Compilateur de Petit Rust

Théo DELEMAZURE – Guillaume BRESSAN

LEXER

Analyse de la syntaxe : Lexer

```
let keyword_list = [ … ]
…
rule token = parse
 |'\n' {new_line lexbuf;token lexbuf}
 |[' ' '\t'] {token lexbuf}
 |['0'-'9']+ as i {Tint (int_of_string
i)}
 |"//"  {comment_line lexbuf}
 |"/*" { multicomment lexbuf; token
lexbuf}
 |'"' {Tstring (parse_string lexbuf)}
 |ident as id {let s = Lexing.lexeme
lexbuf in
           try List.assoc s keyword_list
           with Not_found -> IDENT id}
...
 |eof {EOF}
 |_ as r{ raise (Lexing_error (...))}
```

Analyse de la syntaxe : Lexer

```
let keyword_list = [ … ]
…
rule token = parse
 |'\n' {new_line lexbuf;token lexbuf}
 |[' ' '\t'] {token lexbuf}
 |['0'-'9']+ as i {Tint (int_of_string
i)}
 |"//"  {comment_line lexbuf}
 |"/*" { multicomment lexbuf; token
lexbuf}
 |'"' {Tstring (parse_string lexbuf)}
 |ident as id {let s = Lexing.lexeme
lexbuf in
           try List.assoc s keyword_list
           with Not_found -> IDENT id}
...
 |eof {EOF}
 |_ as r{ raise (Lexing_error (...))}
```

```
and comment_line = parse
 |'\n' {token lexbuf}
 |_ { comment_line lexbuf}
 | eof {EOF}
```

Analyse de la syntaxe : Lexer

```
let keyword_list = [ … ]
…
rule token = parse
 |'\n' {new_line lexbuf;token lexbuf}
 |[' ' '\t'] {token lexbuf}
 |['0'-'9']+ as i {Tint (int_of_string
i)}
 |"//"  {comment_line lexbuf}
 |"/*" { multicomment lexbuf; token
lexbuf}
 |'"' {Tstring (parse_string lexbuf)}
 |ident as id {let s = Lexing.lexeme
lexbuf in
           try List.assoc s keyword_list
           with Not_found -> IDENT id}
...
 |eof {EOF}
 |_ as r{ raise (Lexing_error (...))}
```

```
and comment_line = parse
 |'\n' {token lexbuf}
 |_ { comment_line lexbuf}
 | eof {EOF}


and multicomment = parse
 |"*/" {()}
 |"/*" {multicomment lexbuf;
        multicomment lexbuf;}
 |_ {multicomment lexbuf}
 |eof { raise (Lexing_error (...)}
```

Analyse de la syntaxe : Lexer

```
let keyword_list = [ … ]
…
rule token = parse
 |'\n' {new_line lexbuf;token lexbuf}
 |[' ' '\t'] {token lexbuf}
 |['0'-'9']+ as i {Tint (int_of_string
i)}
 |"//"  {comment_line lexbuf}
 |"/*" { multicomment lexbuf; token
lexbuf}
 |'"' {Tstring (parse_string lexbuf)}
 |ident as id {let s = Lexing.lexeme
lexbuf in
           try List.assoc s keyword_list
           with Not_found -> IDENT id}
...
 |eof {EOF}
 |_ as r{ raise (Lexing_error (...))}
```

```
and parse_string = parse
 |'"' {""}
 |"\\\\" {"\\"^(parse_string lexbuf)}
 |"\\n" {"\n"^(parse_string lexbuf)}
 |"\\\"" {"\""^(parse_string lexbuf)}
 |['\\'] {raise (Lexing_error  (...))}
 |_ as c {c^(parse_string lexbuf)}
 |eof {raise (Lexing_error  (...))}
```

AST

Analyse de la syntaxe : Ast

```
type mutident = bool*ident

type istipe = None |T of tipe

type tipe_desc =
   Tx of ident
 | Tvec of ident*tipe
 | Tref of tipe
 | Trefmut of tipe

type argument = {nom_arg : mutident; typ_arg : tipe}

type pdfun = {nom_pfun : ident; arg_pfun : argument list; typ_pfun : istipe; bloc_pfun : pblock}

type pdstruct = {nom_pstruct : ident; val_pstruct : (ident*tipe) list}

type pdecl_desc = PDfun of  pdfun | PDstruct of pdstruct

type pdecl = {pd_desc : pdecl_desc; pos : Lexing.position*Lexing.position}

type pfichier = pdecl list
```

