Compilers Assignment - Open Arrays

Candidate Number: 1074667

January 2024

Contents

1 Introduction **3**

2 Implementation **4**
2.1 Allowing Open Arrays as Formal Parameters . . . 4
2.2 Implementing the Len function . . . 5
2.3 Implementing Pointers to Open Arrays . . . 6
2.4 Minor Quality of Life Improvement . . . 7
2.5 Avoiding Repetition of Side Effects . . . 7
2.6 Slices . . . 8

3 Code **11**
3.1 check . . . 11
3.2 dict . . . 13
3.3 optree . . . 16
3.4 parser . . . 17
3.5 pas0 . . . 18
3.6 tgen . . . 18
3.7 tran . . . 22
3.8 tree . . . 23

4 Testing **25**
4.1 Open Arrays . . . 25
4.2 Nesting 1 . . . 27
4.3 Nesting 2 . . . 29
4.4 Pointers . . . 31
4.5 Slices . . . 34
4.6 Nesting 3 . . . 40
4.7 Bounds Checking . . . 42
4.8 Avoiding Repeated Evaluation 1 . . . 46
4.9 Avoiding Repeated Evaluation 2 . . . 48

1 Introduction

The assignment was concentrated around the implementation of open arrays – these are arrays whose size are not specified in their definition. The main areas in which they appeared were as a formal parameter for a procedure, allowing the procedure to work for arrays of arbitrary size, or for a pointer to an open array, which could be initialised to point to an array of any size. The assignment also required me to deal with slices, where I permitted specified, contiguous chunks of an array to be passed into a procedure and treated as an open array parameter within the scope of the procedure. There were some other aspects of the compiler that were also improved, such as allowing print_string to work for open arrays and improving bounds checking procedures.

Note: I made the design decision that a procedure that expects an array of fixed size will not accept an open array, even if it is of the correct size - I felt this best fits with the strongly typed nature of Pascal.

What follows in this document is an explanation of the changes made to the compiler. These are adding open arrays as formal parameters, by extending the abstract syntax and modifying all the checking and code generation functions to appropriately respond to them; implementing the "len" function in a way that it works for normal arrays, open arrays, and later, the dereferenced values of pointers to open arrays; implementing pointers to open arrays; editing some other functions to accept open arrays as well as normal arrays; making bounds checking and supplying values for open array parameters robust by ensuring they do not duplicate side effects and implementing slices by again extending the abstract syntax and intermediate functions, but focusing on their unique code generation and address arithmetic, while ensuring necessary validity checks work well for them.

You will then find the code listing, clearly showing all the modifications made to the compiler that were described in the above section.

Finally, there is a series of tests. These demonstrate both the correctness of the solution and the efficiency of the object code that it produces.

2 Implementation

Please note that the blue hyperlinks below will take you to the corresponding code extracts in the code listing (section 3).

2.1 Allowing Open Arrays as Formal Parameters

Initially, I wanted to be able to still use the typexpr Array for open arrays, but give them a special size that would signify that they weren't normal arrays. I tried to do this by adding a new entry in expr_guts (in tree.ml) called "OpenArrayNilSize", and then exploiting the "make_expr" function to initialise an Array with this as the size parameter.

This construction later proved problematic as it became awkward for functions to tell if they were dealing with a normal array, whose size was in its definition, or if they were dealing with an open array, whose size was stored at some offset to its base address.

Therefore, I opted to scrap this approach, and instead edited the grammar to have a new type altogether and not rely on the Array construction that was already in place. No modifications to the lexer were required, as all the necessary symbols have already been accounted for.

I added the new patterns to parser.mly, so that an "Array of typexpr" was recognised as the new openArray, and that formal_decl allows for these new open arrays - permitting them to be recognised as formal parameters. (parser)

To allow for these, I had to add a new type "OpenArray" as a typexpr, so that open arrays behave like their array counterparts (tree). I also added a new type_guts "OpenArrayType" and a new def_kind "OpenArrayParamDef" - these enabled me to formally define open array types and allowed the compiler to recognise them as their own type of parameter - this proves important as open array parameters require the address of the array to be passed in, alongside the base address of the array, so need to be handled in their own specific way, not dissimilar to handling procedure parameters that need an address and a static link. (dict)

I then made some changes to some minor functions, allowing base_type, same_type and is_string to process open arrays in a similar way that they process normal arrays. (dict)

I then edited the check functions, allowing OpenArrayParamDef to be a valid object for use in expressions and I modified param_size to allocate a space value of 2 to them - this is because, as stated above, I need to pass in both the base address of the array and its size - this is similar to higher order functions that require both the address and static link to be passed in for each function parameter - these are denoted by PParamDef. A few tweaks were needed to check_var and do_alloc to allow them to work with the new parameter type. I also edited check_typexpr to ensure that arrays can't contain open arrays and open arrays can't contain arrays (neither fixed nor open) - this was to uphold the restrictions given in the specification. Notice the importance of calling check_typexpr on the argument of OpenArray. This crucially annotates its tree for later use. check_typexpr also had to be able to return the type of an open array expression. So I defined a new function, open_array_row which made use of mk_rowtype in initialising a new open array type. This worked in a similar fashion to how arrays used row, but here the compiler only had to reserve space for an address type as an open array is simply a base address for some array somewhere.(dict)(check)

I then changed check_arg to check that the type of array that gets passed in as an open array parameter is what was expected. For example, an open array of integers must be passed some sort of array of integers (check).

I then had to change expr_type to allow the compiler to correctly return the type of subscripts for open arrays, this is easily done by extracting the "ptype" from its definition. (check)

It was then onto code generation. gen_addr had to be edited to allow for finding the address of this new type of formal parameter. This is the same approach as for VParamDef parameters, as I am treating all open arrays as vars. Then gen_arg was changed so that it would provide the base address and the array size for open array formal parameters. At this moment in time, the only things that could ever get passed into a procedure with an open array formal parameter were normal arrays, or other open arrays which themselves were formal parameters (this later gets changed to deal dereferencing with pointers to open arrays). The former was easy to manage as its size is in its type definition, and I could make use of the bound function. For the latter, I made use of the gen_closure function, which was appropriately adjusted to deal with the new type of parameter. The approach is very similar to how gen_closure is used for higher order functions, in the way that it places the address of both the procedure parameter, as well as its static link, on the stack, in a specified order. Here it places the size of the open array at an offset of +4 to the base address. (tgen)

2.2 Implementing the Len function

The "len" function needed to return the length of any array passed into it, whether an array of fixed size, an open array parameter or a pointer to an open array. This is easy for arrays of fixed size as their size is part of their definition. For open arrays which are parameters, this value appeared at an offset of +4, and for pointers to open arrays, this appeared at an offset of -4 (as shown in the implementation of pointers to open arrays further on). I will discuss the implementation of "len" for pointers to open arrays during the implementation of pointers further on in the document.

The approach I took was to define len as a new library function. This first required me to give it a type (libid) and a name ("len"). (dict)

I then had to add it to the initial environment (init_env), so that programs would always have access to it. It takes a single parameter and returns an integer. I then edited check_libcall to ensure that the argument passed into it is actually an array of some sort. Calling check_expr on the argument supplied not only helped the compiler extract its type, but also annotated its tree for later use. (check)

This involved using the is_array and is_open_array - the latter being a newly defined function that just mirrored is_array. (dict)

Then it was onto code generation. Initially, in gen_libcall, by matching on e_type.t_guts, if it was an ArrayType, I could just extract its size from its definition and return it as a ⟨CONST n⟩. If it was an OpenArrayType, the compiler needed to find its address location and then load from an offset. If it was an open array parameter, the offset was +4. If it wasn't, then since it was an OpenArrayType, it must be a pointer to an open array (by process of elimination), so had an offset of -4 (details in the next section). I later decided to move this code into a function called "size_of_array" as it proved useful to be able to call this procedure for other parts and made the function neater. So now, len simply calls this function. (tgen)

2.3 Implementing Pointers to Open Arrays

The next part involved allowing the creation of pointers to open arrays. For example, allowing "var p: pointer to array of integer", where the array size was reserved with a new library function "newrow" which took a pointer to an open array and an integer as its two parameters, the latter defining how much space should be reserved. (check)

I started by defining newrow as a library function, giving it a libid, a lib_name and adding it to the initial environment. This is an identical approach to adding the len function. (dict) (check)

I then updated check_libcall to deal with the new procedure - it needed to check that its first argument is a pointer to an open array (I made the design choice that for a pointer to an array of fixed size: "var p: pointer to array n of integer", even "newrow(p,n);" would not be valid), and that its second argument is an integer value (using same_type). Calling check_expr annotated the tree of the first argument as necessary. (check)

For its code generation, inside gen_libcall (tgen), I first made use of a new function "deref_chain". This allowed the compiler to successfully manipulate chains of pointers, so it could work with an arbitrarily long chain of pointers, e.g. "pointer to a pointer to ... to pointer to an array of integer". This is a recursive function that just traverses the list of pointers (tree). Then I made a call to the provided C function "palloc2" - as described, this saves the length of the array in the first word and returns a pointer to the second word - **it is important to note, that while open array parameters stored their length at an**

offset of +4, pointers to open arrays stored theirs at an offset of -4. This often meant that they needed to be treated differently in order to accurately extract their size at the correct offset.

