# Simultaneous assignment

Mike Spivey, October 2022

This year's Christmas Assignment asks for an implementation of *simultaneous assignment.* a construct that allows the values of multiple expressions to be assigned, as if simultaneously, to multiple variables. The simplest useful example would be

```
x, y := y, x
```

which swaps the values of variables x and y without explicitly needing a temporary variable to hold the old value of one of the variables while the new value is being assigned, though behind the scenes we would expect it to be implemented that way. Another simple example might occur in a program for Fibonacci numbers:

```
x, y := y, x+y
```

If x and y have the values of $F_n$ and $F_{n+1}$, then this simultaneous assignment sets them to $F_{n+1}$ and $F_{n+2}$.

More complicated examples might use array or pointer variables. For example, the assignment

```
i, a[i] := a[i], i
```

could usefully swap the contents of the cell i with the cell *initially* denoted by a[i]. Implementing this faithfully will require us to ensure that the cell denoted by a[i] on the left-hand side is found using the *old* value of i, so is the same as the one used on the right. (Python does not do this.)

We will also want to use the assignment

```
a[i], a[j] := a[j], a[i]
```

to swap two element of an array, with the statement doing nothing if i and j are equal. This effect arises naturally if the implementation introduces an implicit temporary variable to hold the value of one element while it is being changed. In fact, we will use two registers as temps to hold both values, like this:

```
t := a[j]; u := a[i]; a[i] := t; a[j] := u
```

That is just as efficient when each value must be held in a register just before it is stored in memory.

 

Programs that manipulate pointer-linked structures commonly depend on *pointer rotations* to modify the structures. For example, in destructively reversing a linked list, we would iteratively nibble items from the front of the input list `p` and insert them at the start of an output list `q`. One such step is expressed by the simultaneous assignment,

```
p, p↑.tail, q := p↑.tail, q, p
```

Here, as in a previous example, we rely on the cells denoted on the left-hand side being the same as those on the right, with the old value of `p` used to find `p↑.next`. Another example is rotating a binary tree at the root, so that what was the left child of the root moves up, and what was the root node moves down to become its right child. If `p` points to the root, then this is achieved by the assignment,

```
p, p↑.left, p↑.left↑.right := p↑.left, p↑.left↑.right, p
```

That's a bit obscure, and the movements become clearer if we introduce a name `q` for the left child that moves up:

```
q := p↑.left;
p↑.left := q↑.right;
q↑.right := p;
p := q
```

So perhaps in this case, the simultaneous assignment is best avoided for the sake of a human reader, even if our compiler can make sense of it.

## 1 Initial implementation

As usual, we will follow the implementation through the compiler one pass at a time. In the *abstract syntax*, I chose to keep the existing assignment statement and add simultaneous assignment as an option alongside it. This simplifies the subsequent treatment, as copying of aggregates will be supported in a single assignment, but not in simultaneous assignments.

```
and stmt_guts =
    ...
  | Assign of expr * expr
  | SimAssign of (expr * expr) list
  | ...
```

Despite the concrete syntax which suggests a pair of lists, it seems best to make the abstract syntax a list of pairs. Checking that the two lists have the same length then becomes a clear syntactic matter.

No extensions are needed in the lexer, because all the punctuation marks used in simultaneous assignments are already present. In the parser, we need to relax the syntax of assignments to allow a list of variables on one side and a list of expressions on the other.

```
stmt1 :
    ...
  | var_list ASSIGN expr_list          { assign $1 $3 }
  | ...
```

```
var_list :
    variable                      { [$1] }
  | variable COMMA var_list       { $1 :: $3 } ;
```

Somewhat tediously, we must spell out the syntax of comma-separated lists of variables.