Analyse de la syntaxe : Ast

```
type pexpr = {  pe_desc: pexpr_desc; pe_pos : Lexing.position * Lexing.position }

and pexpr_desc =
   PEint of int
 | PEbool of bool
 | PEident of ident
 | PEbinop of bop*pexpr*pexpr
 | PEunop of uop*pexpr
 | PEselect of pexpr*ident
 | PElen of pexpr
 | PEtab of pexpr*pexpr
 | PEcall of ident*( pexpr list)
 | PEvec of (pexpr list)
 | PEprint of string
 | PEbloc of pblock
 | PEexpr of pexpr
```

Analyse de la syntaxe : Ast

```
and pinstr = {pi_desc : pinstr_desc; pi_pos : Lexing.position * Lexing.position}

and pinstr_desc =
   PInone
 | PIexpr of pexpr
 | PIinit of mutident*pexpr
 | PIinitStruct of mutident*ident*((ident*pexpr) list)
 | PIwhile of pexpr*pblock
 | PIend
 | PIreturn of pexpr
 | PIif of pif
```

```
 and pblock =
   B of pinstr*pblock
  |I of pinstr
  |E of pexpr
  |EmptyBloc
```

PARSER

Analyse de la syntaxe : Parser

```
decl_struct:
 STRUCT i = ident LEFTG l = separated_list(COMMA,decl_sous_struct) RIGHTG
 {{nom_pstruct = i; val_pstruct = l}}
;

decl_sous_struct:
 x = ident TO t = tip {(x,t)}
;

decl_fun:
 |FN i = ident LEFTPAR la = separated_list(COMMA,argument) RIGHTPAR b = bloc
 {{nom_pfun =i;arg_pfun = la;typ_pfun = None; bloc_pfun = b}}
 |FN i = ident LEFTPAR la = separated_list(COMMA,argument) RIGHTPAR MOINS SUP t = tip b = bloc
 {{nom_pfun =i;arg_pfun =la;typ_pfun = T t;bloc_pfun = b}}
;
```

Analyse de la syntaxe : Parser

```
bloc:
 LEFTG b = blockbody  RIGHTG {b }
 | LEFTG RIGHTG { EmptyBloc}
;

blockbody:
i = instr b = blockbody {B (i,b)}
|i = instr  {I i}
|e  = expr {E e}
;
```

```
expr_desc:
...
 |MOINS e = expr %prec umoins {PEunop (Neg,e)}
 |FOIS e = expr %prec ufois {PEunop (Star,e)}
 |u = unaire e = expr {PEunop(u,e)}
...
;
```

```
...
%left PLUS MOINS
%left FOIS DIVISE MOD
%nonassoc ufois umoins
...
```

Analyse de la syntaxe : Parser

```
bloc:
 LEFTG b = blockbody  RIGHTG {b }
 | LEFTG RIGHTG { EmptyBloc}
;

blockbody:
i = instr b = blockbody {B (i,b)}
|i = instr  {I i}
|e  = expr {E e}
;
```

```
expr_desc:
...
 |MOINS e = expr %prec umoins {PEunop (Neg,e)}
 |FOIS e = expr %prec ufois {PEunop (Star,e)}
 |u = unaire e = expr {PEunop(u,e)}
...
;
```

```
...
%left PLUS MOINS
%left FOIS DIVISE MOD
%nonassoc ufois umoins
...
```

```
ident:
 |id = IDENT {id}
 |PRINT  {"print"}
 |LEN {"len"}
;
```

TYPER

PRODUCTION DE CODE

Production de code : Allocateur

```
Smap.t -> int -> Texpr -> (Cexpr*int)

let rec alloc_expr env next = function
 |TEint i -> ...

 |TEbool b -> ...

 |TEident (x,size) ->
  let ofs_x = (Smap.find x env) in
   Cident (ofs_x,size),next
 |TEbinop(o,e1,e2) ->
   let e1,fpmax1 = alloc_expr env next e1 in
   let e2,fpmax2 = alloc_expr env next e2 in
    Cbinop(o,e1,e2), max fpmax1 fpmax2

 |TEunop(u,e1) -> ...

 |TEderef(u,e,i) -> ...
```

```
|TEselect(e,pos,size,totalsize) -> ...

|TElen e -> ...

|TEtab (e1,e2,size) -> ...

|TEcall (x,l) ->
  let l, fpmax =
  List.fold_left  (fun (l, fpmax) e ->
  let e, fpmax' = alloc_expr env next e in
    e::l, max fpmax fpmax')
    ([], next) l in
     Ccall (x,List.rev l), fpmax

|TEvec (l,size) -> ...

|TEprint s -> ...

|TEbloc b -> ...

|TEexpr e -> ...
```