I then had to update the parser's context-free grammar to allow for pointers to open arrays to be allowed within the syntax. This was easily done by adding "POINTER TO openArray" where openArray is the type I defined in my initial construction. (parser)

I then realised I had to tweak a few functions due to the different placement of the length value for open parameters and pointers to open arrays. I edited gen_arg to allow the compiler to pass in the value of a pointer to an open array (via dereferencing it) to a procedure expecting an open array (so as an open array parameter). I had to treat this as a separate case to passing in an open array parameter due to not wanting to repeat side effects, so using gen_addr_bound (implementation explained in next section) instead of gen_closure. I then edited the code generation of size_of_array for the same reason, treating this as a separate case (I also made use of the "deref_chain" function again to deal with finding the length of the final value in the chain of many pointers to an open array). (tgen)

Some other areas that needed updating to deal with these new pointers included "same_type" (so that pointers to open arrays were considered appropriately) and tweaking bounds checking (this is updated significantly in the section after next). (dict)

2.4 Minor Quality of Life Improvement

I improved "print_string" to work for open arrays - this used the previously defined size_of_array function for open arrays. Since this function can handle both pointers to open arrays and open array parameters, they didn't need to be treated as separate cases here. I kept normal arrays as a separate case purely to generate as efficient code as possible. (tgen)

2.5 Avoiding Repetition of Side Effects

As the assignment states, in the current implementation, statements of the form $e_1 \uparrow [e_2]$ are problematic when bounds checking is enabled, or statements of the form $e_1 \uparrow$, where e_1 was a pointer and gets passed as an open array parameter, as e_1 gets evaluated twice - its value is needed for both the bound/size of the array and for the base address. Evaluating e_1 twice becomes an issue when it is a statement that causes side effects, as currently this would cause them to be doubled. To allow the repeated use of e_1's value without evaluating it twice, it can be evaluated it into a temporary register which is used more than once. This is done with the construction suggested, making use of the "AFTER" node to initialise the "DEFTEMP" and "TEMP" nodes.

I wrote an additional function "gen_addr_bound" to help achieve this by returning the operator trees for the base address and for the bound. (tgen)

For consistency, this function will always run when bounds checking is enabled, even when pointers aren't involved. It first reserves a temporary register and then pattern matches on the type of array. In the case that it is a normal array, extracting its base address and size are easy enough, especially as the latter is in its definition. If it is an open array, it pattern matches again to determine if it is a function parameter (so its size is at an offset of +4), or if it is the value of dereferencing some pointer (so its size is at an offset of -4). For normal arrays, since the base address is not needed to extract its size, the bound tree is simply the size from its definition and the base address is simply the result of gen_addr, so no need to utilise any temporary registers. **deref_chain gets used again to deal with arbitrarily long chains of pointers.** For open arrays, both as parameters and pointers, their base address gets loaded into a temporary register, and put within the ⟨AFTER⟩ node. This ensures that their address gets stored in the temporary register before it is needed. They then have their size loaded into the bound tree. This makes use of the stored base address, and slightly different sequences of LOADW instructions for each case. (tgen)

This function gets utilised for bounds checking in gen_addr, where the compiler pattern matches with Sub(a,i) and also for when a pointer to an open array's value gets passed as an open array parameter. It also gets used to protect slices from duplicating side effects (details in the next section). (tgen)

2.6 Slices

I made the design choice that slices could only be accepted by functions that had an open array as the formal parameter - I updated check_arg to enforce this (see pattern matching with CParamDef — VParamDef). I chose to do this as I felt that due to their size being able to vary at runtime, it wouldn't fit well with Pascal's strongly typed nature. (check)

Implementing slices was similar in many senses to implementing open arrays and subscripts. I started by extending the parser's context-free grammar to include them, matching them with the pattern "variable SUB expr DOT DOT expr RPAR" which looked like "a[i..j)" in PicoPascal. Initially, this did not permit for expressions of the form a[i..j) where i is a variable (not a constant). This is because of a shift/reduce conflict, as it matches "i." with the pattern "variable DOT name", referring to a Select statement. To overcome this ambiguity, I added in the additional rule for slices "variable SUB name DOT DOT expr RPAR" which then allowed for a variable to be the first index. There was no need to add additional rules to allow for the second index to be a variable, as there was no ambiguity here. (parser)

I then added a new type of expr_guts "Slice of expr * expr * expr" to represent this new structure. The first expression referred to the array the slice was being taken from, while the second and third expressions referred to the lower and upper bound of the slice respectively. (tree)

In check_var, in similar fashion to the implementation of subscripts, I had to define slices as an expression that denotes a variable - while they aren't a single value, they denote a variable in the same way that an array does. I added slices to expr_type, annotating its abstract syntax tree while also checking that the first argument is an array (normal or open) and the other two arguments are both integers - there was no need to ensure that the integers were withing valid bounds at this point - this gets taken care of by the SLICE operator defined below. (check)

I then added slices to gen_expr so their values could be retrieved. (tgen)

I then introduced the "SLICE" operator in the intermediate language - it has the form ⟨SLICE, i, j, n⟩ where i and j are the lower and upper bounds respectively and n is the length of the array which the slice is taken from. This must return j-i if everything is valid, otherwise it should give a runtime error. I did this by first defining it as an "inst" (optree) and then adding it as a pattern matching option in eval_reg. I then wrote the assembly code that was to be generated. I took the following approach to try and minimise the registers used and the number of comparisons that was required. I evaluated i into the required register for the output, while evaluating j and n into R_any. I first check that $0 \leq i$ and then put into the required register the value of j-i. I then checked if it was greater than or equal to 0 (checking if $i \leq j$), and then checked if $j \leq n$. If any of these constraints were broken, it signaled for a runtime error via a function written in pas0.c, called "slicecheck" (pas0), very similar to the array bound error function "check". Otherwise it returned j-i in the register required. These comparisons allow the compiler to check $0 \leq i \leq j \leq n$ and for the SLICE operator to return j-i if the constraints were satisfied. (tran)

And then finally, for its code generation. **Note: slices can only get passed to functions that are expecting open arrays, and so the compiler must pass both a base address and a size, the latter provided by the new SLICE operator above**.

To avoid repeating side effects, inside gen_arg, I made use of the gen_addr_bound function that I previously defined - this gives the compiler a very easy way of retrieving the base address and size of the array, the latter value needed for the new SLICE operator. Since the lower bound's value is used in both the calculation of the new base address and the SLICE operator, I also made sure to only evaluate this expression once, into a value called lbTree, and then reuse it as necessary. The new base address is the old base address + lower bound value * size of items stored in array (last part calculated with use of "size_of" function). The size of the slice (which must also be passed in as the function must have an open array as the parameter type) is worked out via the "SLICE" operator (defined above) which makes the necessary validity checks. Having the lower bound as the constant 0 is the most common case, I treated this as a separate case in order to generate more efficient code. The only difference being that the base address of the slice is simply the base address of the array in which the slice is taken from, so no need to do any additional address arithmetic. (tgen)

3 Code

What follows is a diff listing showing the changes made to the compiler in order produce the solution described above.

Note: added lines denoted with a '+', and deleted ones with a '-'. The code below shows only the relevant changed parts, so skips areas of functions which went unchanged.

