 49;
  mov r0, #49
  set r1, _x
  str r0, [r1]
@     print_num(g(x));
  mov r1, fp
  set r0, __thunk_2
  bl _g
  bl print_num
@     newline();
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  ldr r4, [fp, #24]
  ldr r10, [r4, #44]
  ldr r0, [r4, #40]
  blx r0
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_2:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  set r0, _x
  ldr r0, [r0]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _x, 4, 4
@ End
]]*)
```

5.9 by_ref_to_by_name.p

```
(* Tests that call by reference parameters
    can be passed as call by name parameters
     *)

proc f(=> x : integer) : integer;
begin
    return x;
end;
proc g(var x : integer) : integer;
begin
    return f(x);
end;
var x : integer;
begin
    x := 23;
    print_num(g(x));
    newline();
end.

(*<<
23
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x : integer) : integer;
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     return x;
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc g(var x : integer) : integer;
_g:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     return f(x);
  mov r1, fp
  set r0, __thunk_1
  bl _f
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     x := 23;
  set r4, _x
  mov r0, #23
  str r0, [r4]
@     print_num(g(x));
  mov r0, r4
  bl _g
  bl print_num
@     newline();
  bl newline
```

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

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  ldr r0, [fp, #24]
  ldr r0, [r0, #40]
  ldr r0, [r0]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _x, 4, 4
@ End
]]*)
```

5.10 for_continue.p

```
(* A continue statement inside a for loop *)

var i : integer;
begin
    for i := 0 to 100 do
        if i mod 2 = 0 then continue end;
        print_num(i);
        print_char(' ')
    end;
    newline()
end.

(*<<
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
   33 35 37 39 41 43 45 47 49 51 53 55 57
   59 61 63 65 67 69 71 73 75 77 79 81 83
   85 87 89 91 93 95 97 99
>>*)
(*[[
@ 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 100 do
  mov r0, #0
  set r1, _i
  str r0, [r1]
  mov r4, #100
.L3:
  set r5, _i
  ldr r6, [r5]
  cmp r6, r4
  bgt .L4
@         if i mod 2 = 0 then continue end;
  mov r1, #2
  mov r0, r6
  bl int_mod
  cmp r0, #0
  beq .L5
@         print_num(i);
  ldr r0, [r5]
  bl print_num
@         print_char(' ')
  mov r0, #32
  bl print_char
.L5:
  set r5, _i
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L3
.L4:
@     newline()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
@ End
]]*)
```

5.11 given_named_param_test.p

```
(* The test given in the task description
    for named parameters *)
var g: integer;
proc println (x:integer);
begin
    print_num(x);
    newline()
end;
proc p (=> x:integer): integer;
begin
    g := g+1;
    return x+x
end;
begin
    g := 0;
    (* evaluates 2+3 twice and prints out 10
     *)
    println(p(2+3));
    (* g=1 at this point *)
    (* when p needs the value of g, it will
    be equal to 2, so p will return 4 *)
    println(p(g));
    (* g=2 at this point *)
    (* 28 will be printed out *)
    println(p(p(7)));
    (* g=5 at this point *)
    println(g)
end.

(*<<
10
4
28
5
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .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}
  .ltorg

@ proc p (=> x:integer): integer;
_p:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     g := g+1;
  set 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}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     g := 0;
  set r4, _g
  mov r0, #0
  str r0, [r4]
@     println(p(2+3));
  mov r1, fp
  set r0, __thunk_1
  bl _p
  bl _println
@     println(p(g));
  mov r1, fp
  set r0, __thunk_2
  bl _p
  bl _println
@     println(p(p(7)));
  mov r1, fp
  set r0, __thunk_3
  bl _p
  bl _println
@     println(g)
  ldr r0, [r4]
  bl _println
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  mov r0, #5
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_2:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  set r0, _g
  ldr r0, [r0]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_3:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  mov r1, fp
  set r0, __thunk_4
  bl _p
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_4:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  mov r0, #7
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _g, 4, 4
@ End
]]*)
```