Production de code : Allocateur

```
Smap.t -> int -> Tinstr -> (Cexpr*int*Smap.t)

and alloc_instr env next i = match i with
 |TInone -> ...
 |TIexpr e -> ...
 |TIinit (x, e, size) ->
   let e,fpmax = alloc_expr env next e in
   let next = next + size in
     CIinit (-next, e),fpmax,(Smap.add x (-next) env)

 |TIinitStruct (x, structenv,size) ->
   let newenv, fpmax =
   Smap.fold (...)  in
   let next = next + size  in
     CIinitStruct ( -next, newenv,size),fpmax,(Smap.add x (-next) env)

 |TIwhile (e,b) -> ...

 |TIend -> ...

 |TIreturn e -> ...

 |TIif i -> ...
```

Production de code : Allocateur

```
Smap.t -> int -> Tbloc -> (Cexpr*int)

and alloc_bloc env next b = match b with
 |TB (i,b) ->
   let i,fpmaxi,newenv = alloc_instr env next i in
   let b, fpmaxb = alloc_bloc newenv fpmaxi b in
     CB (i,b), fpmaxb

 |TI i ->
   let i,fpmax,newenv = alloc_instr env next i in
     CI i,fpmax

 |TE e ->
   let e,fpmax,_ = alloc_expr env next e in
     CE e,fpmax

 |TEmptyBloc ->
     CEmptyBloc, next
```

Production de code : Allocateur

Tfun -> (Cfun*(int*int))

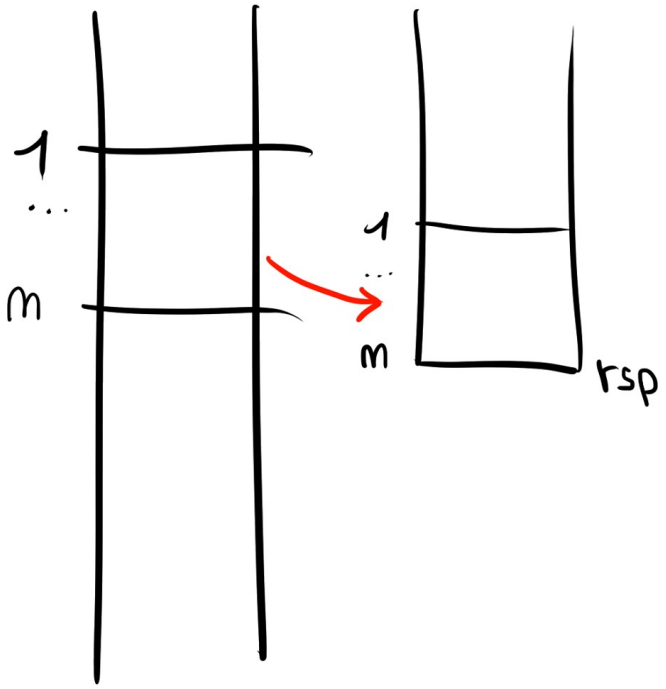
```
let alloc_fun f =
 let env, next, l = List.fold_left
              (fun (env, next, largs) x -> let next = next + x.size_targ in
              (Smap.add x.nom_targ (next-x.size_targ+8) env, next, (next,x.size_targ)::largs))
              (Smap.empty, 8, []) (f.arg_tfun) in
 let b, fpmax = alloc_bloc env 0 f.bloc_tfun in begin
   Hashtbl.add ret_fun f.nom_tfun f.size_tfun;
   ({nom_cfun = f.nom_tfun; arg_cfun = List.rev l; bloc_cfun = b},
    (fpmax,next+f.size_tfun));
  end
```

Tfun -> (Cfun*(int*int)) list

```
let alloc fichier = List.map (alloc_fun) fichier
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Cint i -> let j = ((i + 2147483648) mod 4294967296) - 2147483648 in
     pushq (imm j),8

 |Cbool b ->
   (if b then pushq (imm 1) else pushq (imm 0)),8

 |Cident (fp_x,size) ->
   let p = (size/8 - 1) and forcode = ref nop in begin
   for i = 0 to p
   do
     forcode := pushq (ind ~ofs:(fp_x + i*8) rbp) ++ !forcode
   done;
   !forcode,size ;
   end
```