The semantic action uses a new helper function *assign*, defined in the preamble of parser.mly, that checks the two lists match in length, and builds either an *Assign* or a *SimAssign* node, depending on whether both lists have a single item.

```
let assign vs es =
  let n = List.length vs in
  if List.length es ≠ n then
    parse_error "wrong number of expressions";
  if n = 1 then
    Assign (List.hd vs, List.hd es)
  else
    SimAssign (List.combine vs es)
```

Semantic analysis (module *Check*) is not too hard: in each component of the assignment, we must recursively check the variable and expression, then check that they have the same type, and that this type is a scalar or pointer type.

```
let rec check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with
      ...
    | SimAssign pairs →
        let check (lhs, rhs) =
          let lt = check_expr lhs env
          and rt = check_expr rhs env in
          check_var lhs false;
          if not (same_type lt rt) then
            sem_error "type mismatch in simult assignment" [ ];
          if not (scalar lt || is_pointer lt) then
            sem_error "aggregates not allowed" [ ] in
        List.iter check pairs
    | ...
```

In generating intermediate code, we start to face some harder choices. Our basic approach is to evaluate all the right-hand side expressions into temporary registers, then to begin storing these values into locations denoted by the left-hand sides. For the temporary registers, we will use the same ⟨TEMP *n*⟩ nodes that are introduced in the common subexpression elimination pass of the compiler, but they will now start to appear in its input as well as its output.

To make a start, let's ignore the problem that the LHS addresses might change as values are stored, and just pre-evaluate the expressions on the right. The relevant code is added in the *TGen* module, in function *gen_stmt*.

```
(* gen_stmt – generate code for a statement *)
let rec gen_stmt s =
```

```
let code =
  match s.s_guts with
      ...
    | SimAssign pairs →
        let temps =
          (∗ List of temps for RHS values ∗)
          List.map (fun _ → Regs.new_temp 1) pairs in
        ⟨SEQ,
          (∗ Save the RHS values ∗)
          ⟨SEQ, @(List.map2 (fun (v, e) t →
            ⟨DEFTEMP t, gen_expr e⟩) pairs temps)⟩,
          (∗ Perform the stores ∗)
          ⟨SEQ, @(List.map2 (fun (v, e) t →
            let st =
              if size_of v.e_type = 1 then STOREC else STOREW in
            ⟨st, ⟨TEMP t⟩, g⟩) pairs temps)⟩⟩
```

Higher-order functions are helpful here, and we use both the familiar function *List.map*, whose type is

$$List.map : (\alpha \to \beta) \to \alpha\ list \to \beta\ list,$$

but also the function $List.map_2$ (which Haskell calls *zipWith*), with type

$$List.map_2 : (\alpha \to \beta \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list.$$

There are three steps:

- First, allocate a temp for each expression on the RHS. These temps all have a reference count of 1, because their values will each be used just once to assign to the corresponding LHS variable.
- Next, generate code that defines each temp with the value of the corresponding expression.
- Finally, generate code that stores the values from the temps into expressions on the LHS.

One further adjustment was added to the lab kit in advance of this assignment. The function *Tgen.do_proc* conducts a single procedure through the phases of generating intermediate code, optimising the code, and feeding it into the back-end function *Tran.translate*. It contains a call to *Regs.init* ( ) that initialises the register allocator. Because generating intermediate code can now involve allocating temps, that call must appear before the call to *gen_stmt* that produces code for the procedure body.

We need make no changes to the back end, for the compiler to translate successfully simple instances of simultaneous assignment.

## 2 Refining the implementation

The implementation presented so far can compile simple examples, but it cannot deal with examples where assigning to some of the LHS variables can affect the locations denoted by others. A naive solution to this would be to compute all the addresses of LHS variables into temps in addition to the values of RHS expressions. This would work for some examples, but has three problems:

- Register locals do not have a numeric address, so it isn't possible to compute their addresses into temps.
- Evaluating all the addresses into temps means that two temps will be needed for each component of the assignment, increasing the danger of running out of registers.
- Precomputing addresses into registers prevents the use of more powerful addressing modes in the store instructions.