5.12 gps_primes.p

```
(* Knuth and Merner general problem solver,
    from ALGOL 60 confidential; original
    text:
"
real procedure GPS(I, N, Z, V); real I, N, Z
    , V;
   begin for I := 1 step 1 until N do Z := V
    ; GPS := 1 end;

Method for finding primes with the gps:
I := GPS(I, if I=0 then -1.0 else I, P, if I
    =1 then 1.0 else
   if GPS(A, I, Z, if A=1 then 1.0 else
      if entier(A)*(entier(I)/entier(A))=
    entier(I) ^ A<I
      then 0.0 else Z) = Z then
      (if P<m then P+1 else I*GPS(A, 1.0, I,
     -1.0)) else P)
"
*)

proc gps(var i : integer; => n : integer;
    var z : integer; => v : integer) :
    integer;
    var limit : integer;
begin
    i := 1;
    while i <= n do
        z := v;
        i := i+1;
    end;
    return 1;
end;

proc f1(var i : integer) : integer;
begin
    if i = 0 then
        return -1;
    else
        return i;
    end;
end;
```

```
proc f2(var a, i : integer; var z: integer)
    : integer;
begin
    if a = 1 then
        return 1;
    else
        if ((a * (i div a) = i) and (a < i))
     then
            return 0;
        else
            return z;
        end;
    end;
end;

proc f3(=> p : integer; var a : integer; =>
    m : integer; var i : integer; var z :
    integer) : integer;
    var tmp_i: integer;
begin
    if i = 1 then
        return 1
    else
        if gps(a, i, z, f2(a, i, z)) = z
    then
            if p < m then
                return p + 1;
            else
                (* This temporary is
    necessary since I can't force the lhs to
     be evaluated before the rhs *)
                tmp_i := i;
                return tmp_i * gps(a, 1, i,
    -1)
            end;
        else
            return p;
        end;
    end;
end;

proc find_prime(m : integer) : integer;
    var i, z, p, a : integer;
begin
    i := gps(i, f1(i), p, f3(p, a, m, i, z))
    ;
    return p;
end;

var i : integer;
begin
    i := 1;
    while i < 20 do
        print_num(find_prime(i)); newline();
        i := i+1;
    end;
    print_num(find_prime(100)); newline();
end.

(*<<
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
541
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc gps(var i : integer; => n : integer;
    var z : integer; => v : integer) :
    integer;
  .text
_gps:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     i := 1;
  mov r0, #1
  ldr r1, [fp, #40]
  str r0, [r1]
.L3:
@     while i <= n do
  ldr r10, [fp, #48]
  ldr r0, [fp, #44]
  blx r0
  ldr r1, [fp, #40]
  ldr r1, [r1]
  cmp r1, r0
  bgt .L5
@         z := v;
  ldr r10, [fp, #60]
  ldr r0, [fp, #56]
  blx r0
  ldr r1, [fp, #52]
  str r0, [r1]
@         i := i+1;
  ldr r5, [fp, #40]
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L3
.L5:
@     return 1;
  mov r0, #1
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc f1(var i : integer) : integer;
_f1:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     if i = 0 then
  ldr r0, [fp, #40]
  ldr r0, [r0]
  cmp r0, #0
  bne .L8
@         return -1;
  mov r0, #-1
  b .L6
```

```
.L8:
@         return i;
  ldr r0, [fp, #40]
  ldr r0, [r0]
.L6:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc f2(var a, i : integer; var z: integer
    ) : integer;
_f2:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     if a = 1 then
  ldr r0, [fp, #40]
  ldr r0, [r0]
  cmp r0, #1
  bne .L12
@         return 1;
  mov r0, #1
  b .L10
.L12:
@         if ((a * (i div a) = i) and (a < i
    )) then
  ldr r0, [fp, #40]
  ldr r4, [r0]
  mov r1, r4
  ldr r0, [fp, #44]
  ldr r0, [r0]
  bl int_div
  ldr r1, [fp, #44]
  ldr r5, [r1]
  mul r0, r4, r0
  cmp r0, r5
  bne .L15
  ldr r0, [fp, #40]
  ldr r0, [r0]
  cmp r0, r5
  bge .L15
@             return 0;
  mov r0, #0
  b .L10
.L15:
@             return z;
  ldr r0, [fp, #48]
  ldr r0, [r0]
.L10:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc f3(=> p : integer; var a : integer;
    => m : integer; var i : integer; var z :
     integer) : integer;
_f3:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     if i = 1 then
  ldr r0, [fp, #60]
  ldr r0, [r0]
  cmp r0, #1
  bne .L20
@         return 1
  mov r0, #1
  b .L18
.L20:
@         if gps(a, i, z, f2(a, i, z)) = z
    then
  str fp, [sp, #4]
  set r0, __thunk_4
  str r0, [sp]
  ldr r3, [fp, #64]
  mov r2, fp
  set r1, __thunk_3
  ldr r0, [fp, #48]
  bl _gps
  ldr r1, [fp, #64]
  ldr r1, [r1]
  cmp r0, r1
  bne .L23
@             if p < m then
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  ldr r10, [fp, #56]
  mov r5, r0
  ldr r0, [fp, #52]
  blx r0
  cmp r5, r0
  bge .L26
@                 return p + 1;
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  add r0, r0, #1
  b .L18
.L26:
@                 tmp_i := i;
  ldr r5, [fp, #60]
  ldr r4, [r5]
@                 return tmp_i * gps(a, 1, i
    , -1)
  str fp, [sp, #4]
  set r0, __thunk_2
  str r0, [sp]
  mov r3, r5
  mov r2, fp
  set r1, __thunk_1
  ldr r0, [fp, #48]
  bl _gps
  mul r0, r4, r0
  b .L18
.L23:
@             return p;
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
.L18:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc find_prime(m : integer) : integer;
_find_prime:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #24
@     i := gps(i, f1(i), p, f3(p, a, m, i, z
    ));
  str fp, [sp, #4]
  set r0, __thunk_6
  str r0, [sp]
  add r3, fp, #-12
  mov r2, fp
  set r1, __thunk_5
```

```
  add r0, fp, #-4
  bl _gps
  str r0, [fp, #-4]
@     return p;
  ldr r0, [fp, #-12]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     i := 1;
  mov r0, #1
  set r1, _i
  str r0, [r1]
.L30:
@     while i < 20 do
  set r4, _i
  ldr r5, [r4]
  cmp r5, #20
  bge .L32
@         print_num(find_prime(i)); newline
    ();
  mov r0, r5
  bl _find_prime
  bl print_num
  bl newline
@         i := i+1;
  ldr r0, [r4]
  add r0, r0, #1
  str r0, [r4]
  b .L30
.L32:
@     print_num(find_prime(100)); newline();
  mov r0, #100
  bl _find_prime
  bl print_num
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  mov r0, #1
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_2:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  mov r0, #-1
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_3:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  ldr r0, [r0, #60]
  ldr r0, [r0]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_4:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r4, [fp, #24]
  ldr r2, [r4, #64]
  ldr r1, [r4, #60]
  ldr r0, [r4, #48]
  bl _f2
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_5:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  add r0, r0, #-4
  bl _f1
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_6:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #16
  ldr r4, [fp, #24]
  add r0, r4, #-8
  str r0, [sp, #8]
  add r0, r4, #-4
  str r0, [sp, #4]
  str fp, [sp]
  set r3, __thunk_8
  add r2, r4, #-16
  mov r1, fp
  set r0, __thunk_7
  bl _f3
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_7:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  ldr r0, [r0, #24]
  ldr r0, [r0, #-12]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_8:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  ldr r0, [r0, #24]
  ldr r0, [r0, #40]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
@ End
]]*)
```