Production de code : Compiler

Cexpr -> int -> int -> code*int

```
let rec compile_expr expr fpmax stack =

  |Cbinop (o,e1,e2,v) ->
   let code1,p1 = compile_expr e1 fpmax stack in
   let code2,p2 = compile_expr e2 fpmax (stack + p1) in
   (match o with
    |(Inf|Infeg|Sup|Supeg) ->
     (let abr = (match o with
           |Inf -> jl
           |Infeg -> jle
           |Sup -> jg
           |Supeg -> jge
           |_ -> assert false) in
     code1 ++
     code2 ++
     popq rdx ++
     popq rax ++
     cmpq (reg rdx) (reg rax) ++
     abr ("l1_"^label_string) ++
     pushq (imm 0) ++
     jmp ("l0_"^label_string) ++
     label ("l1_"^label_string) ++
     pushq (imm 1) ++
     label ("l0_"^label_string)),8
```

Rq : *On aurait pu utiliser*
Cmpq (reg rdx) (reg rax)
++ setl (reg rax)

(mais je m'en suis rendu compte trop tard)

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cbinop (o,e1,e2,v) ->
   let code1,p1 = compile_expr e1 fpmax stack in
   let code2,p2 = compile_expr e2 fpmax (stack + p1) in
   (match o with
    |(Equiv|Diff) ->
     (let abr = (match o with
           |Equiv -> jz
           |Diff -> jnz
             |_-> print_string "erreur3";assert false) in
     code1 ++
     code2 ++
     popq rdx ++
     popq rax ++
     subq (reg rdx) (reg rax) ++
     abr ("l1_"^label_string) ++
     pushq (imm 0) ++
     jmp ("l0_"^label_string) ++
     label ("l1_"^label_string) ++
     pushq (imm 1) ++
     label ("l0_"^label_string)),8
```

Production de code : Compiler

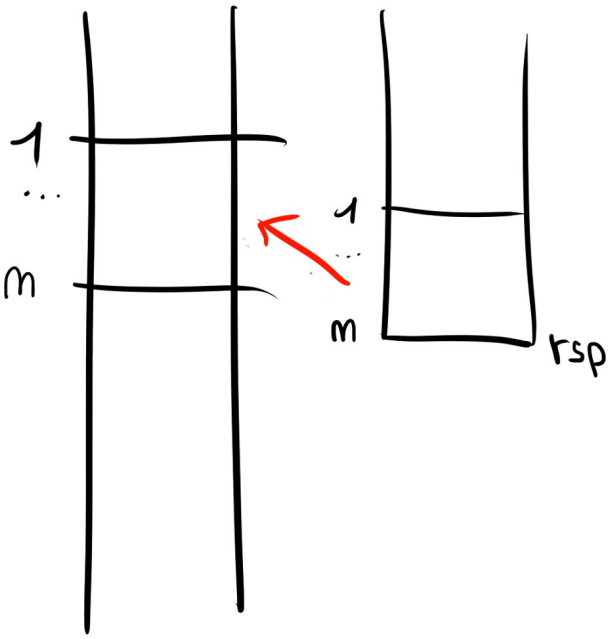
```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cbinop (o,e1,e2,v) ->
   let code1,p1 = compile_expr e1 fpmax stack in
   let code2,p2 = compile_expr e2 fpmax (stack + p1) in
   (match o with
    |(Add|Sub|Times|Div|Mod) ->
      (code1 ++
       code2 ++
       popq rcx ++
       popq rax ++
       (match o with
     |Add -> addq (reg rcx) (reg rax)
     |Sub -> subq (reg rcx) (reg rax)
     |Times -> imulq (reg rcx) (reg rax)
     |Div -> cqto ++ idivq (reg rcx)
     |Mod -> cqto ++ idivq (reg rcx)
          ++ movq (reg rdx) (reg rax)
     |_ -> print_string "erreur3";assert false)
      ++ pushq (reg rax)),8
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cbinop (o,e1,e2,v) ->
   let code1,p1 = compile_expr e1 fpmax stack in
   let code2,p2 = compile_expr e2 fpmax (stack + p1) in
   (match o with
    |And ->
     (code1 ++
     popq rax ++
     movq (imm 0) (reg rdx) ++
     testq (reg rax) (reg rax) ++
     jz ("lfin_"^label_string) ++
     code2 ++
     popq rax ++
     movq (imm 0) (reg rdx) ++
     testq (reg rax) (reg rax) ++
     jz ("lfin_"^label_string) ++
     movq (imm 1) (reg rdx) ++
     label ("lfin_"^label_string) ++
     pushq (reg rdx)),8
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cbinop (o,e1,e2,v) ->
   let code1,p1 = compile_expr e1 fpmax stack in
   let code2,p2 = compile_expr e2 fpmax (stack + p1) in
   (match o with
    | Egal ->
   (let coderef,p1 = compile_expr (Cunop(Ref,e1)) fpmax stack in
   let code2,p2 = compile_expr e2 fpmax (stack + 8) in
      let p = (p2/8 - 1) and forcode = ref nop in begin
      for i = 0 to p
      do
     forcode := !forcode ++ popq rdx ++ movq (reg rdx) (ind ~ofs:(i*8) rax)
      done;
      code2 ++
      coderef ++
      popq rax ++
      !forcode end),0)
```



Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cunop (u,e) ->
  (match u with
   |Neg -> fst(compile_expr e fpmax stack) ++
         popq rax ++
         negq (reg rax) ++
         pushq (reg rax),8

   |Not ->
    fst(compile_expr e fpmax stack) ++
    popq rax ++
    subq (reg rax) (imm 1) ++
    pushq (reg rdx)
    ,8
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cunop (u,e) ->
  (match u with
   |Ref |MutRef ->
    (match e with
     |Cderef(u,e1,size) -> compile_expr e1 fpmax stack
     |Cident (fp_x,size) -> (movq (reg rbp) (reg rax) ++
                    addq (imm fp_x) (reg rax) ++
                    pushq (reg rax)),8
     |Cselect (e1,pos,size,totalsize) ->
                    fst(compile_expr (Cunop(Ref,e1)) fpmax stack) ++
                    popq rax ++
                     addq (imm pos) (reg rax) ++
                     pushq (reg rax),8
```

