Interface for module Sml_ng

1. SML/NG documents. A document is a list of item; an item is either a text of non-space characters, or a space character or a tag containing a list of item.

type *color* =
  | *Red* | *Green* | *Blue* | *Cyan* | *Magenta* | *Yellow* | *Black* | *White*
  | *Any* (* for root context *)

type *tag* =
  | *B*
  | *EM*
  | *I*
  | *PL*
  | *S*
  | *TT*
  | *U*
  | *Size* of *int* (* 0..9 ; -1 for root context *)
  | *Color* of *color*

type *white_space* = *SPC* | *TAB* | *CR* | *LF*

type *item* =
  | *Text* of *string*
  | *Wspace* of *white_space*
  | *Tag* of *tag* × *item list*

type *document* = *item list*

2. Contexts.

type *context* = {
  b : *bool*;
  em : *bool*;
  i : *bool*;
  s : *bool*;
  tt : *bool*;
  u : *int*; (* 0..3 *)
  size : *int*;
  color : *color* }

3. The interpretation of a document is a stream of decorated chars (which pairs of chars and contexts).

type *decorated_char* = *char* × *context*

type *interpretation* = *decorated_char Stream.t*

Interface for module Output

4. This module provides functions to output documents in ASCII format.

```
open Sml_ng
```

5. Pretty-printing of a document on stdout. *size* returns the size of a document in bytes.

```
val output_document : document → unit
```

```
val size : document → int
```

6. These functions are exported to be re-used in module *Optim*.

```
val open_tag : tag → unit
val close_tag : tag → unit
```

7. Dump of interpretations (for debugging purposes only).

```
val dump_char : out_channel → decorated_char → unit
```

```
val dump : out_channel → interpretation → unit
```

Module Output

8. Output of a document. Used to output the original document only, if timeout is reached or optimization didn't give something better. Notice that a color shouldn't be *Any* and that size shouldn't be -1 (an assertion checks it).

```
open Sml_ng
```

```
let print_color = function
  | Red → print_char 'r'
  | Green → print_char 'g'
  | Blue → print_char 'b'
  | Cyan → print_char 'c'
  | Magenta → print_char 'm'
  | Yellow → print_char 'y'
  | Black → print_char 'k'
  | White → print_char 'w'
  | Any → assert false
```

```
let open_tag = function
  | B → print_string "<B>"
  | EM → print_string "<EM>"
  | I → print_string "<I>"
  | PL → print_string "<PL>"
  | S → print_string "<S>"
```

```
  | TT → print_string "<TT>"
  | U → print_string "<U>"
  | Size n → assert (n ≢ − 1); print_char '<'; print_int n; print_char '>'
  | Color c → print_char '<'; print_color c; print_char '>'
let close_tag = function
  | B → print_string "</B>"
  | EM → print_string "</EM>"
  | I → print_string "</I>"
  | PL → print_string "</PL>"
  | S → print_string "</S>"
  | TT → print_string "</TT>"
  | U → print_string "</U>"
  | Size n → assert (n ≢ − 1); print_string "</"; print_int n; print_char '>'
  | Color c → print_string "</"; print_color c; print_char '>'
let rec output_item = function
  | Text s → print_string s
  | Wspace SPC → print_char ' '
  | Wspace TAB → print_char '\t'
  | Wspace CR → print_char '\013'
  | Wspace LF → print_char '\010'
  | Tag (t, d) → open_tag t; List.iter output_item d; close_tag t
let output_document = List.iter output_item
```

9. Size of a document (when output, in bytes).

```
let tag_size = function
  | EM | PL | TT → 9
  | _ → 7
let rec size_item n = function
  | Text s → n + String.length s
  | Wspace _ → succ n
  | Tag (t, d) → n + tag_size t + size d
and size d = List.fold_left size_item 0 d
```

10. Dumping of interpretations.

```
open Printf
let color_name = function
  | Red → 'r'
  | Green → 'g'
  | Blue → 'b'
  | Cyan → 'c'
  | Magenta → 'm'
  | Yellow → 'y'
```

```
  | Black → 'k'
  | White → 'w'
  | Any → '-'

let dump_char ch (c, ctx) =
  fprintf ch
    "'%s'␣(code␣%d)␣B=%b␣EM=%b␣I=%b␣S=%b␣TT=%b␣u=%d␣size=%d␣color=%c\n"
    (Char.escaped c) (Char.code c)
    ctx.b ctx.em ctx.i ctx.s ctx.tt ctx.u ctx.size (color_name ctx.color)

let dump ch s = Stream.iter (dump_char ch) s; flush ch
```