5.13 jensen1.p

```
(* Jensen's device, simple application,
    using local variables*)

proc test();
```

```
    var i : integer;
    var a: array 10 of integer;

    proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
        var retval: integer;
    begin
        retval := 0;
        i := left;
        while i <= right do
            retval := retval + v;
            i := i+1;
        end;
        return retval;
    end;

begin
    a[0] := 0;
    a[1] := 1;
    a[2] := 2;
    a[3] := 3;
    a[4] := 4;
    a[5] := 5;
    a[6] := 6;
    a[7] := 7;
    a[8] := 8;
    a[9] := 9;
    print_num(sum(i, 0, 9, a[i])); newline()
    ;
end;
begin
    test();
end.

(*<<
45
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc test();
  .text
_test:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #48
@     a[0] := 0;
  mov r0, #0
  str r0, [fp, #-44]
@     a[1] := 1;
  mov r0, #1
  str r0, [fp, #-40]
@     a[2] := 2;
  mov r0, #2
  str r0, [fp, #-36]
@     a[3] := 3;
  mov r0, #3
  str r0, [fp, #-32]
@     a[4] := 4;
  mov r0, #4
  str r0, [fp, #-28]
@     a[5] := 5;
  mov r0, #5
  str r0, [fp, #-24]
@     a[6] := 6;
  mov r0, #6
  str r0, [fp, #-20]
@     a[7] := 7;
  mov r0, #7
  str r0, [fp, #-16]
@     a[8] := 8;
  mov r0, #8
  str r0, [fp, #-12]
@     a[9] := 9;
  mov r0, #9
  str r0, [fp, #-8]
@     print_num(sum(i, 0, 9, a[i])); newline
    ();
  str fp, [sp]
  set r3, __thunk_1
  mov r2, #9
  mov r1, #0
  add r0, fp, #-4
  mov r10, fp
  bl _sum
  bl print_num
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@     proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
_sum:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@         retval := 0;
  mov r4, #0
@         i := left;
  ldr r0, [fp, #44]
  ldr r1, [fp, #40]
  str r0, [r1]
.L4:
@         while i <= right do
  ldr r0, [fp, #40]
  ldr r0, [r0]
  ldr r1, [fp, #48]
  cmp r0, r1
  bgt .L6
@             retval := retval + v;
  ldr r10, [fp, #56]
  ldr r0, [fp, #52]
  blx r0
  add r4, r4, r0
@             i := i+1;
  ldr r5, [fp, #40]
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L4
.L6:
@         return retval;
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

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

__thunk_1:
```

```
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  ldr r4, [fp, #24]
  add r0, r4, #-44
  ldr r1, [r4, #-4]
  lsl r1, r1, #2
  add r0, r0, r1
  ldr r0, [r0]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ End
]]*)
```