3.1 check

```
diff -r 6e99c96e0a56 lab4/check.ml
--- a/lab4/check.ml     Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/check.ml     Sat Jan 20 23:57:42 2024 +0000
@@ -90,7 +90,7 @@ let try_binop w v1 v2 =
 (* |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 | OpenArrayParamDef -> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -100,16 +100,22 @@ let rec check_var e addressible =
         let d = get_def x in
         begin
           match d.d_kind with
-              VarDef | VParamDef | CParamDef ->
+              VarDef | VParamDef | CParamDef | OpenArrayParamDef ->
                 d.d_mem <- d.d_mem || addressible
             | _ ->
                 sem_error "$ is not a variable" [fId x.x_name]
         end
     | Sub (a, i) -> check_var a addressible
+    | Slice (a,i,j) -> check_var a addressible
     | Select (r, x) -> check_var r addressible
     | Deref p -> ()
     | _ -> sem_error "a variable is needed here" []

+let get_value e =
+  match e.e_value with
+      Some v -> v
+    | None -> failwith "get_value"
+
 (* |check_expr| -- check and annotate an expression, return its type *)
 let rec check_expr e env =
   let t = expr_type e env in
@@ -138,8 +144,21 @@ and expr_type e env =
         begin
           match t1.t_guts with
               ArrayType (upb, u1) -> u1
+            | OpenArrayType t1 -> t1
             | _ -> sem_error "subscripting a non-array" []
         end
+    | Slice (e1,e2,e3) ->
+        let t1 = check_expr e1 env
+        and t2 = check_expr e2 env
+        and t3 = check_expr e3 env in
+        if not (same_type t2 integer && same_type t3 integer) then
+            sem_error "subscripts must be integers for slice" [];
+        begin
+            match t1.t_guts with
+                ArrayType(upb,u1) -> t1
+              | OpenArrayType u1 -> t1
+              | _ -> sem_error "subscripting a non-array" [];

 and check_arg formal arg env =
    match formal.d_kind with
              CParamDef | VParamDef ->
        let t1 = check_expr arg env in
        if not (same_type formal.d_type t1) then
          sem_error "argument has wrong type" [];

+        begin
+              match arg.e_guts with
+                                  Slice(a,b,c) -> sem_error "Slice can only be passed to function expecting
    open array" [];
+                                |_ ->
+                                      if formal.d_kind = VParamDef then
+                                        check_var arg true
+        end
        ...
+    | OpenArrayParamDef ->
+      let t1 = check_expr arg env in
+      if not (same_type formal.d_type t1) then
+        sem_error "bad arg!!" [];
+      check_var arg true
+    | _ -> failwith "bad formal"
...skipping...
(* |check_libcall| -- check call to a library procedure *)
and check_libcall q args env v =
  ...
  match (q.q_id, args) with
    ...
+    | (NewRowProc, [e1;arrLength]) ->
+      let t1 = check_expr e1 env in
+      if not (is_pointer t1) then
+        sem_error "first parameter of newrow must be a pointer" [];
+      begin
+        match (base_type t1).t_guts with
+          OpenArrayType t ->
+                        let t2 = check_expr arrLength env in
+                        if not (same_type t2 integer) then
+                          sem_error "second parameter of newrow must be an integer" [];
+                        check_var e1 false
+          | _ -> sem_error "pointer must point to an open array to use newrow" []
+      end
    ...
+    | (Len, [e]) ->
+      let t1 = check_expr e env in
+      if not(is_array t1 || is_openArray t1) then
+        sem_error "parameter of len is not an array" []
+    | _ -> ()
```

```
  let param_size d =
  match d.d_kind with
      CParamDef | VParamDef -> 1
-    | PParamDef -> 2
+    | PParamDef | OpenArrayParamDef -> 2
     | _ -> failwith "param_size"

 (* param_alloc -- allocate space for formal parameters *)
@@ -434,7 +483,7 @@ let global_alloc d =
 let do_alloc alloc ds =
   let h d =
     match d.d_kind with
-        VarDef | CParamDef | VParamDef | FieldDef | PParamDef ->
+        VarDef | CParamDef | VParamDef | FieldDef | PParamDef | OpenArrayParamDef ->
          alloc d
       | _ -> () in
   List.iter h ds
@@ -458,6 +507,8 @@ let rec check_typexpr te env =
  match te with
...
    | Array (upb, value) ->
        let (t1, v1) = check_const upb env
        and t2 = check_typexpr value env in
        if not (same_type t1 integer) then
          sem_error "upper bound must be an integer" [];
+       if (is_openArray t2) then
+         sem_error "array can't contain open arrays" [];
        row v1 t2
...
+
+   | OpenArray t1 ->
+     let t2 = check_typexpr t1 env in
+     if (is_openArray t2 || is_array t2) then
+       sem_error "open array can't contain arrays" [];
+     openArrayRow t2

 (* |check_decl| -- check a declaration and add it to the environment *)
 and check_decl d env =
@@ -612,7 +670,9 @@ let init_env =
       ("asr", operator Asr [integer; integer], integer);
       ("bitand", operator BitAnd [integer; integer], integer);
       ("bitor", operator BitOr [integer; integer], integer);
-      ("bitnot", operator BitNot [integer], integer)] empty
+      ("bitnot", operator BitNot [integer], integer);
+      ("len", libproc Len 1 [], integer);
+      ("newrow", libproc NewRowProc 2 [], voidtype)] empty

 (* |annotate| -- annotate the whole program *)
 let annotate (Prog (Block (globals, ss, fsize, nregv), glodefs)) =
```

3.2 dict

```
diff -r 6e99c96e0a56 lab4/dict.ml
--- a/lab4/dict.ml      Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/dict.ml      Sat Jan 20 23:57:42 2024 +0000
```

```
@@ -37,7 +37,7 @@ let fId x = fStr (spelling x)
 (* |libid| -- type of picoPascal library procedures *)
 type libid = ChrFun | OrdFun | PrintNum | PrintChar | PrintString
   | NewLine | ReadChar | ExitProc | NewProc | ArgcFun | ArgvProc
-  | OpenIn | CloseIn | Operator of Optree.op
+  | OpenIn | CloseIn | Operator of Optree.op | Len | NewRowProc

 (* |lib_name| -- name of a library procedure *)
 let lib_name x =
@@ -49,6 +49,8 @@ let lib_name x =
     | ArgcFun -> "argc" | ArgvProc -> "argv"
     | OpenIn -> "open_in" | CloseIn -> "close_in"
     | Operator op -> sprintf "$" [Optree.fOp op]
+    | Len -> "len"
+    | NewRowProc -> "newrow"

 (* |fLibId| -- format library name for printing *)
 let fLibId l = fStr (lib_name l)
@@ -86,6 +88,7 @@ type def_kind =
   | LibDef of libproc           (* Lib proc (data) *)
   | HoleDef of ptype ref        (* Pending type *)
   | DummyDef                    (* Dummy *)
+  | OpenArrayParamDef

 (* |def| -- definitions in environment *)
 and def =
@@ -109,6 +112,7 @@ and ptype =
 and type_guts =
     BasicType of basic_type
   | ArrayType of int * ptype
+  | OpenArrayType of ptype
   | RecordType of def list
   | ProcType of proc_data
   | PointerType of ptype ref
@@ -202,6 +206,9 @@ let row n t =
   let r = t.t_rep in
   mk_type (ArrayType (n, t)) { r_size = n * r.r_size; r_align = r.r_align }

+let openArrayRow t =
+  mk_type (OpenArrayType t) addr_rep
+
 let discrete t =
   match t.t_guts with
       BasicType (IntType | CharType | BoolType) -> true
@@ -218,6 +225,16 @@ let is_pointer t =
       PointerType t1 -> true
     | _ -> false

+let is_array t =
+  match t.t_guts with
+    ArrayType (n,t1) -> true
+    | _ -> false
+
+let is_openArray t =
+  match t.t_guts with
+    OpenArrayType t1 -> true
+    | _ -> false
```

```
+
 let bound t =
   match t.t_guts with
       ArrayType (n, t1) -> n
@@ -227,8 +244,14 @@ let base_type t =
   match t.t_guts with
       PointerType t1 -> !t1
     | ArrayType (n, t1) -> t1
+    | OpenArrayType t1 -> t1
     | _ -> failwith "base_type"

+let rec get_final_pointer t =
+  match t.t_guts with
+      PointerType t1 -> get_final_pointer !t1
+    | x -> x
+
 let get_proc t =
   match t.t_guts with
       ProcType p -> p
@@ -241,8 +264,16 @@ let rec same_type t1 t2 =
         && same_type p1.p_result p2.p_result
     | (ArrayType (n1, u1), ArrayType(n2, u2)) ->
         n1 = n2 && same_type u1 u2
+    | (OpenArrayType t1, OpenArrayType t2) -> same_type t1 t2
+    | (OpenArrayType t1, ArrayType(n2, t2)) -> same_type t1 t2
+    | (ArrayType(n1, t1), OpenArrayType t2) ->  same_type t1 t2
+    | (OpenArrayType _, PointerType p) -> same_type t1 !p
+    | (ArrayType(_,_), PointerType p) -> same_type t1 !p
+    | (PointerType p, OpenArrayType _) -> same_type !p t2
+    | (PointerType p, ArrayType(_,_)) -> same_type !p t2
     | (PointerType _, BasicType x) -> x = AddrType
     | (BasicType x, PointerType _) -> x = AddrType
+    | (PointerType p, PointerType q) -> same_type !p !q
     | (_, _) -> t1.t_id = t2.t_id

 and match_args fp1 fp2 =
@@ -256,6 +287,7 @@ and match_args fp1 fp2 =
 let is_string t =
   match t.t_guts with
       ArrayType (n, t1) -> same_type t1 character
+    | OpenArrayType t1 -> same_type t1 character
     | _ -> false

 let symbol_of d =
diff -r 6e99c96e0a56 lab4/dict.mli
--- a/lab4/dict.mli     Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/dict.mli     Sat Jan 20 23:57:42 2024 +0000
@@ -21,7 +21,7 @@ val fId : ident -> Print.arg
 (* |libid| -- type of picoPascal library procedures *)
 type libid = ChrFun | OrdFun | PrintNum | PrintChar | PrintString
   | NewLine | ReadChar | ExitProc | NewProc | ArgcFun | ArgvProc
-  | OpenIn | CloseIn | Operator of Optree.op
+  | OpenIn | CloseIn | Operator of Optree.op | Len | NewRowProc

 (* |fLibId| -- format libid for printing *)
 val fLibId : libid -> Print.arg
@@ -50,6 +50,7 @@ type def_kind =
   | LibDef of libproc           (* Lib proc (data) *)
   | HoleDef of ptype ref        (* Pending type *)
   | DummyDef                    (* Dummy *)
+  | OpenArrayParamDef

 (* |def| -- definitions in environment *)
 and def =
@@ -71,6 +72,7 @@ and ptype =
 and type_guts =
     BasicType of basic_type
   | ArrayType of int * ptype
+  | OpenArrayType of ptype
   | RecordType of def list
   | ProcType of proc_data
   | PointerType of ptype ref
@@ -135,6 +137,9 @@ val empty : environment
 (* |row| -- construct array type *)
 val row : int -> ptype -> ptype

+(* construct open array type *)
+val openArrayRow : ptype -> ptype
+
 (* |mk_type| -- construct new (uniquely labelled) type *)
 val mk_type : type_guts -> Mach.metrics -> ptype

@@ -147,12 +152,18 @@ val scalar : ptype -> bool
 (* |is_string| -- test if a type is 'array N of char' *)
 val is_string : ptype -> bool

+val get_final_pointer : ptype -> type_guts
+
 (* |bound| -- get bound of array type *)
 val bound : ptype -> int

 (* |is_pointer| -- test if a type is 'pointer to T' *)
 val is_pointer : ptype -> bool

+val is_array : ptype -> bool
+
+val is_openArray  :ptype -> bool
+
 (* |base_type| -- get base type of pointer or array *)
 val base_type : ptype -> ptype
```