Production de code : Compiler

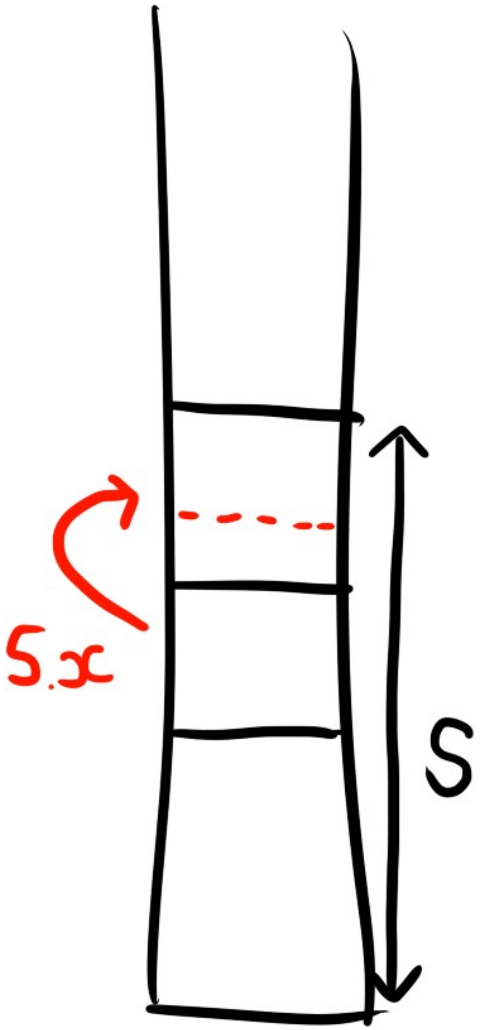
```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cunop (u,e) ->
  (match u with
   |Ref |MutRef ->
    (match e with
      |Ctab (e1,e2,size) -> (fst(compile_expr e1 fpmax stack) ++
                  fst(compile_expr e2 fpmax (stack+8)) ++
                  popq r13 ++
                  popq r12 ++
                  cmpq (reg r13) (reg r12) ++
                  jg ("erreur_oob"^label_string) ++
                  fst(compile_expr (Cbinop(Div,Cint(1),Cint(0),false)) fpmax stack) ++
                  popq rax++
                  label ("erreur_oob"^label_string) ++
                  popq rax ++
                       imulq (imm (size)) (reg r13) ++
                  addq (reg r13) (reg rax) ++
                  pushq (reg rax)),8
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

  |Cderef (u,e,size) ->
   let p = (size/8 - 1) and forcode = ref nop in begin
   for i = 0 to p
   do
     forcode :=  pushq (ind ~ofs:(i*8)  rax) ++ !forcode
   done;
   fst(compile_expr e fpmax stack) ++
   popq rax ++
   !forcode,size ;
   end
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Cselect (e,pos,size,totalsize) ->
  let code,_ = compile_expr e fpmax stack in
  let movselect = ref nop and p = (size/8-1) in begin
  for i = 0 to p do
      movselect := !movselect ++
              movq (ind ~ofs:(pos+i*8) rsp) (reg rax) ++
              movq (reg rax) (ind ~ofs:(totalsize-size+i*8) rsp)
  done;
  code ++
  !movselect ++
  popn (totalsize-size), size;
  end
```



Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Clen e ->
  (fst(compile_expr e fpmax stack) ++
   popq rdx ++
   popq rax ++
   pushq (reg rdx)),8
```

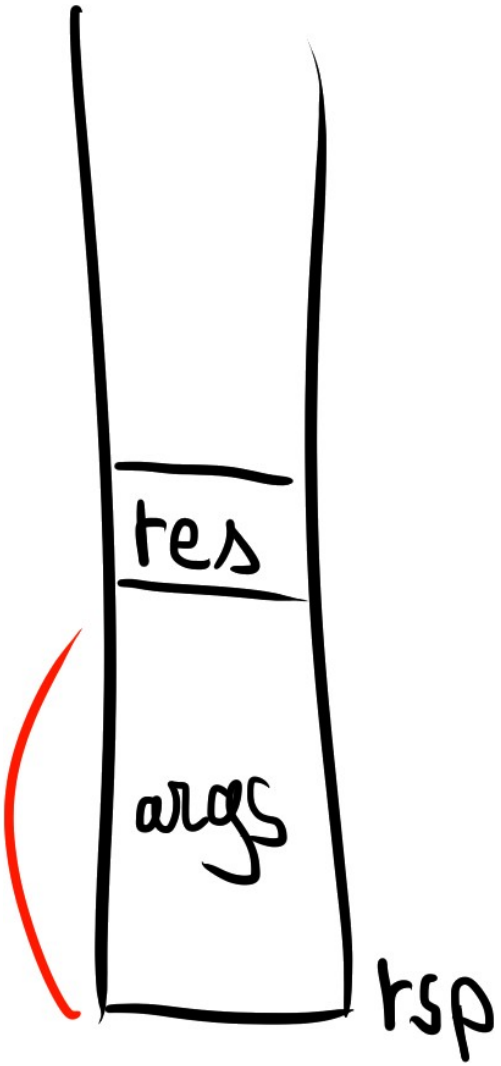
Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Ctab (e1,e2,size) ->
  let p = (size/8-1) and forcode = ref nop in begin
  for i = 0 to p
  do
     forcode := pushq (ind ~ofs:(i*8)  rax) ++ !forcode
  done;
  (   fst(compile_expr e1 fpmax (stack)) ++
      fst(compile_expr e2 fpmax (stack+16)) ++
   popq r13 ++ (* # de l'élement *)
   popq r12 ++ (*taille*)
   cmpq (reg r13) (reg r12) ++
   jg ("erreur_oob"^label_string) ++
   fst(compile_expr (Cbinop(Div,Cint(1),Cint(0),false)) fpmax stack) ++
   popq rax ++
   label ("erreur_oob"^label_string) ++
   popq rax ++ (*adresse des elements *)

   imulq (imm (size)) (reg r13) ++
   addq (reg r13) (reg rax) ++
   !forcode),size;
   end
```

Production de code

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Ccall (f,l) ->
   let (code_arg, size_arg) = List.fold_left (fun (code,size) e ->
     let (code_e,size_e) = compile_expr e fpmax (stack+size) in
     (code_e ++ code,size+size_e)) (nop,0) l
   and size_f = Hashtbl.find ret_fun f in
   pushn size_f ++
   code_arg ++
   call f ++
   popn size_arg, size_f
```



Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Cvec (l,size) ->
   let n = List.length l and compilcode = ref nop and
       l2 = ref l and initvect = ref nop in
     let p = (size/8)*n - 1 in begin
   for i = 0 to (n-1) do
     let x = List.hd(!l2) in begin
     l2 := List.tl(!l2);
     compilcode := fst(compile_expr x fpmax (stack+size*i)) ++ !compilcode
     end
   done;
   for i = 0 to p do
        initvect := popq rdx ++
              movq (reg rdx) (ind ~ofs:((p-i)*8) rax) ++ !initvect
   done;
   !compilcode ++
   movq (imm (size*n)) (reg rdi) ++
   call "malloc" ++
   !initvect ++
   pushq (reg rax) ++
   pushq (imm n),16
   end
```

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Cprint s ->
  (begin
   string_count := !string_count + 1;
   segment_donnes := !segment_donnes ++
                     label ("chaine_"^string_of_int(!string_count)) ++
                     string (s);
   movq (ilab ("chaine_"^string_of_int(!string_count))) (reg rdi) ++
   movq (imm 0) (reg rax) ++
   call "printf"
               end),0
```

Rq : *pour améliorer ce système, On pourrait vérifier que la Chaîne n'existe pas déjà avant De la rajouter, avec Smap.find.*

Production de code : Compiler