5.14 jensen2.p

```
(* Jensen's device, more involved
    application, using global variables*)

var i : integer;

var a: array 10 of integer;

proc sum(var i: integer; left, right:integer
    ; => v: integer): integer;
    var retval: integer;
begin
    retval := 0;
    i := left;
    while i <= right do
        retval := retval + v;
        i := i+1;
    end;
    return retval;
end;

proc print_and_ret(x : integer): integer;
begin
    print_num(x);
    newline();
    return x;
end;

begin
    a[0] := 0;
    a[1] := 1;
    a[2] := 2;
    a[3] := 3;
    a[4] := 4;
    a[5] := 5;
    a[6] := 6;
    a[7] := 7;
    a[8] := 8;
    a[9] := 9;
    print_num(sum(i, 0, 9, print_and_ret(a[i
    ] * a[i]))); newline();
end.

(*<<
0
1
4
9
16
25
36
49
64
81
285
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
  .text
_sum:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     retval := 0;
  mov r4, #0
@     i := left;
  ldr r0, [fp, #44]
  ldr r1, [fp, #40]
  str r0, [r1]
.L3:
@     while i <= right do
  ldr r0, [fp, #40]
  ldr r0, [r0]
  ldr r1, [fp, #48]
  cmp r0, r1
  bgt .L5
@         retval := retval + v;
  ldr r10, [fp, #56]
  ldr r0, [fp, #52]
  blx r0
  add r4, r4, r0
@         i := i+1;
  ldr r5, [fp, #40]
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L3
.L5:
@     return retval;
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc print_and_ret(x : integer): integer;
_print_and_ret:
  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
@     return x;
  ldr r0, [fp, #40]
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     a[0] := 0;
  set r4, _a
  mov r0, #0
```

```
  str r0, [r4]
@     a[1] := 1;
  mov r0, #1
  str r0, [r4, #4]
@     a[2] := 2;
  mov r0, #2
  str r0, [r4, #8]
@     a[3] := 3;
  mov r0, #3
  str r0, [r4, #12]
@     a[4] := 4;
  mov r0, #4
  str r0, [r4, #16]
@     a[5] := 5;
  mov r0, #5
  str r0, [r4, #20]
@     a[6] := 6;
  mov r0, #6
  str r0, [r4, #24]
@     a[7] := 7;
  mov r0, #7
  str r0, [r4, #28]
@     a[8] := 8;
  mov r0, #8
  str r0, [r4, #32]
@     a[9] := 9;
  mov r0, #9
  str r0, [r4, #36]
@     print_num(sum(i, 0, 9, print_and_ret(a
    [i] * a[i]))); newline();
  str fp, [sp]
  set r3, __thunk_1
  mov r2, #9
  mov r1, #0
  set r0, _i
  bl _sum
  bl print_num
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  set r0, _a
  set r1, _i
  ldr r1, [r1]
  lsl r1, r1, #2
  add r0, r0, r1
  ldr r4, [r0]
  mul r0, r4, r4
  bl _print_and_ret
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
  .comm _a, 40, 4
@ End
]]*)
```