Module Parser (Yacc)

11. Yacc grammar for SML/NG documents. Immediate. One remark, yet: when checking that the closing tag matched the opening one, we use physical equality since tags are statically allocated in the lexer.

```
open Sml_ng

%token <Sml_ng.tag> OPEN
%token <Sml_ng.tag> CLOSE
%token <string> TEXT
%token <Sml_ng.white_space> WSP
%token EOF

%type <Sml_ng.document> document
%start document

document ::=
  items EOF { $1 }

items ::=
  /* epsilon */ { [] }
| item items { $1 :: $2 }

item ::=
  TEXT
    { Text $1 }
| WSP
    { Wspace $1 }
| OPEN items CLOSE
```

```
{ let t1 = $1 and t2 = $3 in
  if t1 ≢ t2 then raise Parsing.Parse_error;
  Tag (t1, $2) }
```

Module Lexer (Lex)

12. Lexer for SML/NG documents. A little trick here: many spaces are replaced by a single one outside of any `TT` tag (a counter *tt_level* counting the `TT` depth is used for that purpose). Of course, some other spaces may be compressed if, for instance, a `PL` tag is used inside a `TT` tag; this will be treated later. But this trick gives us an "initial" document already a little smaller; note that this document will be printed on timeout, not the exact original one.

```
{
  open Lexing
  open Parser
  open Sml_ng
```

13. Statically allocated tags (see remark in module *Parser*).

```
let spc = WSP SPC
let tab = WSP TAB
let lf = WSP LF
let cr = WSP CR
let size_0 = Size 0
let size_1 = Size 1
let size_2 = Size 2
let size_3 = Size 3
let size_4 = Size 4
let size_5 = Size 5
let size_6 = Size 6
let size_7 = Size 7
let size_8 = Size 8
let size_9 = Size 9
let red = Color Red
let green = Color Green
let blue = Color Blue
let cyan = Color Cyan
let magenta = Color Magenta
let yellow = Color Yellow
let black = Color Black
let white = Color White
```

14. The `TT` depth.

```
  let tt_level = ref 0
}
```

15. Shortcuts for characters.

```
let spc = ' '
let tab = '\t'
let lf = '\n'
let cr = '\r'
let space = spc | tab | lf | cr
let char = ['\033'-'\059' '\061' '\063'-'\126']
```

16. Rules.

```
rule token = parse
  | spc { if !tt_level ≡ 0 then space lexbuf else spc }
  | tab { if !tt_level ≡ 0 then space lexbuf else tab }
  | lf { if !tt_level ≡ 0 then space lexbuf else lf }
  | cr { if !tt_level ≡ 0 then space lexbuf else cr }
  (* Opening tags *)
  | "<B>" { OPEN B }
  | "<EM>" { OPEN EM }
  | "<I>" { OPEN I }
  | "<PL>" { OPEN PL }
  | "<S>" { OPEN S }
  | "<TT>" { incr tt_level; OPEN TT }
  | "<U>" { OPEN U }
  | "<0>" { OPEN size_0 }
  | "<1>" { OPEN size_1 }
  | "<2>" { OPEN size_2 }
  | "<3>" { OPEN size_3 }
  | "<4>" { OPEN size_4 }
  | "<5>" { OPEN size_5 }
  | "<6>" { OPEN size_6 }
  | "<7>" { OPEN size_7 }
  | "<8>" { OPEN size_8 }
  | "<9>" { OPEN size_9 }
  | "<r>" { OPEN red }
  | "<g>" { OPEN green }
  | "<b>" { OPEN blue }
  | "<c>" { OPEN cyan }
  | "<m>" { OPEN magenta }
  | "<y>" { OPEN yellow }
  | "<k>" { OPEN black }
  | "<w>" { OPEN white }
  (* Closing tags *)
```

```
  | "</B>" { CLOSE B }
  | "</EM>" { CLOSE EM }
  | "</I>" { CLOSE I }
  | "</PL>" { CLOSE PL }
  | "</S>" { CLOSE S }
  | "</TT>" { decr tt_level; CLOSE TT }
  | "</U>" { CLOSE U }
  | "</0>" { CLOSE size_0 }
  | "</1>" { CLOSE size_1 }
  | "</2>" { CLOSE size_2 }
  | "</3>" { CLOSE size_3 }
  | "</4>" { CLOSE size_4 }
  | "</5>" { CLOSE size_5 }
  | "</6>" { CLOSE size_6 }
  | "</7>" { CLOSE size_7 }
  | "</8>" { CLOSE size_8 }
  | "</9>" { CLOSE size_9 }
  | "</r>" { CLOSE red }
  | "</g>" { CLOSE green }
  | "</b>" { CLOSE blue }
  | "</c>" { CLOSE cyan }
  | "</m>" { CLOSE magenta }
  | "</y>" { CLOSE yellow }
  | "</k>" { CLOSE black }
  | "</w>" { CLOSE white }
  (* Other chars *)
  | char+ { TEXT (lexeme lexbuf) }
  | eof { EOF }
  | _ { raise Parsing.Parse_error }
and space = parse
  | space+ { spc }
  | eof { lexbuf.lex_curr_pos ← lexbuf.lex_start_pos; spc }
  | _ { lexbuf.lex_curr_pos ← lexbuf.lex_start_pos; spc }
```