3.3 optree

```
diff -r 6e99c96e0a56 lab4/optree.ml
--- a/lab4/optree.ml    Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/optree.ml    Sat Jan 20 23:57:42 2024 +0000
@@ -61,6 +61,7 @@ type inst =
   | BINOP of op                 (* Perform binary operation (op) *)
   | OFFSET                      (* Add address and offset *)
   | BOUND                       (* Array bound check *)
+  | SLICE
   | NCHECK                      (* Null pointer check *)
   | LABEL of codelab            (* Set code label *)
   | JUMP of codelab              (* Unconditional branch (dest) *)
@@ -93,6 +94,7 @@ let fInst =
     | BINOP w ->        fMeta "BINOP $" [fOp w]
     | OFFSET ->         fStr "OFFSET"
     | BOUND ->          fStr "BOUND"
+    | SLICE ->           fStr "SLICE"
     | NCHECK ->         fStr "NCHECK"
     | LABEL l ->        fMeta "LABEL $" [fLab l]
     | JUMP l ->         fMeta "JUMP $" [fLab l]
diff -r 6e99c96e0a56 lab4/optree.mli
--- a/lab4/optree.mli   Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/optree.mli   Sat Jan 20 23:57:42 2024 +0000
@@ -50,6 +50,7 @@ type inst =
   | BINOP of op                 (* Perform binary operation (op) *)
   | OFFSET                      (* Add address and offset *)
   | BOUND                       (* Array bound check *)
+  | SLICE
   | NCHECK                      (* Null pointer check *)
   | LABEL of codelab            (* Set code label *)
   | JUMP of codelab             (* Unconditional branch (dest) *)
```

3.4 parser

```
diff -r 6e99c96e0a56 lab4/parser.mly
--- a/lab4/parser.mly   Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/parser.mly   Sat Jan 20 23:57:42 2024 +0000
@@ -85,7 +85,9 @@ formal_decls :
   | formal_decl SEMI formal_decls       { $1 :: $3 } ;

 formal_decl :
-    ident_list COLON typexpr            { VarDecl (CParamDef, $1, $3) }
+    ident_list COLON openArray          { VarDecl (OpenArrayParamDef, $1, $3) }
+  | VAR ident_list COLON openArray      { VarDecl (OpenArrayParamDef, $2, $4) }
+  | ident_list COLON typexpr            { VarDecl (CParamDef, $1, $3) }
   | VAR ident_list COLON typexpr        { VarDecl (VParamDef, $2, $4) }
   | proc_heading                        { PParamDecl $1 } ;

@@ -181,14 +183,21 @@ expr_list :

 variable :
     name                                { make_expr (Variable $1) }
+  | variable SUB expr DOT DOT expr RPAR { make_expr (Slice ($1,$3,$6)) }
+  | variable SUB name DOT DOT expr RPAR { make_expr (Slice ($1,make_expr (Variable $3),$6)) }
   | variable SUB expr BUS               { make_expr (Sub ($1, $3)) }
   | variable DOT name                   { make_expr (Select ($1, $3)) }
   | variable ARROW                      { make_expr (Deref $1) } ;

+
+openArray :
+  ARRAY OF typexpr                    { OpenArray $3 }
+
 typexpr :
     name                                { TypeName $1 }
   | ARRAY expr OF typexpr               { Array ($2, $4) }
   | RECORD fields END                   { Record $2 }
+  | POINTER TO openArray                { Pointer $3 }
```

```
  | POINTER TO typexpr                  { Pointer $3 } ;

 fields :
```

3.5 pas0

```
diff -r 6e99c96e0a56 lab4/pas0.c
--- a/lab4/pas0.c       Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/pas0.c       Sat Jan 20 23:57:42 2024 +0000
@@ -70,6 +70,11 @@ void check(int n) {
      exit(2);
 }

+void slicecheck(int n) {
+     fprintf(stderr, "Slice error on line %d\n", n);
+     exit(2);
+}
+
 void nullcheck(int n) {
      fprintf(stderr, "Null pointer check on line %d\n", n);
      exit(2);
```