5.15 jensen2d.p

```
(* Jensen's device, 2d application, using
    local variables*)

proc test();
    var i, j: integer;
    var a: array 10 of array 10 of integer;

    proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
        var retval: integer;
    begin
        retval := 0;
        i := left;
        while i <= right do
            retval := retval + v;
            i := i+1;
        end;
        return retval;
    end;
begin
    i := 0;
    while i < 10 do
        j := 0;
        while j < 10 do
            a[i][j] := i * j;
            j := j + 1;
        end;
        i := i+1;
    end;

    print_num(sum(i, 0, 9, sum(j, 0, 9, a[i
    ][j]))); newline();
end;
begin
    test();
end.

(*<<
2025
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc test();
  .text
_test:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #416
@     i := 0;
  mov r0, #0
  str r0, [fp, #-4]
.L3:
@     while i < 10 do
  ldr r0, [fp, #-4]
  cmp r0, #10
  bge .L5
@         j := 0;
  mov r0, #0
  str r0, [fp, #-8]
.L6:
@         while j < 10 do
  ldr r4, [fp, #-8]
  cmp r4, #10
  bge .L8
@             a[i][j] := i * j;
  ldr r5, [fp, #-4]
  mul r0, r5, r4
  add r1, fp, #-408
  mov r2, #40
  mul r2, r5, r2
  add r1, r1, r2
```

```
  lsl r2, r4, #2
  add r1, r1, r2
  str r0, [r1]
@             j := j + 1;
  ldr r0, [fp, #-8]
  add r0, r0, #1
  str r0, [fp, #-8]
  b .L6
.L8:
@         i := i+1;
  ldr r0, [fp, #-4]
  add r0, r0, #1
  str r0, [fp, #-4]
  b .L3
.L5:
@     print_num(sum(i, 0, 9, sum(j, 0, 9, a[
    i][j]))); newline();
  str fp, [sp]
  set r3, __thunk_1
  mov r2, #9
  mov r1, #0
  add r0, fp, #-4
  mov r10, fp
  bl _sum
  bl print_num
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@     proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
_sum:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@         retval := 0;
  mov r4, #0
@         i := left;
  ldr r0, [fp, #44]
  ldr r1, [fp, #40]
  str r0, [r1]
.L10:
@         while i <= right do
  ldr r0, [fp, #40]
  ldr r0, [r0]
  ldr r1, [fp, #48]
  cmp r0, r1
  bgt .L12
@             retval := retval + v;
  ldr r10, [fp, #56]
  ldr r0, [fp, #52]
  blx r0
  add r4, r4, r0
@             i := i+1;
  ldr r5, [fp, #40]
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L10
.L12:
@         return retval;
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

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

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@
  ldr r4, [fp, #24]
  str fp, [sp]
  set r3, __thunk_2
  mov r2, #9
  mov r1, #0
  add r0, r4, #-8
  mov r10, r4
  bl _sum
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_2:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  ldr r0, [fp, #24]
  ldr r4, [r0, #24]
  add r0, r4, #-408
  ldr r1, [r4, #-4]
  mov r2, #40
  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}
  .ltorg

@ End
]]*)
```

5.16 jensen3.p

```
(* Jensen's device, complicated application,
     with global variables*)

var i : integer;

var a : array 10 of integer;

proc sum(var i: integer; left, right:integer
    ; => v: integer): integer;
    var retval: integer;
begin
    retval := 0;
    i := left;
    while i <= right do
        retval := retval + v;
        i := i+1;
    end;
    return retval;
end;

proc return_a_i_and_increment_i(): integer;
    var retval: integer;
begin
    retval := a[i];
    i := i+1;
```

```
    return retval;
end;


begin
    a[0] := 0;
    a[1] := 1;
    a[2] := 2;
    a[3] := 3;
    a[4] := 4;
    a[5] := 5;
    a[6] := 6;
    a[7] := 7;
    a[8] := 8;
    a[9] := 9;
    print_num(sum(i, 0, 9,
    return_a_i_and_increment_i())); newline
    ();
end.

(*<<
20
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
  .text
_sum:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     retval := 0;
  mov r4, #0
@     i := left;
  ldr r0, [fp, #44]
  ldr r1, [fp, #40]
  str r0, [r1]
.L3:
@     while i <= right do
  ldr r0, [fp, #40]
  ldr r0, [r0]
  ldr r1, [fp, #48]
  cmp r0, r1
  bgt .L5
@         retval := retval + v;
  ldr r10, [fp, #56]
  ldr r0, [fp, #52]
  blx r0
  add r4, r4, r0
@         i := i+1;
  ldr r5, [fp, #40]
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L3
.L5:
@     return retval;
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ proc return_a_i_and_increment_i(): integer
    ;
_return_a_i_and_increment_i:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     retval := a[i];
  set r5, _i
  ldr r6, [r5]
  set r0, _a
  lsl r1, r6, #2
  add r0, r0, r1
  ldr r4, [r0]
@     i := i+1;
  add r0, r6, #1
  str r0, [r5]
@     return retval;
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     a[0] := 0;
  set r4, _a
  mov r0, #0
  str r0, [r4]
@     a[1] := 1;
  mov r0, #1
  str r0, [r4, #4]
@     a[2] := 2;
  mov r0, #2
  str r0, [r4, #8]
@     a[3] := 3;
  mov r0, #3
  str r0, [r4, #12]
@     a[4] := 4;
  mov r0, #4
  str r0, [r4, #16]
@     a[5] := 5;
  mov r0, #5
  str r0, [r4, #20]
@     a[6] := 6;
  mov r0, #6
  str r0, [r4, #24]
@     a[7] := 7;
  mov r0, #7
  str r0, [r4, #28]
@     a[8] := 8;
  mov r0, #8
  str r0, [r4, #32]
@     a[9] := 9;
  mov r0, #9
  str r0, [r4, #36]
@     print_num(sum(i, 0, 9,
    return_a_i_and_increment_i())); newline
    ();
  str fp, [sp]
  set r3, __thunk_1
  mov r2, #9
  mov r1, #0
  set r0, _i
  bl _sum
  bl print_num
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
```

```
  mov fp, sp
@
  bl _return_a_i_and_increment_i
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
  .comm _a, 40, 4
@ End
]]*)
```