To overcome these difficulties, we can formulate a plan where only some left-hand side variables have their addresses precomputed. For each variable, we will form two fragments of code: a fragment $f$ that may define a temp, and a fragment $g$ that uses the temp (if any) to form the address of the variable. We can consider various cases for a variable $v$:

- If $v$ is a simple variable, then $f$ can be a no-op, and we use the code *gen_addr v* for $g$. This case covers register variables without requiring them to have a numeric address.
- If $v$ is any other variable, then we can make $f$ compute its address into a temp:

  ⟨DEFTEMP *t*, *gen_addr v*⟩,

  then make $g$ be just ⟨TEMP *t*⟩. However, to improve the treatment of some common, simple cases, we can add two further rules.
- If $v$ has the form $a[e]$, where $a$ is an array variable, then we can make fragment $f$ compute the value of $e$ into a temp, then perform the subscript calculation in fragment $g$.
- If $v$ is a field selection $r.x$, then we can make fragment $f$ compute the address of record $r$, and have fragment $g$ add the offset of field $x$.

To implement the third and fourth rules, it's convenient to extract the address calculations for subscripts and record fields from the compiler function *gen_addr* as two subroutines *subscript* and *select* that can be reused in the implementation of simultaneous assignment.

Here's my implementation of *prep_addr*:

```
(* prep_addr – prepare LHS of simultaneous assignment *)
let prep_addr v =
  let s = size_of v.e_type in
  match v.e_guts with
      Variable _ →
        (* A simple variable – fixed address *)
        (s, ⟨NOP⟩, gen_addr v)
    | Sub ({ e_guts = Variable _ } as a, e1) →
        (* A subscript a[i] – save the value of i *)
        let t = Regs.new_temp 1 in
        (s, ⟨DEFTEMP t, gen_expr e1⟩,
          subscript a ⟨TEMP t⟩)
    | Select (r, x) →
```

```
            (* A selection r.x – save the address of r *)
            let t = Regs.new_temp 1 in
            (s, ⟨DEFTEMP t, gen_addr r⟩, select ⟨TEMP t⟩ x)
        | _ →
            (* General case – save the address of v *)
            let t = Regs.new_temp 1 in
            (s, ⟨DEFTEMP t, gen_addr v⟩, ⟨TEMP t⟩)
```

This function, in addition to the code fragments $f$ and $g$, returns also the size of the value being assigned, to help choose a store instruction. Here is an improved case for *gen_stmt* that uses the new treatment for the LHS.

```
(* gen_stmt – generate code for a statement *)
let rec gen_stmt s =
  let code =
    match s.s_guts with
        . . .
      | SimAssign pairs →
          let temps =
            (* List of temps for RHS values *)
            List.map (fun _ → Regs.new_temp 1) pairs in
          let addrs =
            (* List of (s, f, g) triples for the LHS *)
            List.map (fun (v, e) → prep_addr v) pairs in
          ⟨SEQ,
            (* Save the RHS values *)
            ⟨SEQ, @(List.map2 (fun (v, e) t →
              ⟨DEFTEMP t, gen_expr e⟩) pairs temps)⟩,
            (* Save what's needed for the LHS *)
            ⟨SEQ, @(List.map (fun (s, f, g) → f) addrs)⟩,
            (* Perform the stores *)
            ⟨SEQ, @(List.map2 (fun (s, f, g) t →
              let st = if s = 1 then STOREC else STOREW in
              ⟨st, ⟨TEMP t⟩, g⟩) addrs temps)⟩⟩
      | . . .
```

An applicative approach pays dividends here, because it permits us to compute separately the $f$ and $g$ trees for each element and then incorporate them later into a tree for the whole construct.

## 3 Evaluation

The simplest example is

```
x, y := y, x
```

where `x` and `y` are register variables. Our implementation produces the following code.

```
mov r6, r4
mov r4, r5
mov r5, r6
```

The values of x and y first become the values of two temps, with the temps simply sharing the same registers. Then there's an assignment of the value of y to the register variable x, which first spills the temp living in x into another register r6. The first mov instruction is the spill, and the second is the assignment to x. The last move assigns to y from the spilled temp.