3.6 tgen

```
diff -r 6e99c96e0a56 lab4/tgen.ml
--- a/lab4/tgen.ml      Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/tgen.ml      Sat Jan 20 23:57:42 2024 +0000
@@ -12,6 +12,7 @@ let boundchk = ref false
 let optlevel = ref 0
 let debug = ref 0

+
 (* |level| -- nesting level of current procedure *)
 let level = ref 0

@@ -46,13 +47,33 @@ let address d =
     | Nowhere ->
         failwith (sprintf "address $" [fId d.d_tag])

+(* 3 cases - normal array (size in def), openarray parameter (size at +4) or pointer to open array (size at -4)
    *)
+let size_of_array e1 =
+    let deepestExpr = deref_chain e1 in
+    match e1.e_type.t_guts with
+      (* normal array, size in definition *)
+        ArrayType(n,t) -> <CONST n>
+       | OpenArrayType t ->
+                            (
+                                  match deepestExpr.e_guts with
+                                      Variable x ->(match (get_def x).d_kind with
+                                                            (* open array parameter - offset is +4 *)
+                                                            OpenArrayParamDef -> <LOADW, <OFFSET, address (
    get_def x), <CONST 4>>>
+                                                          (* must be pointer to open array by process of
    elimination - offset is -4 *)
+                                                           | _ -> <LOADW, <OFFSET, <LOADW, address (get_def x)>,
     <CONST (-4)>>>)
+                                   | _ -> failwith "can't take length of non-variable"
+                             )
+       | _ -> failwith "not a valid array"
+
+
 (* |gen_closure| -- two trees for a (code, envt) pair *)
 let gen_closure d =
   match d.d_kind with
       ProcDef ->
         (<GLOBAL (symbol_of d)>,
           if d.d_level = 0 then <CONST 0> else schain (!level - d.d_level))
-     | PParamDef ->
+     | PParamDef | OpenArrayParamDef ->
         (<LOADW, address d>,
           <LOADW, <OFFSET, address d, <CONST addr_rep.r_size>>>)
     | _ -> failwith "missing closure"
@@ -71,8 +92,44 @@ let libcall sym args rtype =
 let gen_copy dst src n =
   libcall "memcpy" [dst; src; <CONST n>] voidtype

+let rec gen_addr_bound a =
+    let temp = Regs.new_temp() in
+    match get_final_pointer a.e_type with
+        ArrayType(n,t) ->
+                            let deepestExpr = deref_chain a in
+                            begin
+                                  match deepestExpr.e_guts with
+                                      Variable x ->
+                                                    let baseAddrTree = gen_addr deepestExpr in
+                                                    let boundTree = <CONST n> in
+                                                    (baseAddrTree,boundTree)
+                                        | _ -> failwith "non-variable is not suitable for bounds checking!"
+                            end
+
+       | OpenArrayType t ->
+                            let deepestExpr = deref_chain a in (*need to determine if it is an open array
    parameter or if it is the dereference value of some pointer (or chain of pointers)*)
+                            begin
+                                match deepestExpr.e_guts with
+                                    Variable x -> let d = get_def x in
+                                                  begin
+                                                        match d.d_kind with
+                                                            OpenArrayParamDef ->
+                                                                                  let addr = address d in
+                                                                                  let baseAddrTree = <AFTER, <
    DEFTEMP temp, addr>, <LOADW, <TEMP temp>>> in
+                                                                                  let boundTree = <LOADW, <OFFSET,
    <TEMP temp>, <CONST 4>>>
+                                                                                  in (baseAddrTree,boundTree)
+                                                          |_ -> (* pointer to open array *)
+                                                                                  let addr = address d in
+                                                                                  let baseAddrTree = <AFTER, <
    DEFTEMP temp, <LOADW, addr>>, <TEMP temp>> in
+                                                                                  let boundTree = <LOADW, <OFFSET,
    <TEMP temp>, <CONST (-4)>>>
```

```
+                                                                                  in (baseAddrTree,boundTree)
+                                                     end
+                                 | _ -> failwith "non-variable is not suitable for bounds checking"
+                           end
+
+
 (* |gen_addr| -- code for the address of a variable *)
-let rec gen_addr v =
+and gen_addr v =
   match v.e_guts with
       Variable x ->
         let d = get_def x in
@@ -89,24 +146,35 @@ let rec gen_addr v =
                     <LOADW, address d>
             | StringDef ->
                 address d
+            | OpenArrayParamDef ->
+              <LOADW, address d>
            | _ ->
                 failwith "load_addr"
         end
     | Sub (a, i) ->
-        let bound_check t =
-          if not !boundchk then t else <BOUND, t, <CONST (bound a.e_type)>> in
-        <OFFSET,
-          gen_addr a,
-          <BINOP Times, bound_check (gen_expr i), <CONST (size_of v.e_type)>>>
+        begin
+            match a.e_type.t_guts with
+                ArrayType(n,t) ->     if not !boundchk then
+                                          <OFFSET, gen_addr a, <BINOP Times, gen_expr i, <CONST (size_of v.
    e_type)>>>
+                                      else
+                                          let (baseAddrTree,boundTree) = gen_addr_bound a in
+                                          <OFFSET, baseAddrTree, <BINOP Times, <BOUND, gen_expr i, boundTree>,
    <CONST (size_of v.e_type)>>>
+              | OpenArrayType t ->
+                                      if not !boundchk then
+                                          <OFFSET, gen_addr a, <BINOP Times, gen_expr i, <CONST (size_of v.
    e_type)>>>
+                                      else
+                                          let (baseAddrTree,boundTree) = gen_addr_bound a in
+                                           <OFFSET, baseAddrTree, <BINOP Times, <BOUND, gen_expr i, boundTree>,
     <CONST (size_of v.e_type)>>>
+        end
    | Select (r, x) ->
        let d = get_def x in
        <OFFSET, gen_addr r, <CONST (offset_of d)>>
    | Deref p ->
        let null_check t =
          if not !boundchk then t else <NCHECK, t> in
-        null_check (gen_expr p)
+        null_check (gen_expr (deref_chain p))
    | String (lab, n) -> <GLOBAL lab>
-    | _ -> failwith "gen_addr"
+    | _ -> failwith "gen_addr!"
```

```
 (* |gen_expr| -- tree for the value of an expression *)
 and gen_expr e =
@@ -116,7 +184,7 @@ and gen_expr e =
     | None ->
         begin
           match e.e_guts with
-              Variable _ | Sub _ | Select _ | Deref _ ->
+              Variable _ | Sub _ | Select _ | Deref _ | Slice _->
                 let ld = if size_of e.e_type = 1 then LOADC else LOADW in
                 <ld, gen_addr e>
             | Monop (w, e1) ->

@@ -162,7 +230,52 @@ and gen_arg f a =
   match f.d_kind with
...
+    | OpenArrayParamDef ->
+      begin
+        match a.e_guts with
+          Variable x ->
+            let def = get_def x in
+            let kind = def.d_kind in
+            begin
+
+              match kind with
+                OpenArrayParamDef -> let (addr, length) = gen_closure def in [addr;length]
+
+                | _ ->
+
+                    begin
+                          match a.e_type.t_guts with (*if it is not an open array parameter, can only be a
    normal array or a pointer to an (open) array*)
+                              ArrayType(n,t) -> [gen_addr a; <CONST n>]
+                            | PointerType p ->
+                                                  let (baseAddrTree,boundTree) = gen_addr_bound (deref_chain a)
    in
+                                                  [baseAddrTree; boundTree]
+                            | _ -> failwith "illegal argument"
+                    end
+            end
+          | Deref e -> gen_arg f (deref_chain e)
+          | Slice(ar,lb,ub) ->
+              begin
+                    match ar.e_type.t_guts with
+                        ArrayType _ | OpenArrayType _ ->
+                                              let lbTree = gen_expr lb in
+                                              let ubTree = gen_expr ub in
+                                              let (baseAddrTree,boundTree) = gen_addr_bound ar in (* to avoid
    repeating side effects *)
+
+                                              if (lb.e_value != None && get_value lb == 0) then (* trivial case
     *)
+                                                  let arrLength = <SLICE, lbTree, ubTree, boundTree> in
+                                                  [baseAddrTree; arrLength]
+                                              else
+                                                  let baseAddr = <OFFSET, baseAddrTree, <BINOP Times, lbTree, <
    CONST (size_of(base_type ar.e_type))>>> in
+                                                  let arrLength = <SLICE, lbTree, ubTree, boundTree> in
+                                                     [baseAddr; arrLength]
+                    | _ -> failwith "not an array"
+              end
+
+
+
+          | _ -> failwith "bad"
+      end
+    | _ -> failwith "bad arg!"

 (* |gen_libcall| -- generate code to call a built-in procedure *)
 and gen_libcall q args =
@@ -170,13 +283,22 @@ and gen_libcall q args =
      (ChrFun, [e]) -> gen_expr e
    | (OrdFun, [e]) -> gen_expr e
    | (PrintString, [e]) ->
-       libcall "print_string"
-         [gen_addr e; <CONST (bound e.e_type)>] voidtype
+       begin
+           match e.e_type.t_guts with
+               ArrayType(n,t) ->  libcall "print_string" [gen_addr e; <CONST (bound e.e_type)>] voidtype
+
+             | OpenArrayType t -> libcall "print_string" [gen_addr e; size_of_array e] voidtype
+       end
+
    | (ReadChar, [e]) ->
        libcall "read_char" [gen_addr e] voidtype
    | (NewProc, [e]) ->
        let size = size_of (base_type e.e_type) in
        <STOREW, libcall "palloc" [<CONST size>] addrtype, gen_addr e>
+   | (NewRowProc, [e1;arrLength]) ->
+       let e1 = deref_chain e1 in
+       let elemSize = size_of (base_type e1.e_type) in
+       <STOREW, libcall "palloc2" [<CONST (get_value arrLength)>; <CONST elemSize>] addrtype, gen_addr e1>
    | (ArgcFun, []) ->
        libcall "argc" [] integer
    | (ArgvProc, [e1; e2]) ->
@@ -189,6 +311,42 @@ and gen_libcall q args =
        <MONOP op, gen_expr e1>
    | (Operator op, [e1; e2]) ->
        <BINOP op, gen_expr e1, gen_expr e2>
+   | (Len, [e]) ->     (*different cases for normal array, open array parameter and pointer to (open) array*)
+       size_of_array e
+
    | (_, _) ->
        let proc = sprintf "$" [fLibId q.q_id] in
        libcall proc (List.map gen_expr args) voidtype
```

3.7 tran

```
diff -r 6e99c96e0a56 lab4/tran.ml
--- a/lab4/tran.ml      Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/tran.ml      Sat Jan 20 23:57:42 2024 +0000
@@ -152,6 +152,13 @@ let rec eval_reg t r =
        gen_reg "cmp $1, $2 / ldrhs r0, =$3 / blhs check"
          (reg_of v1) [v1; v2; number !line]
```

```
+    | <SLICE, i, j, n> ->
+        let v1 = eval_reg i r in
+        let v2 = eval_reg j R_any in
+        let v3 = eval_reg n R_any in
+        gen_reg "cmp $1, #0 / ldrmi r0, =$4 / blmi slicecheck / sub $1, $2, $1 / cmp $1, #0 / ldrmi r0, =$4 /
    blmi slicecheck / cmp $3, $2 / ldrmi r0, =$4 / blmi slicecheck"
+          (reg_of v1) [v1; v2; v3; number !line]
+
     | <NCHECK, t1> ->
         let v1 = eval_reg t1 r in
         gen_reg "cmp $1, #0 / ldreq r0, =$2 / bleq nullcheck"
```