Interface for module Parse

17. SGML/NG documents parsers, from a channel and from stdin, respectively.

```
val from_channel : in_channel → Sml_ng.document
val from_stdin : unit → Sml_ng.document
```

Module Parse

18. SGML/NG documents parsers.

```
let from_channel c =
  let lb = Lexing.from_channel c in
  try
    Parser.document Lexer.token lb
  with Parsing.Parse_error →
    Printf.eprintf "Parse error at character %d\n" (Lexing.lexeme_start lb);
    exit 1

let from_stdin () = from_channel stdin
```

Interface for module Interp

19. SML/NG interpretation.

```
open Sml_ng

val interp_document : document → interpretation
```

20. Utility exports for module *Optim*. *space_context* computes the space context corresponding to a given context.

```
val root_context : context
val space_context : context → context
```

Module Interp

21. Interpretation.

```
open Sml_ng
```

22. Root contexts and space contexts.

```
let root_context =
  { b = false; em = false; i = false; s = false; tt = false;
    u = 0; size = − 1 (∗ any ∗); color = Any (∗ any ∗) }

let space_context ctx =
  { ctx with s = false; em = false; i = false; b = false;
      color = if ctx.u ≡ 0 then White else ctx.color }
```

23. The interpretation of a sequence of non-space characters is the stream of these characters, all decorated with the current context *ctx*.

```
let interp_string ctx s =
  let n = String.length s in
```

```
  let rec interp i =
    if i ≡ n then [⟨ ⟩] else [⟨ '(s.[i], ctx); interp (succ i) ⟩]
  in
  interp 0
```

24. Interpretation of a space.

```
let char_of_space = function
  | SPC → ' '
  | TAB → '\t'
  | LF → '\n'
  | CR → '\r'
```

25. The interpretation of a document is computed by the recursive functions *interp* (for an item) and *interp_many* (for a list of items). *ctx* is the current context. *lsp* is the last space context, of type *context option*; if *None*, the previous outut character was not a space; if *Some c*, it was a space with context *c*.

```
let interp_document d =
  let rec interp lsp ctx = function
    | Text s →
        interp_string ctx s, None
    | Wspace w →
        let sp_ctx = space_context ctx in
        let c = char_of_space w in
        if ctx.tt then
          [⟨ '(c, sp_ctx) ⟩], None
        else begin match lsp with
          | Some sp_ctx' when sp_ctx' = sp_ctx →
              [⟨ ⟩], lsp
          | _ →
              [⟨ ' (' ', sp_ctx) ⟩], (Some sp_ctx)
        end
    | Tag (B, d) →
        interp_many lsp { ctx with b = true } d
    | Tag (EM, d) →
        interp_many lsp (if ctx.s then ctx else { ctx with em = ¬ ctx.em }) d
    | Tag (I, d) →
        interp_many lsp { ctx with i = true } d
    | Tag (PL, d) →
        interp_many lsp { ctx with u = 0; s = false; em = false; i = false;
                                   b = false; tt = false } d
    | Tag (S, d) →
        interp_many lsp { ctx with s = true; em = false } d
    | Tag (TT, d) →
        interp_many lsp { ctx with tt = true } d
```

```
    | Tag (U, d) →
        interp_many
          lsp (if ctx.u < 3 then {ctx with u = succ ctx.u} else ctx) d
    | Tag ((Size n), d) →
        interp_many lsp { ctx with size = n } d
    | Tag ((Color c), d) →
        interp_many lsp { ctx with color = c } d
  and interp_many lsp ctx = function
    | [] →
        [⟨ ⟩], lsp
    | item :: items →
        let st, lsp′ = interp lsp ctx item in
        let st′, lsp″ = interp_many lsp′ ctx items in
        [⟨ st; st′ ⟩], lsp″
  in
  fst (interp_many None root_context d)
```