Let's now look at a different example, the assignment i, a[i] := a[i], i, where i is a register variable. Our compiler generates the following code.[1]

```
ldr r5, =_a
ldr r6, [r5, r4, LSL #2]
mov r7, r4
mov r4, r6
str r7, [r5, r7, LSL #2]
```

Again, a temp shares r4 with the register variable i, and it is spilled before assigning to i. The spill could be avoided by swapping the last two instructions and eliminating r7, but some extra moves between registers are inevitable if the compiler does things in a fixed order.

For the swap a[i], a[j] := a[j], a[i], our compiler generates the attractive code,

```
ldr r6, =_a
ldr r7, [r6, r5, LSL #2]
ldr r8, [r6, r4, LSL #2]
str r7, [r6, r4, LSL #2]
str r8, [r6, r5, LSL #2]
```

(again, this is with the addressing improvements introduced in the solution to Lab 4, and with the taming of excessive CSE that formed part of last year's assignment.)

Two test cases provided as part of the solution implement destructive list reversal and tree rotation, the two pointer-based examples in the introduction. For destructive reversal, the assignment

```
p, p↑.tail, q := p↑.tail, q, p
```

results in the code

```
ldr r6, [r4, #4]
mov r7, r4
mov r4, r6
str r5, [r7, #4]
mov r5, r7
```

As before, this contains a redundant register-to-register move, but is otherwise acceptable.

Similarly, the tree rotation

```
p, p↑.left, p↑.left↑.right := p↑.left, p↑.left↑.right, p
```

results in code that has just one redundant move:

```
ldr r6, [r5, #4]
```

[1] The code given here uses a compiler based on the *solution* to Lab4, and is able to use the addressing mode that adds two registers with an optional shift. Participants basing their work on the unenhanced compiler will see less good code.

```
ldr r7, [r6, #8]
mov r8, r5
mov r5, r6
str r7, [r8, #4]
str r8, [r6, #8]
```