5.17 jensen_function.p

```
(* Jensen's device, functional application
    *)

var i : integer;

proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
    var retval: integer;
begin
    retval := 0;
    i := left;
    while i <= right do
        retval := retval + v;
        i := i+1;
    end;
    return retval;
end;

begin
    print_num(sum(i, 0, 9, i * i * i * i));
    newline();
end.

(*<<
15333
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc sum(var i: integer; left, right:
    integer; => v: integer): integer;
  .text
_sum:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     retval := 0;
  mov r4, #0
@     i := left;
  ldr r0, [fp, #44]
  ldr r1, [fp, #40]
  str r0, [r1]
.L3:
@     while i <= right do
  ldr r0, [fp, #40]
  ldr r0, [r0]
  ldr r1, [fp, #48]
  cmp r0, r1
  bgt .L5
@         retval := retval + v;
  ldr r10, [fp, #56]
  ldr r0, [fp, #52]
  blx r0
  add r4, r4, r0
@         i := i+1;
  ldr r5, [fp, #40]
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L3
.L5:
@     return retval;
  mov r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  sub sp, sp, #8
@     print_num(sum(i, 0, 9, i * i * i * i))
    ; newline();
  str fp, [sp]
  set r3, __thunk_1
  mov r2, #9
  mov r1, #0
  set r0, _i
  bl _sum
  bl print_num
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  set r0, _i
  ldr r4, [r0]
  mul r0, r4, r4
  mul r0, r0, r4
  mul r0, r0, r4
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
@ End
]]*)
```

5.18 lazy_evaluate_by_name.p

```
(* Tests that call by name parameters are
    only evaluated when needed *)

proc f(=> x : integer);
begin
    print_string("hello");
    newline();
end;

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

var x : integer;
begin
    x := 0;
    f(h(x));
    print_num(x);
    newline()
```

```
end.

(*<<
hello
0
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x : integer);
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     print_string("hello");
  mov r1, #5
  set r0, g2
  bl print_string
@     newline();
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

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

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     x := 0;
  set r4, _x
  mov r0, #0
  str r0, [r4]
@     f(h(x));
  mov r1, fp
  set r0, __thunk_1
  bl _f
@     print_num(x);
  ldr r0, [r4]
  bl print_num
@     newline()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  set r0, _x
  bl _h
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _x, 4, 4
  .data
g2:
  .byte 104, 101, 108, 108, 111
  .byte 0
@ End
]]*)
```

5.19 multiple_continues.p

```
(* multiple continue statements in a single
    loop *)

var i : integer;
begin
    for i := 1 to 20 do;
        if i mod 2 = 0 then continue else
    end;
        if i mod 3 = 0 then continue else
    end;
        if i mod 5 = 0 then continue else
    end;
        print_num(i);
        print_char(' ')
    end;
    newline()
end.

(*<<
1 7 11 13 17 19
>>*)
(*[[
@ 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 := 1 to 20 do;
  mov r0, #1
  set r1, _i
  str r0, [r1]
  mov r4, #20
.L3:
  set r5, _i
  ldr r6, [r5]
  cmp r6, r4
  bgt .L4
@         if i mod 2 = 0 then continue else
    end;
  mov r1, #2
  mov r0, r6
  bl int_mod
  cmp r0, #0
  beq .L5
@         if i mod 3 = 0 then continue else
    end;
  mov r1, #3
  ldr r0, [r5]
  bl int_mod
  cmp r0, #0
  beq .L5
@         if i mod 5 = 0 then continue else
```

```
    end;
  mov r1, #5
  ldr r0, [r5]
  bl int_mod
  cmp r0, #0
  beq .L5
@         print_num(i);
  ldr r0, [r5]
  bl print_num
@         print_char(' ')
  mov r0, #32
  bl print_char
.L5:
  set r5, _i
  ldr r0, [r5]
  add r0, r0, #1
  str r0, [r5]
  b .L3
.L4:
@     newline()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
@ End
]]*)
```