3.8 tree

```
diff -r 6e99c96e0a56 lab4/tree.ml
--- a/lab4/tree.ml      Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/tree.ml      Sat Jan 20 23:57:42 2024 +0000
@@ -49,6 +49,7 @@ and expr =
 and expr_guts =
     Constant of int * ptype
   | Variable of name
+  | Slice of expr * expr * expr
   | Sub of expr * expr
   | Select of expr * name
   | Deref of expr
@@ -61,9 +62,15 @@ and expr_guts =
 and typexpr =
     TypeName of name
   | Array of expr * typexpr
+  | OpenArray of typexpr
   | Record of decl list
   | Pointer of typexpr

+let rec deref_chain e =
+    match e.e_guts with
+      Deref e2 -> deref_chain e2
+    | _ -> e
+
 (* |make_expr| -- construct an expression node with dummy annotations *)
 let make_expr e =
   { e_guts = e; e_type = voidtype; e_value = None }
@@ -156,6 +163,7 @@ and fExpr e =
   match e.e_guts with
       Constant (n, t) -> fMeta "(CONST $)" [fNum n]
     | Variable x -> fName x
+    | Slice (e1,e2,e3) -> fMeta "(SLICE $ $ $)" [fExpr e1; fExpr e2; fExpr e3]
     | Sub (e1, e2) -> fMeta "(SUB $ $)" [fExpr e1; fExpr e2]
     | Select (e1, x) -> fMeta "(SELECT $ $)" [fExpr e1; fName x]
     | Deref e1 -> fMeta "(DEREF $)" [fExpr e1]
@@ -172,6 +180,7 @@ and fType =
   function
      TypeName x -> fName x
     | Array (e, t1) -> fMeta "(ARRAY $ $)" [fExpr e; fType t1]
+    | OpenArray t1 -> fMeta "(OPENARRAY $)" [fType t1]
     | Record fields -> fMeta "(RECORD$)" [fTail(fDecl) fields]
     | Pointer t1 -> fMeta "(POINTER $)" [fType t1]

diff -r 6e99c96e0a56 lab4/tree.mli
--- a/lab4/tree.mli     Thu Nov 02 18:58:50 2023 +0000
+++ b/lab4/tree.mli     Sat Jan 20 23:57:42 2024 +0000
@@ -64,6 +64,7 @@ and expr =
 and expr_guts =
     Constant of int * ptype
   | Variable of name
+  | Slice of expr * expr * expr
   | Sub of expr * expr
   | Select of expr * name
   | Deref of expr
@@ -76,9 +77,12 @@ and expr_guts =
 and typexpr =
     TypeName of name
   | Array of expr * typexpr
+  | OpenArray of typexpr
   | Record of decl list
   | Pointer of typexpr

+val deref_chain : expr -> expr
```

4 Testing

Note: bounds checking disabled unless specified in order to save space

4.1 Open Arrays

The following test is to show that open array parameters work for arrays of arbitrary size, including use of the new len function.

Code:

```
    (* demonstrates that functions with open arrays work for arrays of arbitrary size
including len function*)

var b : array 5 of integer;
var c : array 6 of integer;

proc sum(a :array of integer) : integer;
    var i,s : integer;
begin
    s := 0;
    for i := 0 to len(a) - 1 do
        s := s + a[i]
    end;
    return s
end;

begin
    b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;
    c[0] := 1; c[1] := 2; c[2] := 3; c[3] := 4; c[4] := 5; c[5] := 6;
    print_num(sum(b));
    newline();
    print_num(sum(c));
    newline();
end.
```

Output:

```
15
21
```

Assembly Code (array value assignments omitted to save space):

```
    @ picoPascal compiler output
        .global pmain

@ proc sum(a :array of integer) : integer;
        .text
_sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
```

```
@ for i := 0 to len(a) - 1 do
        mov r4, #0
        add r0, fp, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L2:
        cmp r4, r6
        bgt .L3
@ s := s + a[i]
        ldr r0, [fp, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
        add r4, r4, #1
        b .L2
.L3:
@ return s
        mov r0, r5
        b .L1
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;

@ c[0] := 1; c[1] := 2; c[2] := 3; c[3] := 4; c[4] := 5; c[5] := 6;

@ print_num(sum(b));
        mov r1, #5
        ldr r0, =_b
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ print_num(sum(c));
        mov r1, #6
        ldr r0, =_c
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ end.
.L4:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _b, 20, 4
        .comm _c, 24, 4
        .section .note.GNU-stack
```

```
@ End
```

4.2 Nesting 1

This shows that open array parameters can also be supplied to functions themselves.

Code:

```
    (* showing open array parameters can be passed into functions as well *)

var b : array 5 of integer;

proc f(a: array of integer): integer;
    proc sum(a :array of integer) : integer;
        var i,s : integer;
    begin
        s := 0;
        for i := 0 to len(a) - 1 do
            s := s + a[i]
        end;
        return s
    end;
begin
    return sum(a)
end;

begin
    b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;
    print_num(f(b));
    newline();
end.
```

Output

```
    15
```

Assembly Code (array value assignments omitted to save space):

```
    @ picoPascal compiler output
        .global pmain

@ proc f(a: array of integer): integer;
        .text
_f:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ return sum(a)
        add r0, fp, #40
        ldr r1, [r0, #4]
        ldr r0, [fp, #40]
        mov r10, fp
        bl _f.sum
        b .L1
```

```
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@     proc sum(a :array of integer) : integer;
_f.sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
@ for i := 0 to len(a) - 1 do
        mov r4, #0
        add r0, fp, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L3:
        cmp r4, r6
        bgt .L4
@ s := s + a[i]
        ldr r0, [fp, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
        add r4, r4, #1
        b .L3
.L4:
@ return s
        mov r0, r5
        b .L2
.L2:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;

@ print_num(f(b));
        mov r1, #5
        ldr r0, =_b
        mov r10, #0
        bl _f
        bl print_num
@ newline();
        bl newline
@ end.
.L5:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _b, 20, 4
        .section .note.GNU-stack
@ End
```

4.3 Nesting 2

This shows that a nested procedure can make access to an open array parameter of an enclosing procedure.

Code:

```
    (* showing nested procedure making access to open array parameter of enclosing procedure *)

var b : array 5 of integer;

proc f(a: array of integer): integer;
    proc sum() : integer;
        var i,s : integer;
    begin
        s := 0;
        for i := 0 to len(a) - 1 do
            s := s + a[i]
        end;
        return s
    end;
begin
    return sum()
end;

begin
    b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;
    print_num(f(b));
    newline();
end.
```

Output:

```
    15
```

Assembly Code (array assignment omitted):

```
    @ picoPascal compiler output
        .global pmain

@ proc f(a: array of integer): integer;
        .text
_f:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ return sum()
        mov r10, fp
        bl _f.sum
        b .L1
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@     proc sum() : integer;
_f.sum:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
@ for i := 0 to len(a) - 1 do
        mov r4, #0
        ldr r0, [fp, #24]
        add r0, r0, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L3:
        cmp r4, r6
        bgt .L4
@ s := s + a[i]
        ldr r0, [fp, #24]
        ldr r0, [r0, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
        add r4, r4, #1
        b .L3
.L4:
@ return s
        mov r0, r5
        b .L2
.L2:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;

@ print_num(f(b));
        mov r1, #5
        ldr r0, =_b
        mov r10, #0
        bl _f
        bl print_num
@ newline();
        bl newline
@ end.
.L5:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _b, 20, 4
        .section .note.GNU-stack
```

```
@ End
```

4.4 Pointers

This shows that pointers and chains of pointers behave well, as well as the len function for the dereferenced value of a pointer, and that dereferencing a pointer to an open array can be passed to a function.

Code:

```
    (* demonstrates pointers and arbitrary chains of pointers
 that len works for pointers and that pointers to open arrays can be
 dereferenced and successfully passed to functions *)

var p: pointer to array of integer;
var q: pointer to pointer to array of integer;
var x: pointer to pointer to pointer to array of integer;

proc sum(a :array of integer) : integer;
    var i,s : integer;
begin
    s := 0;
    for i := 0 to len(a) - 1 do
        s := s + a[i]
    end;
    return s
end;

begin
    newrow(p,3);
    new(q);
    newrow(q^,4);
    new(x);
    new(x^);
    newrow(x^^,5);
    print_num(len(p^));
    newline();
    print_num(len(q^^));
    newline();
    print_num(len(x^^^));
    newline();

    x^^^[0] := 1; x^^^[1] := 2; x^^^[2] := 3; x^^^[3] := 4; x^^^[4] := 5;
    print_num(sum(x^^^));
    newline();
end.
```