Common subexpression elimination has had a good effect here in avoiding redundant loads.

```
--- ../../labs/lab4s/check.ml   2023-10-05 17:08:52.117515957 +0100
+++ check.ml    2023-10-20 16:24:04.372065652 +0100
@@ -306,6 +306,17 @@
          if not (same_type lt rt) then
            sem_error "type mismatch in assignment" []

+    | SimAssign pairs ->
+        let check (lhs, rhs) =
+          let lt = check_expr lhs env
+          and rt = check_expr rhs env in
+          check_var lhs false;
+          if not (same_type lt rt) then
+            sem_error "type mismatch in simult assignment" [];
+          if not (scalar lt || is_pointer lt) then
+            sem_error "simult assignment not allowed for aggregates" [] in
+        List.iter check pairs
+
     | ProcCall (p, args) ->
         let rt = check_funcall p args env (ref None) in
         if rt <> voidtype then
--- ../../labs/lab4s/parser.mly 2023-10-05 17:08:52.133516753 +0100
+++ parser.mly  2023-10-20 16:24:04.372065652 +0100
@@ -29,6 +29,15 @@

 %{
 let const n t = make_expr (Constant (n, t))
+
+let assign vs es =
+  let n = List.length vs in
+  if List.length es <> n then
+    parse_error "Wrong number of expressions in simultaneous assignment";
+  if n = 1 then
+    Assign (List.hd vs, List.hd es)
+  else
+    SimAssign (List.combine vs es)
 %}

 %%
@@ -110,7 +119,7 @@

 stmt1 :
     /* empty */                        { Skip }
-  | variable ASSIGN expr               { Assign ($1, $3) }
+  | var_list ASSIGN expr_list         { assign $1 $3 }
   | name actuals                       { ProcCall ($1, $2) }
   | RETURN expr_opt                    { Return $2 }
   | IF expr THEN stmts elses END       { IfStmt ($2, $4, $5) }
@@ -185,6 +194,10 @@
   | variable DOT name                  { make_expr (Select ($1, $3)) }
   | variable ARROW                     { make_expr (Deref $1) } ;

+var_list :
+    variable                          { [$1] }
+  | variable COMMA var_list           { $1 :: $3 } ;
+
 typexpr :
     name                               { TypeName $1 }
   | ARRAY expr OF typexpr              { Array ($2, $4) }
--- ../../labs/lab4s/tgen.ml    2023-10-20 16:13:52.955228646 +0100
+++ tgen.ml     2023-10-20 16:25:46.832911439 +0100
@@ -93,14 +93,9 @@
                  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)>>>
+         subscript a (gen_expr i)
     | Select (r, x) ->
-         let d = get_def x in
-         <OFFSET, gen_addr r, <CONST (offset_of d)>>
+         select (gen_addr r) x
     | Deref p ->
         let null_check t =
           if not !boundchk then t else <NCHECK, t> in
@@ -108,6 +103,16 @@
     | String (lab, n) -> <GLOBAL lab>
     | _ -> failwith "gen_addr"

+and subscript a i =
+  let ty = base_type a.e_type in
+  <OFFSET, gen_addr a,
+    <BINOP Times,
+      if not !boundchk then i else <BOUND, i, <CONST (bound a.e_type)>>,
+      <CONST (size_of ty)>>>
+
+and select a x =
+  let d = get_def x in <OFFSET, a, <CONST (offset_of d)>>
+
 (* |gen_expr| -- tree for the value of an expression *)
 and gen_expr e =
   match e.e_value with
@@ -240,6 +245,30 @@
       <BINOP Minus, sel, <CONST lobound>>>
   end

+(* |prep_addr| -- prepare LHS of simultaneous assignment *)
+let prep_addr (v, _) =
+  (* Return (s, f, g) where s is the value size,
+     f defines any temps needed to preserve the address of v,
+     g produces the address of v for storing. *)
+  let s = size_of v.e_type in
+  match v.e_guts with
+      Variable _ ->
+        (* A simple variable -- fixed address *)
+        (s, <NOP>, gen_addr v)
+    | Sub ({ e_guts = Variable _ } as a, e1) ->
+        (* A subscript a[i] -- save the value of i *)
+        let t = Regs.new_temp () in
+        (s, <DEFTEMP t, gen_expr e1>,
+          subscript a <TEMP t>)
+    | Select (r, x) ->
+        (* A selection r.x -- save the address of r *)
+        let t = Regs.new_temp () in
+        (s, <DEFTEMP t, gen_addr r>, select <TEMP t> x)
+    | _ ->
+        (* General case -- save the address of v *)
+        let t = Regs.new_temp () in
+        (s, <DEFTEMP t, gen_addr v>, <TEMP t>)
+
 (* |gen_stmt| -- generate code for a statement *)
 let rec gen_stmt s =
```

