Module Display

1. Visualization of Caml objects.
Different memory adresses are displayed as different nodes. Nodes are numbered from 1 to n, using the global counter *node_number*. All the objects already traversed are stored in a table, in order to be shared in the graph, as they are in the memory. That table *node_table* is simply an association list, but in which keys are compared using *physical equality* (*mem_assq* and *assq*).

```
let node_number = ref 0

let node_table = ref ([] : (Obj.t × int) list)

let in_table t = List.mem_assq t !node_table

let assoc_node t = List.assq t !node_table

let new_node t =
  incr node_number;
  node_table := (t, !node_number) :: !node_table;
  !node_number
```

2. When traversing recursively the Caml objects, we stop as soon as an object is not a structured object, even for infixes, closures and objects — in the sense of OO. Therefore, we do not use *Obj.no_scan_tag* but the following lower value.

```
let no_scan_tag = 248
```

3. Here we define the functions that print the objects. The value of a field is printed if it is not a block (otherwise, it will appear as a pointer to another node). A node is displayed by the function *output_node*. If it is a structured block, then it is displayed as a record. In the other cases, it is displayed in different ways, following its tag, by the function *output_non_block*. An edge, representing a pointer, is produced by the function *output_edge*, starts from the right side of a field and points to the left side a the first field of the destination object.

```
let output_field c i x =
  fprintf c "<f%d>␣" i;
  if ¬ (is_block x) then fprintf c "%d" (magic x)
```

```
let output_non_block c t =
  let tg = tag t in
  if tg = closure_tag then
    output_string c "<f0> closure"
  else if tg = object_tag then
    output_string c "<f0> object"
  else if tg = abstract_tag then
    output_string c "<f0> abstract"
  else if tg = string_tag then
    fprintf c "<f0> \\\"%s\\\"" (magic t : string)
  else if tg = double_tag then
    fprintf c "<f0> %f" (magic t : float)
  else if tg = double_array_tag then
    fprintf c "<f0> double_array"
  else if tg = final_tag then
    fprintf c "<f0> final"
  else
    fprintf c "<f0> ???"

let output_node c n t =
  fprintf c "  \"node%d\" [ label = \"" n;
  if is_block t then
    if tag t < no_scan_tag then
      let n = size t - 1 in
      for i = 0 to n do
        output_field c i (field t i);
        if i < n then fprintf c " | "
      done
    else
      output_non_block c t
  else
    fprintf c "%d" (magic t);
  fprintf c "\" shape = \"record\" ];\n"

let output_edge c (n1, f1) n2 =
  fprintf c "  \"node%d\":f%d -> \"node%d\":f0 [ ];\n" n1 f1 n2
```

4. We come to the main part of the program. It is made of a function *traverse*, which takes a Caml object t as argument, produces a node for it and traverses it recursively. It works as follows:

1. if the object t is in the global table, we do nothing;

2. if not, we do the following three steps:

   (a) we create a new node n for it with *new_node*;

   (b) we call recursively *traverse* on all its fields which are blocks;

   (c) we produce the pointers leaving t i.e. the node n.

```
let rec traverse c t =
  if ¬ (in_table t) then begin
    let n = new_node t in
    output_node c n t;
    if is_block t ∧ tag t < no_scan_tag then begin
      for i = 0 to size t − 1 do
        let f = field t i in
        if is_block f then traverse c f
      done;
      for i = 0 to size t − 1 do
        let f = field t i in
        if is_block f then output_edge c (n, i) (assoc_node f)
      done
    end
  end
```

5. The main function, *display*, only consists in opening the output file, writing the header of the *dot* file, calling *traverse* on the given object, and writing the trailer of the file. The part of the job concerning the file is shared in a common function *produce_file*, which is instantiated twice to make *display* and *display_list*.

```
let reset () =
  node_number := 0;
  node_table := []

let produce_file file f x =
  let c = open_out file in
  output_string c "digraph␣g␣{\n
   ␣␣graph␣[␣rankdir␣=␣\"LR\"␣];
   ␣␣node␣[␣fontsize␣=␣\"16\"␣shape␣=␣\"ellipse\"␣];
   ␣␣edge␣[␣];\n\n";
  f c x;
  output_string c "\n\n}\n";
  close_out c
```

```
let display f x =
  reset();
  produce_file f traverse (repr x)

let display_list f l =
  produce_file f
    (fun c → List.iter (fun x → traverse c (repr x))) l
```

Module Example

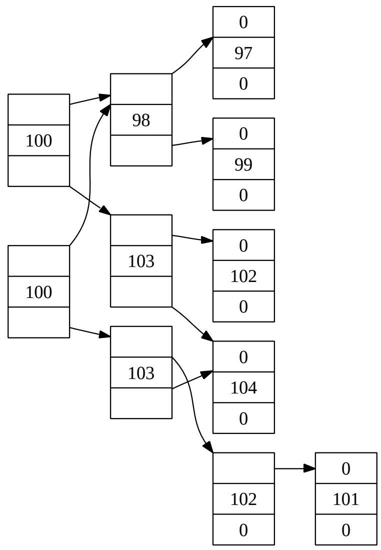
6. This is a short program to demonstrate the use of the module *Display*. We first construct a tree x containing the characters `'a'` to `'h'` except `'e'`, and then the tree y resulting from the insertion of `'e'` into x.

```
open Display

type α tree = E | T of α tree × α × α tree

let b = T (T(E, 'a', E), 'b', T(E, 'c', E))
let f = T (E, 'f', E)
let h = T (E, 'h', E)
let g = T (f, 'g', h)
let x = T (b, 'd', g)
let y = T (b, 'd', T(T(T(E, 'e', E), 'f', E), 'g', h))

let _ = display_list "example.dot" [x; y]
```

7. Once this short program is run, a file `"example.dot"` is created. Then a PostScript file can be produced using the *dot* tool, with the following command:

```
dot -Tps -o example.ps example.dot
```

The PostScript file looks like:


0
97
0
98
100
0
99
0
103
0
102
0
100
103
0
104
0
102
0
0
101
0

Index