```
Cexpr -> int -> int -> code*int

let rec compile_expr expr fpmax stack =

 |Cbloc b ->
   compile_bloc b fpmax stack
```

Production de code : Compiler

```
Cbloc -> int -> int -> code*int

and compile_bloc bloc fpmax stack = match bloc with
 |CEmptyBloc ->
   nop,0
 |CE e ->
   compile_expr e fpmax stack
 |CI i ->
   compile_instr i fpmax stack
 |CB (i,b) ->
  let (c,size) = compile_bloc b fpmax stack in
  (fst(compile_instr i fpmax stack) ++ c),size
```

Production de code : Compiler

```
Cinstr -> int -> int -> code*int

and compile_instr instr fpmax stack =
      match instr with
 |CInone ->
   nop,0

 |CIexpr e ->
   let (code,size) = compile_expr e fpmax stack in
   code ++ popn size ,0
```

Production de code : Compiler

Cinstr -> int -> int -> code*int

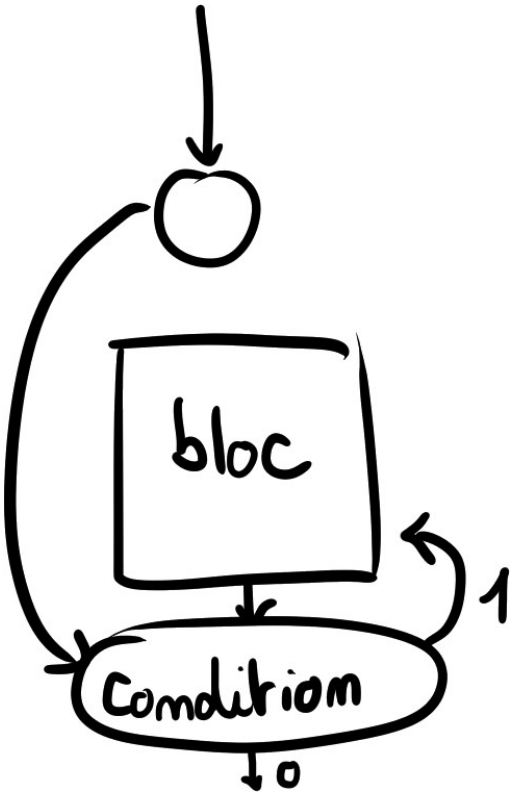
```
and compile_instr instr fpmax stack =
     match instr with
 |CIinit (fp_x,e) ->
   let (code,size) = compile_expr e fpmax stack in
   (let forcode = ref nop and p = (size/8 - 1) in begin
   for i = 0 to p do
      let ofs = fp_x + (i)*8 in
       forcode := !forcode ++
             popq rax ++
             movq (reg rax) (ind ~ofs(*:(fp_x+8*(p-i))*) rbp)
   done;
   code ++ !forcode;
   end),0
```

Production de code : Compiler

```
Cinstr -> int -> int -> code*int

and compile_instr instr fpmax stack =
     match instr with
 |CIinitStruct (fp_x,structenv,size) ->
   (Smap.fold (fun x (e,x_pos) precode ->
          let (code,x_size) = compile_expr e fpmax stack in
          let forcodebis = ref nop
          and pbis = (x_size/8-1) in begin
          for i = 0 to pbis do
            let pos_act = fp_x + x_pos + (i)*8 in
            forcodebis := !forcodebis ++
                    popq rax ++
                    movq (reg rax) (ind ~ofs:(pos_act) rbp)

          done;
          precode ++ code ++ !forcodebis;
          end) structenv nop),0
```

Production de code : Compiler

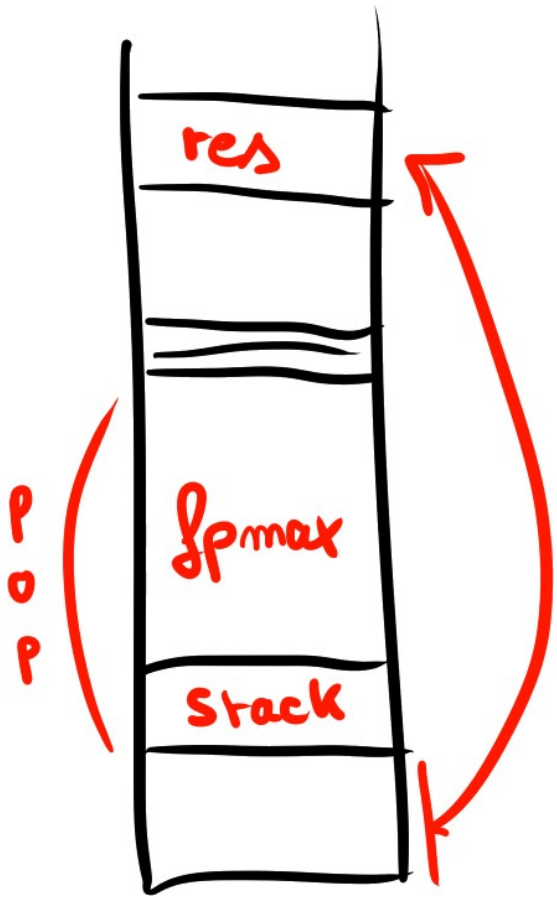
```
Cinstr -> int -> int -> code*int