Output:

```
3
4
5
15
```

Assembly Code:

```
@ picoPascal compiler output
        .global pmain

@ proc sum(a :array of integer) : integer;
        .text
_sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
@ for i := 0 to len(a) - 1 do
        mov r4, #0
        add r0, fp, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L2:
        cmp r4, r6
        bgt .L3
@ s := s + a[i]
        ldr r0, [fp, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
        add r4, r4, #1
        b .L2
.L3:
@ return s
        mov r0, r5
        b .L1
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ newrow(p,3);
        mov r1, #4
        mov r0, #3
        bl palloc2
        ldr r1, =_p
        str r0, [r1]
@ new(q);
        mov r0, #4
        bl palloc
        ldr r1, =_q
        str r0, [r1]
@ newrow(q^,4);
        mov r1, #4
        mov r0, #4
        bl palloc2
        ldr r1, =_q
        str r0, [r1]
@ new(x);
        mov r0, #4
        bl palloc
        ldr r1, =_x
        str r0, [r1]
@ new(x^);
        mov r0, #4
        bl palloc
        ldr r1, =_x
        ldr r1, [r1]
        str r0, [r1]
@ newrow(x^^,5);
        mov r1, #4
        mov r0, #5
        bl palloc2
        ldr r1, =_x
        str r0, [r1]
@ print_num(len(p^));
        ldr r0, =_p
        ldr r0, [r0]
        ldr r0, [r0, #-4]
        bl print_num
@ newline();
        bl newline
@ print_num(len(q^^));
        ldr r0, =_q
        ldr r0, [r0]
        ldr r0, [r0, #-4]
        bl print_num
@ newline();
        bl newline
@ print_num(len(x^^^));
        ldr r0, =_x
        ldr r0, [r0]
        ldr r0, [r0, #-4]
        bl print_num
@ newline();
        bl newline
@ x^^^[0] := 1; x^^^[1] := 2; x^^^[2] := 3; x^^^[3] := 4; x^^^[4] := 5;
        mov r0, #1
        ldr r1, =_x
        ldr r1, [r1]
        mov r2, #0
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        mov r0, #2
        ldr r1, =_x
        ldr r1, [r1]
        mov r2, #1
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        mov r0, #3
        ldr r1, =_x
        ldr r1, [r1]
        mov r2, #2
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        mov r0, #4
        ldr r1, =_x
        ldr r1, [r1]
        mov r2, #3
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
        mov r0, #5
        ldr r1, =_x
        ldr r1, [r1]
        mov r2, #4
        mov r3, #4
        mul r2, r2, r3
        add r1, r1, r2
        str r0, [r1]
@ print_num(sum(x^^^));
        ldr r0, =_x
        ldr r4, [r0]
        ldr r1, [r4, #-4]
        mov r0, r4
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ end.
.L4:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _p, 4, 4
        .comm _q, 4, 4
        .comm _x, 4, 4
        .section .note.GNU-stack
@ End
```

4.5 Slices

This shows the workings of slices and the validation done by the SLICE operator.

Code:

```
    var a : array 5 of integer;
var b : pointer to array of integer;
var i : integer ;
var j : integer;
```

```
proc sum(b : array of integer) : integer;
    var i, s: integer;
begin
    s := 0;
    for i := 0 to len(b)-1 do
        s := s + b[i];
    end;
    return s
end;

begin
    i := 2;
    j := 4;
    a[0]:=1;a[1]:=2;a[i]:=3;a[3]:=4;a[4]:=5;

    print_num(sum(a[2..5]));
    newline();

    print_num(sum(a[i..j]));
    newline();

    newrow(b,3);
    b^[0] := 1; b^[1] := 2; b^[2] := 3;
    print_num(sum(b^[i-1..j-1]));
    newline();

    (* common case - efficient code *)
    print_num(sum(a[0..4]));
    newline();

    (* this is essentially an empty array *)
    print_num(sum(a[0..0]));
    newline();


    (* checking validity of slices *)
    print_num(sum(a[0..6]));
    newline();


    print_num(sum(a[3..1]));
    newline();
end.
```

Output:

```
12
7
5
10
0
Slice error on line 42
```

(Note that while a runtime error is produced for "print_num(sum(a[0..6]));" the same error is produced for "print_num(sum(a[3..1]));" when it is commented out - this shows that the validity checks work well.)

Assembly Code (array value assignments omitted) - Please note the particularly efficient code for the case when the initial index is 0:

```
@ picoPascal compiler output
        .global pmain

@ proc sum(b : array of integer) : integer;
        .text
_sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
@ for i := 0 to len(b)-1 do
        mov r4, #0
        add r0, fp, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L2:
        cmp r4, r6
        bgt .L3
@ s := s + b[i];
        ldr r0, [fp, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
@ end;
        add r4, r4, #1
        b .L2
.L3:
@ return s
        mov r0, r5
        b .L1
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ i := 2;
        mov r0, #2
        ldr r1, =_i
        str r0, [r1]
@ j := 4;
        mov r0, #4
        ldr r1, =_j
        str r0, [r1]
@ a[0]:=1;a[1]:=2;a[i]:=3;a[3]:=4;a[4]:=5;

@ print_num(sum(a[2..5)));
        ldr r4, =_a
        mov r1, #2
        mov r0, #5
        mov r2, #5
        cmp r1, #0
        ldrmi r0, =21
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =21
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =21
        blmi slicecheck
        mov r0, #2
        mov r2, #4
        mul r0, r0, r2
        add r0, r4, r0
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ print_num(sum(a[i..j]));
        ldr r4, =_a
        ldr r0, =_i
        ldr r1, [r0]
        ldr r0, =_j
        ldr r0, [r0]
        mov r2, #5
        cmp r1, #0
        ldrmi r0, =24
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =24
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =24
        blmi slicecheck
        ldr r0, =_i
        ldr r0, [r0]
        mov r2, #4
        mul r0, r0, r2
        add r0, r4, r0
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ newrow(b,3);
        mov r1, #4
        mov r0, #3
        bl palloc2
        ldr r1, =_b
        str r0, [r1]
@ b^[0] := 1; b^[1] := 2; b^[2] := 3;

@ print_num(sum(b^[i-1..j-1]));
```

```
        ldr r0, =_b
        ldr r4, [r0]
        ldr r0, =_i
        ldr r0, [r0]
        sub r1, r0, #1
        ldr r0, =_j
        ldr r0, [r0]
        sub r0, r0, #1
        ldr r2, [r4, #-4]
        cmp r1, #0
        ldrmi r0, =29
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =29
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =29
        blmi slicecheck
        ldr r0, =_i
        ldr r0, [r0]
        sub r0, r0, #1
        mov r2, #4
        mul r0, r0, r2
        add r0, r4, r0
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ print_num(sum(a[0..4]));
        ldr r4, =_a
        mov r1, #0
        mov r0, #4
        mov r2, #5
        cmp r1, #0
        ldrmi r0, =33
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =33
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =33
        blmi slicecheck
        mov r0, r4
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ print_num(sum(a[0..0]));
        ldr r4, =_a
        mov r1, #0
        mov r0, #0
        mov r2, #5
        cmp r1, #0
        ldrmi r0, =37
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =37
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =37
        blmi slicecheck
        mov r0, r4
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ print_num(sum(a[0..6]));
        ldr r4, =_a
        mov r1, #0
        mov r0, #6
        mov r2, #5
        cmp r1, #0
        ldrmi r0, =42
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =42
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =42
        blmi slicecheck
        mov r0, r4
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ print_num(sum(a[3..1]));
        ldr r4, =_a
        mov r1, #3
        mov r0, #1
        mov r2, #5
        cmp r1, #0
        ldrmi r0, =46
        blmi slicecheck
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =46
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =46
        blmi slicecheck
        mov r0, #3
        mov r2, #4
        mul r0, r0, r2
        add r0, r4, r0
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ end.
.L4:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _a, 20, 4
        .comm _b, 4, 4
        .comm _i, 4, 4
        .comm _j, 4, 4
        .section .note.GNU-stack
@ End
```

4.6 Nesting 3

This shows that the compiler can take slices of open array parameters.

Code:

```
(* taking an open array parameter and passing on a slice to another parameter *)

var b : array 5 of integer;

proc f(a: array of integer): integer;
    proc sum(a : array of integer) : integer;
        var i,s : integer;
    begin
        s := 0;
        for i := 0 to len(a) - 1 do
            s := s + a[i]
        end;
        return s
    end;
begin
    return sum(a[2..4))
end;

begin
    b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;
    print_num(f(b));
    newline();
end.
```

Output:

```
7
```

Assembly Code (array assignments omitted):

```
@ picoPascal compiler output
        .global pmain

@ proc f(a: array of integer): integer;
        .text
```

```
_f:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ return sum(a[2..4))
        add r4, fp, #40
        mov r1, #2
        mov r0, #4
        ldr r2, [r4, #4]
        sub r1, r0, r1
        cmp r1, #0
        ldrmi r0, =16
        blmi slicecheck
        cmp r2, r0
        ldrmi r0, =16
        blmi slicecheck
        ldr r0, [r4]
        mov r2, #2
        mov r3, #4
        mul r2, r2, r3
        add r0, r0, r2
        mov r10, fp
        bl _f.sum
        b .L1
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@     proc sum(a : array of integer) : integer;
_f.sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
@ for i := 0 to len(a) - 1 do
        mov r4, #0
        add r0, fp, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L3:
        cmp r4, r6
        bgt .L4
@ s := s + a[i]
        ldr r0, [fp, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
        add r4, r4, #1
        b .L3
.L4:
@ return s
        mov r0, r5
        b .L2
.L2:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ b[0] := 1; b[1] := 2; b[2] := 3; b[3] := 4; b[4] := 5;

@ print_num(f(b));
        mov r1, #5
        ldr r0, =_b
        mov r10, #0
        bl _f
        bl print_num
@ newline();
        bl newline
@ end.
.L5:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _b, 20, 4
        .section .note.GNU-stack
@ End
```

4.7 Bounds Checking

This demonstrates that bounds checking works for normal arrays, open array parameters and pointers to open arrays.