Interface for module Optim

26. The optimizer. It works on the interpretation, destructing it while working.

```
open Sml_ng
```

```
type optimized_document
```

```
val optim_document : interpretation → optimized_document
```

27. Size computation and output of optimized documents.

```
val size : optimized_document → int
val output_optimized_document : optimized_document → unit
```

Module Optim

28. The optimizer.

```
open Sml_ng
open Output
open Interp
```

29. The optimized document is the list, in reverse order, of the following items.

```
type optimized_item = Char of char | Open of tag | Close of tag
```

```
type optimized_document = optimized_item list
```

30. To output an optimized document, we reverse the list and then output the items; it could be done directly, but this way is tail recursive.

```
let output_optimized_document od =
  let output_item = function
    | Char c → print_char c
    | Open t → open_tag t
    | Close t → close_tag t
  in
  List.iter output_item (List.rev od)
```

31. The optimizer is based on the following function *switch*, which finds out how to get the new context ctx' from the old one ctx. *opens* is the stack of opened tags (in reverse order, the most recent coming first); they are paired with the context right before their opening. *acc* is the current optimized document (the list of output items, in reverse order).

We proceed naively, dealing with the context features one by one (and thus *switch* is recursive). For properties to unset like B, we close all the necessary tags, until the property disappear, using the function *close_until* (which takes a property to be satisfied as first argument). To set a property, we open the corresponding tag. For sizes and colors, we apply the following heuristic: if closing the last opened tag gives us the expected size (or color) we do that; otherwise, we open the corresponding tag.

```
let rec switch opens acc ctx ctx' =
  if ctx = ctx' then (* nothing to do *)
    opens, acc
  else if ctx.b ∧ ¬ ctx'.b then (* B to unset *)
    close_until (fun c → ¬ c.b) opens acc ctx'
  else if ¬ ctx.b ∧ ctx'.b then (* B to set *)
    switch ((B, ctx) :: opens) (Open B :: acc) { ctx with b = true } ctx'
  else if ctx.em ∧ ¬ ctx'.em then (* EM to unset *)
    close_until (fun c → ¬ c.em) opens acc ctx'
  else if ¬ ctx.em ∧ ctx'.em then (* EM to set *)
    switch ((EM, ctx) :: opens) (Open EM :: acc) { ctx with em = true } ctx'
  else if ctx.i ∧ ¬ ctx'.i then (* I to unset *)
    close_until (fun c → ¬ c.i) opens acc ctx'
  else if ¬ ctx.i ∧ ctx'.i then (* I to set *)
    switch ((I, ctx) :: opens) (Open I :: acc) { ctx with i = true } ctx'
  else if ctx.s ∧ ¬ ctx'.s then (* S to unset *)
    close_until (fun c → ¬ c.s) opens acc ctx'
  else if ¬ ctx.s ∧ ctx'.s then (* S to set *)
    switch ((S, ctx) :: opens) (Open S :: acc) { ctx with s = true } ctx'
  else if ctx.tt ∧ ¬ ctx'.tt then (* TT to unset *)
    close_until (fun c → ¬ c.tt) opens acc ctx'
  else if ¬ ctx.tt ∧ ctx'.tt then (* TT to set *)
    switch ((TT, ctx) :: opens) (Open TT :: acc) { ctx with tt = true } ctx'
  else if ctx.u > ctx'.u then (* U to decrease *)
```

```
    close_until (fun c → c.u ≡ ctx'.u) opens acc ctx'
  else if ctx.u < ctx'.u then (* U to increase *)
    switch ((U, ctx) :: opens) (Open U :: acc) { ctx with u = succ ctx.u } ctx'
  else if ctx.size ≢ ctx'.size then (* Size to change *)
    let s = ctx'.size in (* desired size *)
    if s ≡ −1 then
      close_until (fun c → c.size ≡ −1) opens acc ctx'
    else
      (match opens with
          | (ti, ctxi) :: opens' when ctxi.size ≡ s → (* we're lucky *)
                switch opens' (Close ti :: acc) ctxi ctx'
          | _ →
                let t = Size s in
                switch ((t, ctx) :: opens) (Open t :: acc) {ctx with size = s} ctx')
  else if ctx.color ≢ ctx'.color then (* Color to change *)
    let c = ctx'.color in (* desired color *)
    if c ≡ Any then
      close_until (fun c → c.color ≡ Any) opens acc ctx'
    else
      (match opens with
          | (ti, ctxi) :: opens' when ctxi.color ≡ c → (* we're lucky *)
                switch opens' (Close ti :: acc) ctxi ctx'
          | _ →
                let t = Color c in
                switch ((t, ctx) :: opens) (Open t :: acc) {ctx with color = c} ctx')
  else (* necessarily ctx = ctx' *)
    assert false (* opens, acc *)

and close_until k opens acc ctx' = match opens with
  | [] →
      switch opens acc root_context ctx'
  | (ti, ctxi) :: opens' →
      if k ctxi then
        switch opens' (Close ti :: acc) ctxi ctx'
      else
        close_until k opens' (Close ti :: acc) ctx'
```