```
   let code =
@@ -256,6 +285,24 @@
              gen_copy (gen_addr v) (gen_addr e) (size_of v.e_type)
            end

+       | SimAssign pairs ->
+           let temps =
+             (* List of temps for RHS values *)
+             List.map (fun _ -> Regs.new_temp ()) pairs in
+           let addrs =
+             (* List of (s, f, g) triples for the LHS *)
+             List.map prep_addr pairs in
+           <SEQ,
+             (* Save the RHS values *)
+             <SEQ, @(List.map2 (fun (v, e) t ->
+               <DEFTEMP t, gen_expr e>) pairs temps)>,
+             (* Save what's needed for the LHS *)
+             <SEQ, @(List.map (fun (s, f, g) -> f) addrs)>,
+             (* Perform the stores *)
+             <SEQ, @(List.map2 (fun (s, f, g) t ->
+               let st = if s = 1 then STOREC else STOREW in
+               <st, <TEMP t>, g>) addrs temps)>>
+
        | ProcCall (p, args) ->
            gen_call p args

--- ../../labs/lab4s/tree.ml    2023-10-05 17:08:52.125516354 +0100
+++ tree.ml     2023-10-20 16:24:04.372065652 +0100
@@ -33,6 +33,7 @@
      Skip
    | Seq of stmt list
    | Assign of expr * expr
+   | SimAssign of (expr * expr) list
    | ProcCall of name * expr list
    | Return of expr option
    | IfStmt of expr * stmt * stmt
@@ -137,6 +138,9 @@
        Skip -> fStr "(SKIP)"
      | Seq stmts -> fMeta "(SEQ$)" [fTail(fStmt) stmts]
      | Assign (e1, e2) -> fMeta "(ASSIGN $ $)" [fExpr e1; fExpr e2]
+     | SimAssign pairs ->
+         let f (e1, e2) = fMeta "($ $)" [fExpr e1; fExpr e2] in
+         fMeta "(SIMASSIGN $)" [fList(f) pairs]
      | ProcCall (p, aps) -> fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
      | Return (Some e) -> fMeta "(RETURN $)" [fExpr e]
      | Return None -> fStr "(RETURN)"
--- ../../labs/lab4s/tree.mli   2023-10-05 17:08:52.129516553 +0100
+++ tree.mli    2023-10-20 16:24:04.372065652 +0100
@@ -48,6 +48,7 @@
      Skip
    | Seq of stmt list
    | Assign of expr * expr
+   | SimAssign of (expr * expr) list
    | ProcCall of name * expr list
    | Return of expr option
    | IfStmt of expr * stmt * stmt
```

```
type list = pointer to cell;
  cell = record head: char; tail: list end;

proc reverse(a: list): list;
  var p, q: list;
begin
  p, q := a, nil;
  while p <> nil do
    p, p^.tail, q := p^.tail, q, p
  end;
  return q
end;

proc test();
  const mike = "mike";

  var i: integer; p, q: list;
begin
  p := nil; i := 0;
  while mike[i] <> chr(0) do
    new(q);
    p, q^.head, q^.tail, i := q, mike[i], p, i+1
  end;

  p := reverse(p);

  q := p;
  while q <> nil do
    print_char(q^.head);
    q := q^.tail
  end;
  newline()
end;

begin test() end.

(*<<
mike
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc reverse(a: list): list;
        .text
_reverse:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   p, q := a, nil;
        ldr r6, [fp, #40]
        mov r7, #0
        mov r4, r6
        mov r5, r7
.L3:
@   while p <> nil do
        cmp r4, #0
        beq .L5
@     p, p^.tail, q := p^.tail, q, p
        ldr r6, [r4, #4]
```

```
        mov r7, r4
        mov r4, r6
        str r5, [r7, #4]
        mov r5, r7
        b .L3
.L5:
@   return q
        mov r0, r5
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc test();
_test:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   p := nil; i := 0;
        mov r5, #0
        mov r4, #0
.L7:
@   while mike[i] <> chr(0) do
        ldr r7, =g1
        ldrb r0, [r7, r4]
        cmp r0, #0
        beq .L9
@     new(q);
        mov r0, #8
        bl new
        mov r6, r0
@     p, q^.head, q^.tail, i := q, mike[i], p, i+1
        ldrb r7, [r7, r4]
        add r8, r4, #1
        mov r9, r5
        mov r5, r6
        strb r7, [r6]
        str r9, [r6, #4]
        mov r4, r8
        b .L7
.L9:
@   p := reverse(p);
        mov r0, r5
        bl _reverse
        mov r5, r0
@   q := p;
        mov r6, r5
.L10:
@   while q <> nil do
        cmp r6, #0
        beq .L12
@     print_char(q^.head);
        ldrb r0, [r6]
        bl print_char
@     q := q^.tail
        ldr r6, [r6, #4]
        b .L10
.L12:
@   newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
```