Code:

```
(* demonstrates that bounds checking works for normal arrays, open arrays
and open arrays accessed via deferencing a pointer*)

var a : array 3 of integer;
var b : pointer to array of integer;

proc f(b: array of integer) : integer;
begin
    return b[2];
end;

begin
    a[0] := 1; a[1] := 2; a[2] := 3;
    print_num(f(a)); newline();
    print_num(a[3]); newline();
    newrow(b,3);
    b^[0] := 1; b^[1] := 2; b^[2] := 33;
    print_num(f(b^)); newline();
    newrow(b,2);
    print_num(f(b^)); newline();
    print_num(b^[3]);
    newline();
end.
```

Output (bounds checking enabled):

Code as given:

```
3
Array bound error on line 15
```

Commenting out “print_num(a[3]); newline();”:

```
3
33
Array bound error on line 9
```

Commenting out “print_num(f(b̂)); newline();*” as well:

```
3
33
Array bound error on line 21
```

Assembly Code (for code provided):

```
@ picoPascal compiler output
        .global pmain

@ proc f(b: array of integer) : integer;
        .text
_f:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ return b[2];
        add r4, fp, #40
        ldr r0, [r4]
        mov r1, #2
        ldr r2, [r4, #4]
        cmp r1, r2
        ldrhs r0, =9
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r0, r0, r1
        ldr r0, [r0]
        b .L1
@ end;
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ a[0] := 1; a[1] := 2; a[2] := 3;
```

```
        ldr r4, =_a
        mov r0, #1
        mov r1, #0
        cmp r1, #3
        ldrhs r0, =13
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
        ldr r4, =_a
        mov r0, #2
        mov r1, #1
        cmp r1, #3
        ldrhs r0, =13
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
        ldr r4, =_a
        mov r0, #3
        mov r1, #2
        cmp r1, #3
        ldrhs r0, =13
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
@ print_num(f(a)); newline();
        mov r1, #3
        ldr r0, =_a
        mov r10, #0
        bl _f
        bl print_num
        bl newline
@ print_num(a[3]); newline();
        ldr r4, =_a
        mov r0, #3
        cmp r0, #3
        ldrhs r0, =15
        blhs check
        mov r1, #4
        mul r0, r0, r1
        add r0, r4, r0
        ldr r0, [r0]
        bl print_num
        bl newline
@ newrow(b,3);
        mov r1, #4
        mov r0, #3
        bl palloc2
        ldr r1, =_b
        str r0, [r1]
@ b^[0] := 1; b^[1] := 2; b^[2] := 33;
        ldr r0, =_b
        ldr r4, [r0]
        mov r0, #1
        mov r1, #0
        ldr r2, [r4, #-4]
        cmp r1, r2
        ldrhs r0, =17
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
        ldr r0, =_b
        ldr r4, [r0]
        mov r0, #2
        mov r1, #1
        ldr r2, [r4, #-4]
        cmp r1, r2
        ldrhs r0, =17
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
        ldr r0, =_b
        ldr r4, [r0]
        mov r0, #33
        mov r1, #2
        ldr r2, [r4, #-4]
        cmp r1, r2
        ldrhs r0, =17
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
@ print_num(f(b^)); newline();
        ldr r0, =_b
        ldr r4, [r0]
        ldr r1, [r4, #-4]
        mov r0, r4
        mov r10, #0
        bl _f
        bl print_num
        bl newline
@ newrow(b,2);
        mov r1, #4
        mov r0, #2
        bl palloc2
        ldr r1, =_b
        str r0, [r1]
@ print_num(f(b^)); newline();
        ldr r0, =_b
        ldr r4, [r0]
        ldr r1, [r4, #-4]
        mov r0, r4
        mov r10, #0
        bl _f
        bl print_num
        bl newline
@ print_num(b^[3]);
        ldr r0, =_b
        ldr r4, [r0]
        mov r0, #3
        ldr r1, [r4, #-4]
        cmp r0, r1
        ldrhs r0, =21
        blhs check
        mov r1, #4
        mul r0, r0, r1
        add r0, r4, r0
        ldr r0, [r0]
        bl print_num
@ newline();
        bl newline
@ end.
.L2:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _a, 12, 4
        .comm _b, 4, 4
        .section .note.GNU-stack
@ End
```

4.8 Avoiding Repeated Evaluation 1

This is to show that dereferencing a pointer to an open array and passing it in as an open array parameter doesn't lead to repeated evaluation of the base address by making use of a temporary register.

Code:

```
(* show that passing in a pointer to an open array as an open array parameter doesn't lead to repeated
evaluation - so not to duplicate side effects *)

var p : pointer to array of integer;

proc sum(b : array of integer) : integer;
    var i, s: integer;
begin
    s := 0;
    for i := 0 to len(b)-1 do
        s := s + b[i];
    end;
    return s
end;

begin
    newrow(p,5);
    p^[0] := 1; p^[1] := 2; p^[2] := 3; p^[3] := 4; p^[4] := 5;
    print_num(sum(p^));
    newline();
end.
```

Output:

```
   15
```

The abstract syntax tree below shows the base address getting evaluated into a temporary register and then used for the address and again to find the size of the array. The AFTER node is not present due to the use of Optree.canonicalise:

```
@     print_num(sum(p^));
@ <DEFTEMP 6, <LOADW, <GLOBAL _p>>>
@ <CALL
@   1,
@   <GLOBAL print_num>,
@   <ARG
@     0,
@     <CALL
@       2,
@       <GLOBAL _sum>,
@       <STATLINK, <CONST 0>>,
@       <ARG 0, <TEMP 6>>,
@       <ARG 1, <LOADW, <OFFSET, <TEMP 6>, <CONST -4>>>>>>>
```

Assembly (array assignments omitted):

```
@ picoPascal compiler output
        .global pmain

@ proc sum(b : array of integer) : integer;
        .text
_sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ s := 0;
        mov r5, #0
@ for i := 0 to len(b)-1 do
        mov r4, #0
        add r0, fp, #40
        ldr r0, [r0, #4]
        sub r6, r0, #1
.L2:
        cmp r4, r6
        bgt .L3
@ s := s + b[i];
        ldr r0, [fp, #40]
        mov r1, #4
        mul r1, r4, r1
        add r0, r0, r1
        ldr r0, [r0]
        add r5, r5, r0
@ end;
        add r4, r4, #1
        b .L2
.L3:
@ return s
```

```
        mov r0, r5
        b .L1
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ newrow(p,5);
        mov r1, #4
        mov r0, #5
        bl palloc2
        ldr r1, =_p
        str r0, [r1]
@ p^[0] := 1; p^[1] := 2; p^[2] := 3; p^[3] := 4; p^[4] := 5;

@ print_num(sum(p^));
        ldr r0, =_p
        ldr r4, [r0]
        ldr r1, [r4, #-4]
        mov r0, r4
        mov r10, #0
        bl _sum
        bl print_num
@ newline();
        bl newline
@ end.
.L4:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _p, 4, 4
        .section .note.GNU-stack
@ End
```

4.9 Avoiding Repeated Evaluation 2

This is to show that when bounds checking is enabled, given a pointer to an open array p, p↑[i] doesn't cause the p to be evaluated more than once - it is needed for the base address, and also to determine its size for use in the bounds checking to ensure that i is a valid subscript.

Code:

```
var p : pointer to array of integer;

begin
    newrow(p,3);
    p^[0] := 1;
end.
```

Output:

```
1
```

The abstract syntax tree below shows the base address getting evaluated into a temporary register, so that it is used both to retrieve the value in the array, and also to work out the size of the array for use by the BOUND operator.

```
@     p^[0] := 1;
@ <DEFTEMP 1, <LOADW, <GLOBAL _p>>>
@ <STOREW,
@   <CONST 1>,
@   <OFFSET,
@     <TEMP 1>,
@     <BINOP
@       Times,
@       <BOUND, <CONST 0>, <LOADW, <OFFSET, <TEMP 1>, <CONST -4>>>>,
@       <CONST 4>>>>
```

Assembly:

```
@ picoPascal compiler output
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@ newrow(p,3);
        mov r1, #4
        mov r0, #3
        bl palloc2
        ldr r1, =_p
        str r0, [r1]
@ p^[0] := 1;
        ldr r0, =_p
        ldr r4, [r0]
        mov r0, #1
        mov r1, #0
        ldr r2, [r4, #-4]
        cmp r1, r2
        ldrhs r0, =5
        blhs check
        mov r2, #4
        mul r1, r1, r2
        add r1, r4, r1
        str r0, [r1]
@ print_num(p^[0]); newline();
        ldr r0, =_p
        ldr r4, [r0]
        mov r0, #0
        ldr r1, [r4, #-4]
        cmp r0, r1
        ldrhs r0, =6
        blhs check
        mov r1, #4
        mul r0, r0, r1
        add r0, r4, r0
        ldr r0, [r0]
        bl print_num
        bl newline
@ end.
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _p, 4, 4
        .section .note.GNU-stack
@ End
```