5.20 nested_continue.p

```
(* a test designed to make sure that we jump
     to the end of the correct loop *)

var i, j : integer;
begin
    for i := 1 to 4 do;
        if i mod 3 = 0 then continue end;
        for j := 1 to 4 do;
            if (i + j) mod 2 = 0 then
    continue end;
            print_num(i);
            print_char(' ');
            print_num(j);
            newline()
        end
    end
end.

(*<<
1 2
1 4
2 1
2 3
4 1
4 3
>>*)
(*[[
@ 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 := 1 to 4 do;
  mov r0, #1
  set r1, _i
  str r0, [r1]
  mov r5, #4
.L3:
  set r0, _i
  ldr r6, [r0]
  cmp r6, r5
  bgt .L2
@         if i mod 3 = 0 then continue end;
  mov r1, #3
  mov r0, r6
  bl int_mod
  cmp r0, #0
  beq .L5
@         for j := 1 to 4 do;
  mov r0, #1
  set r1, _j
  str r0, [r1]
  mov r4, #4
.L9:
  set r6, _j
  ldr r7, [r6]
  cmp r7, r4
  bgt .L5
@             if (i + j) mod 2 = 0 then
    continue end;
  set r8, _i
  mov r1, #2
  ldr r0, [r8]
  add r0, r0, r7
  bl int_mod
  cmp r0, #0
  beq .L11
@             print_num(i);
  ldr r0, [r8]
  bl print_num
@             print_char(' ');
  mov r0, #32
  bl print_char
@             print_num(j);
  ldr r0, [r6]
  bl print_num
@             newline()
  bl newline
.L11:
  set r6, _j
  ldr r0, [r6]
  add r0, r0, #1
  str r0, [r6]
  b .L9
.L5:
  set r6, _i
  ldr r0, [r6]
  add r0, r0, #1
  str r0, [r6]
  b .L3
.L2:
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
  .comm _j, 4, 4
@ End
]]*)
```

5.21 reevaluate_by_name.p

```
(* Tests that call by name parameters are re
    -evaluated each time they are used *)

proc f(=> x : integer);
begin
```

```
    print_num(x);
    newline();
    print_num(x);
    newline();
end;

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

var x : integer;
begin
    x := 0;
    f(h(x));
end.

(*<<
1
2
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x : integer);
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r1}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     print_num(x);
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  bl print_num
@     newline();
  bl newline
@     print_num(x);
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  bl print_num
@     newline();
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

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

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     x := 0;
  mov r0, #0
  set r1, _x
  str r0, [r1]
@     f(h(x));
  mov r1, fp
  set r0, __thunk_1
  bl _f
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  set r0, _x
  bl _h
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _x, 4, 4
@ End
]]*)
```

5.22 repeat_continue.p

```
(* A continue statement inside a while loop
    *)

var i: integer;
begin
    i := 0;
    repeat
        i := i+1;
        if i mod 2 = 0 then continue end;
        print_num(i);
        print_char(' ')
    until i >= 100;
    newline()
end.

(*<<
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
   33 35 37 39 41 43 45 47 49 51 53 55 57
   59 61 63 65 67 69 71 73 75 77 79 81 83
   85 87 89 91 93 95 97 99
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

  .text
pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     i := 0;
  mov r0, #0
  set r1, _i
  str r0, [r1]
.L3:
@         i := i+1;
  set r4, _i
  ldr r0, [r4]
  add r5, r0, #1
  str r5, [r4]
```

```
@         if i mod 2 = 0 then continue end;
  mov r1, #2
  mov r0, r5
  bl int_mod
  cmp r0, #0
  beq .L5
@         print_num(i);
  ldr r0, [r4]
  bl print_num
@         print_char(' ')
  mov r0, #32
  bl print_char
.L5:
  set r0, _i
  ldr r0, [r0]
  cmp r0, #100
  blt .L3
@     newline()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
@ End
]]*)
```

5.23 rightarrow_non_int_error.p

```
(* a test to check that only integers can be
     call by name parameters *)

type arr = array 10 of integer;

proc f(=> x : arr) : integer;
begin
    return x[2];
end;
var x : arr;
begin
    x[2] := 1;
    print_num(f(x));
    newline()
end.

(*<<
"test/rightarrow_non_int_error.p", line 5:
    Call by name parameter must be an
    integer
>>*)
(*[[
]]*)
```