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

        .data
g1:
        .byte 109, 105, 107, 101
        .byte 0
@ End
]]*)
```

```
type ptr = pointer to node;
  node = record data: integer; left, right: ptr end;

var u: array 10 of integer;

proc setu();
begin
  u[0] := 3; u[1] := 1; u[2] := 4; u[3] := 1;
  u[4] := 5; u[5] := 9; u[6] := 2; u[7] := 6;
  u[8] := 5; u[9] := 3
end;

proc mktree(a, b: integer): ptr;
  var m: integer; p: ptr;
begin
  if a >= b then
    return nil
  else
    m := (a+b) div 2;
    new(p);
    p^.data, p^.left, p^.right :=
      u[a], mktree(a+1, m+1), mktree(m+1, b);
    return p
  end
end;

proc print(p: ptr);
begin
  if p = nil then
    print_char('.')
  else
    print_num(p^.data); print(p^.left); print(p^.right)
  end
end;

(*
    A          B
   / \        / \
  B   3      1   A
 / \            / \
1   2          2   3
*)

proc sum(p: ptr): integer;
  var s: integer; q: ptr;
begin
  s, q := 0, p;
  while q <> nil do
    while q^.left <> nil do
      q, q^.left, q^.left^.right := q^.left, q^.left^.right, q
    end;
    s, q := s + q^.data, q^.right
  end;
  return s
end;

var t: ptr;

begin
  setu();
  t := mktree(0, 10);
  print(t); newline();
```

```
  print_num(sum(t)); newline()
end.

(*<<
3141...59...265...3..
39
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc setu();
        .text
_setu:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   u[0] := 3; u[1] := 1; u[2] := 4; u[3] := 1;
        ldr r4, =_u
        mov r0, #3
        str r0, [r4]
        mov r0, #1
        str r0, [r4, #4]
        mov r0, #4
        str r0, [r4, #8]
        mov r0, #1
        str r0, [r4, #12]
@   u[4] := 5; u[5] := 9; u[6] := 2; u[7] := 6;
        mov r0, #5
        str r0, [r4, #16]
        mov r0, #9
        str r0, [r4, #20]
        mov r0, #2
        str r0, [r4, #24]
        mov r0, #6
        str r0, [r4, #28]
@   u[8] := 5; u[9] := 3
        mov r0, #5
        str r0, [r4, #32]
        mov r0, #3
        str r0, [r4, #36]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc mktree(a, b: integer): ptr;
_mktree:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   if a >= b then
        ldr r0, [fp, #40]
        ldr r1, [fp, #44]
        cmp r0, r1
        blt .L4
@     return nil
        mov r0, #0
        b .L2
.L4:
@     m := (a+b) div 2;
        mov r1, #2
```

```
        ldr r0, [fp, #40]
        ldr r2, [fp, #44]
        add r0, r0, r2
        bl int_div
        mov r4, r0
@     new(p);
        mov r0, #12
        bl new
        mov r5, r0
@     p^.data, p^.left, p^.right :=
        ldr r6, [fp, #40]
        ldr r0, =_u
        ldr r7, [r0, r6, LSL #2]
        add r1, r4, #1
        add r0, r6, #1
        bl _mktree
        ldr r1, [fp, #44]
        mov r6, r0
        add r0, r4, #1
        bl _mktree
        str r7, [r5]
        str r6, [r5, #4]
        str r0, [r5, #8]
@     return p
        mov r0, r5
.L2:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc print(p: ptr);
_print:
        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 .L8
@     print_char('.')
        mov r0, #46
        bl print_char
        b .L6
.L8:
@     print_num(p^.data); print(p^.left); print(p^.right)
        ldr r0, [fp, #40]
        ldr r0, [r0]
        bl print_num
        ldr r0, [fp, #40]
        ldr r0, [r0, #4]
        bl _print
        ldr r0, [fp, #40]
        ldr r0, [r0, #8]
        bl _print
.L6:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc sum(p: ptr): integer;
_sum:
        mov ip, sp
        stmfd sp!, {r0-r1}
```