32. At the very end of the optimization, we close all the opened tags with the following function.

```
let rec close_all acc = function
  | [] → acc
  | (t, _) :: opens → close_all (Close t :: acc) opens
```

33. Test for a space character.

```
let is_space = function
```

```
  | ' ' | '\t' | '\r' | '\n' → true
  | _ → false
```

34. *update ctx ctx′* builds the context to reach from current context *ctx* when *ctx′* is the space context of the next character (thus a space).

```
let update ctx ctx′ =
  { ctx with color = if ctx′.u > 0 then ctx′.color else ctx.color;
             u = ctx′.u; tt = ctx′.tt; size = ctx′.size }
```

35. The optimizer itself.

```
let optim_document st =
  let rec loop opens acc ctx = parser
    | [⟨ '(c, ctx′); st ⟩] →
        let ctx′ = if is_space c then update ctx ctx′ else ctx′ in
        if ctx′ = ctx then
          loop opens (Char c :: acc) ctx st
        else
          let opens′, acc′ = switch opens acc ctx ctx′ in
          loop opens′ (Char c :: acc′) ctx′ st
    | [⟨ ⟩] →
        close_all acc opens
  in
  loop [] [] root_context st
```

36. Size of the optimized document.

```
let size_item n = function
  | Char _ → succ n
  | Open (EM | PL | TT) → n + 4
  | Open _ → n + 3
  | Close (EM | PL | TT) → n + 5
  | Close _ → n + 4
let size od = List.fold_left size_item 0 od
```

Module Main

37. Set the *Alarm* signal according to the given time limit. 5 seconds should be enough to output the solution.

```
let tlimit = int_of_string Sys.argv.(1)
let _ = Unix.alarm (tlimit − 5)
```

38. Read the initial document from stdin, store it in *init_doc*, and output it on timeout.

```
let init_doc = Parse.from_stdin ()
```

```
let timeout _ = Output.output_document init_doc; flush stdout; exit 0
let _ = Sys.set_signal Sys.sigalrm (Sys.Signal_handle timeout)
```

39. Main program. We compute first the interpretation *id* of the document, then the optimized document *od* from it; if it is smaller than the original document, we output it; otherwise we output the original document.

```
let main () =
  let id = Interp.interp_document init_doc in
  let od = Optim.optim_document id in
  let old_size = Output.size init_doc in
  let new_size = Optim.size od in
  if new_size < old_size then begin
    ignore (Sys.set_signal Sys.sigalrm Sys.Signal_ignore);
    Optim.output_optimized_document od
  end else
    Output.output_document init_doc

let _ =
  Printexc.catch main ()
```

Index