5.24 rightarrow_same_as_var.p

```
proc f(=> x, y : integer) : integer;
begin
    return x + y;
end;
begin
    print_num(f(1, 2));
    newline()
end.

(*<<
3
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

@ proc f(=> x, y : integer) : integer;
  .text
_f:
  mov ip, sp
  stmfd sp!, {r0-r3}
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     return x + y;
  ldr r10, [fp, #44]
  ldr r0, [fp, #40]
  blx r0
  ldr r10, [fp, #52]
  mov r4, r0
  ldr r0, [fp, #48]
  blx r0
  add r0, r4, r0
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     print_num(f(1, 2));
  mov r3, fp
  set r2, __thunk_2
  mov r1, fp
  set r0, __thunk_1
  bl _f
  bl print_num
@     newline()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_1:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@
  mov r0, #1
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

__thunk_2:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
  mov r0, #2
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

@ End
]]*)
```

5.25 while_continue.p

```
(* A continue statement inside a while loop
    *)

var i: integer;
begin
    i := 0;
    while i < 100 do
        i := i+1;
        if i mod 2 = 0 then continue end;
        print_num(i);
        print_char(' ')
    end;
```

```
    newline()
end.

(*<<
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
    33 35 37 39 41 43 45 47 49 51 53 55 57
    59 61 63 65 67 69 71 73 75 77 79 81 83
    85 87 89 91 93 95 97 99
>>*)
(*[[
@ picoPascal compiler output
  .include "fixup.s"
  .global pmain

  .text
pmain:
  mov ip, sp
  stmfd sp!, {r4-r10, fp, ip, lr}
  mov fp, sp
@     i := 0;
  mov r0, #0
  set r1, _i
  str r0, [r1]
.L3:
@     while i < 100 do
  set r4, _i
  ldr r5, [r4]
  cmp r5, #100
  bge .L5
@         i := i+1;
  add r5, r5, #1
  str r5, [r4]
@         if i mod 2 = 0 then continue end;
  mov r1, #2
  mov r0, r5
  bl int_mod
  cmp r0, #0
  beq .L3
@         print_num(i);
  ldr r0, [r4]
  bl print_num
@         print_char(' ')
  mov r0, #32
  bl print_char
  b .L3
.L5:
@     newline()
  bl newline
  ldmfd fp, {r4-r10, fp, sp, pc}
  .ltorg

  .comm _i, 4, 4
@ End
]]*)
```

6 Sources

Sources used:

- The course book by M. Spivey
- The recommended book: Understanding and Writing Compilers by R. Bornat
- `https://en.wikipedia.org/wiki/Evaluation_strategy` for general information about evaluation strategies.
- `https://en.wikipedia.org/wiki/Thunk` for information about thunks.
- `https://en.wikipedia.org/wiki/Jensen%27s_Device` for a description of Jensen's device, and for a description of Knuth and Merner's general problem solver.
- `https://cseweb.ucsd.edu/~goguen/courses/230w02/GPS.html` for a fuller description of Knuth and Merner's general problem solver.
- ALGOL 60 confidential, by D. Knuth and J. Merner, doi 10.1145/366573.366599

Contents

1 Task **1**

2 Continue statements **1**
2.1 Lexical analysis . . . 1
2.2 Abstract syntax . . . 1
2.3 Parsing . . . 1
2.4 Semantic analysis . . . 2
2.5 Intermediate code generation . . . 2
2.6 Machine code generation . . . 3
2.7 Testing . . . 4

3 Call by name semantics **4**
3.1 Lexical analysis . . . 5
3.2 Abstract Syntax . . . 5
3.3 Parsing . . . 5
3.4 Semantic Analysis . . . 5
3.5 Intermediate code generation . . . 7
3.6 Machine code generation . . . 9
3.7 Testing . . . 9

4 Appendix A **10**

5 Appendix B **16**
5.1 **bare_continue.p** . . . 16
5.2 **by_name_iterator.p** . . . 17
5.3 **by_name_local.p** . . . 19
5.4 **by_name_nested_function.p** . . . 20
5.5 **by_name_not_assigned.p** . . . 21
5.6 **by_name_not_by_ref.p** . . . 21
5.7 **by_name_not_param.p** . . . 21
5.8 **by_name_to_by_name.p** . . . 21
5.9 **by_ref_to_by_name.p** . . . 22
5.10 **for_continue.p** . . . 23
5.11 **given_named_param_test.p** . . . 23
5.12 **gps_primes.p** . . . 24
5.13 **jensen1.p** . . . 28
5.14 **jensen2.p** . . . 29
5.15 **jensen2d.p** . . . 30
5.16 **jensen3.p** . . . 31
5.17 **jensen_function.p** . . . 33
5.18 **lazy_evaluate_by_name.p** . . . 33
5.19 **multiple_continues.p** . . . 34
5.20 **nested_continue.p** . . . 35
5.21 **reevaluate_by_name.p** . . . 36

5.22 repeat_continue.p 36
5.23 rightarrow_non_int_error.p 37
5.24 rightarrow_same_as_var.p 37
5.25 while_continue.p 38

6 Sources **39**