```
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   s, q := 0, p;
        mov r6, #0
        ldr r7, [fp, #40]
        mov r4, r6
        mov r5, r7
.L11:
@   while q <> nil do
        cmp r5, #0
        beq .L13
.L14:
@     while q^.left <> nil do
        ldr r6, [r5, #4]
        cmp r6, #0
        beq .L16
@       q, q^.left, q^.left^.right := q^.left, q^.left^.right, q
        ldr r7, [r6, #8]
        mov r8, r5
        mov r5, r6
        str r7, [r8, #4]
        str r8, [r6, #8]
        b .L14
.L16:
@     s, q := s + q^.data, q^.right
        ldr r0, [r5]
        add r6, r4, r0
        ldr r7, [r5, #8]
        mov r4, r6
        mov r5, r7
        b .L11
.L13:
@   return s
        mov r0, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   setu();
        bl _setu
@   t := mktree(0, 10);
        mov r1, #10
        mov r0, #0
        bl _mktree
        ldr r4, =_t
        str r0, [r4]
@   print(t); newline();
        bl _print
        bl newline
@   print_num(sum(t)); newline()
        ldr r0, [r4]
        bl _sum
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _u, 40, 4
        .comm _t, 4, 4
```

```
@ End
]]*)
```

```
var u: array 10 of integer;

proc unravel(a: integer);
  var i: integer;
begin
  i := a;
  while u[i] <> i do
    i, u[i] := u[i], i
  end
end;

proc setu();
begin
  u[0] := 3; u[1] := 1; u[2] := 4; u[3] := 1;
  u[4] := 5; u[5] := 9; u[6] := 2; u[7] := 6;
  u[8] := 5; u[9] := 3
end;

proc print();
  var i: integer;
begin
  for i := 0 to 9 do
    print_char(' '); print_num(u[i])
  end;
  newline()
end;

begin
  setu();
  unravel(8);
  print()
end.

(*<<
 3 1 4 3 5 5 2 6 8 9
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

@ proc unravel(a: integer);
        .text
_unravel:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   i := a;
        ldr r4, [fp, #40]
.L2:
@   while u[i] <> i do
        ldr r5, =_u
        ldr r6, [r5, r4, LSL #2]
        cmp r6, r4
        beq .L1
@     i, u[i] := u[i], i
        mov r7, r4
        mov r4, r6
        str r7, [r5, r7, LSL #2]
        b .L2
.L1:
```

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

@ proc setu();
_setu:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   u[0] := 3; u[1] := 1; u[2] := 4; u[3] := 1;
        ldr r4, =_u
        mov r0, #3
        str r0, [r4]
        mov r0, #1
        str r0, [r4, #4]
        mov r0, #4
        str r0, [r4, #8]
        mov r0, #1
        str r0, [r4, #12]
@   u[4] := 5; u[5] := 9; u[6] := 2; u[7] := 6;
        mov r0, #5
        str r0, [r4, #16]
        mov r0, #9
        str r0, [r4, #20]
        mov r0, #2
        str r0, [r4, #24]
        mov r0, #6
        str r0, [r4, #28]
@   u[8] := 5; u[9] := 3
        mov r0, #5
        str r0, [r4, #32]
        mov r0, #3
        str r0, [r4, #36]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

@ proc print();
_print:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   for i := 0 to 9 do
        mov r4, #0
        mov r5, #9
.L7:
        cmp r4, r5
        bgt .L8
@     print_char(' '); print_num(u[i])
        mov r0, #32
        bl print_char
        ldr r0, =_u
        ldr r0, [r0, r4, LSL #2]
        bl print_num
        add r4, r4, #1
        b .L7
.L8:
@   newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

pmain:
        mov ip, sp
```

```
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   setu();
        bl _setu
@   unravel(8);
        mov r0, #8
        bl _unravel
@   print()
        bl _print
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .comm _u, 40, 4
@ End
]]*)
```