and compile_instr instr fpmax stack =
      match instr with
 |CIwhile (e,b) ->
   let (c,size) = compile_bloc b fpmax stack in
   (jmp ("w_cond_"^label_string) ++
    label ("w_bloc_"^label_string) ++
    c ++
    label ("w_cond_"^label_string) ++
    fst(compile_expr e fpmax stack ) ++
    popq rax ++
    testq (reg rax) (reg rax) ++
    jnz ("w_bloc_"^label_string)),0
```



Production de code

Production de code : Compiler

```
Cinstr -> int -> int -> code*int

and compile_instr instr fpmax stack =
     match instr with
 |CIend ->
     popn stack ++
     popn (fst(fpmax)) ++
     popq rbp ++
     movq (imm 0) (reg rax) ++
     ret,0

 |CIreturn e ->
    let (c,size) = compile_expr e fpmax stack in
    let retcode = ref nop
    and pos = snd(fpmax) in begin
    for i = 0 to (size/8-1) do
      retcode := !retcode ++
                 popq rax ++
                 movq (reg rax) (ind ~ofs:(pos + i*8) rbp)
    done;
    c ++
    !retcode ++
    popn (fst(fpmax)) ++
    popq rbp ++
    ret,0;
    end
```



Production de code : Compiler

```
Cinstr -> int -> int -> code*int

and compile_instr instr fpmax stack =
     match instr with
 |CIif i ->
    compile_if i fpmax stack
```

Production de code : Compiler

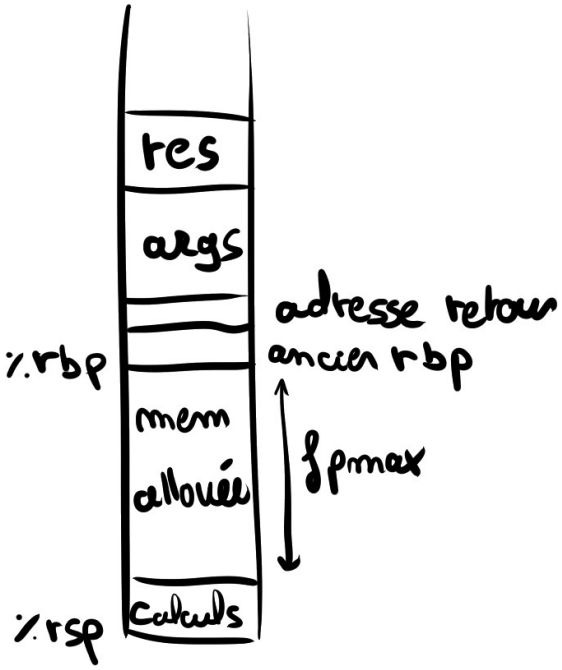
```
Cif -> int -> int -> code*int

and compile_if i fpmax stack =
     match i with
 |CifThen (e,b) ->
   let (c,size) = compile_bloc b fpmax stack in
   (fst(compile_expr e fpmax stack ) ++
    popq rax ++
    testq (reg rax) (reg rax) ++
    jz ("if_end_"^label_string) ++
    c ++
    label ("if_end_"^label_string)),0

 |CifElse (e,b1,b2) ->...

 |CifElseIf (e,b1,i2) -> ...
```

Production de code : Compiler

```
code*code -> Cfun*int -> code*code

let compile_fun (codemain, codefuns) (df,fpmax) =
   match df.nom_cfun with
    | "main" ->
     (codemain ++
           pushq (reg rbp) ++
           movq (reg rsp) (reg rbp) ++
            pushn (fst(fpmax)) ++
           fst(compile_bloc df.bloc_cfun fpmax 0)++ popn (fst (fpmax)),
         codefuns)
```

Production de code : Compiler

```
code*code -> Cfun*int -> code*code

let compile_fun (codemain, codefuns) (df,fpmax) =
   match df.nom_cfun with
    |_ ->
     (codemain,
          let codebloc,size = compile_bloc df.bloc_cfun fpmax 0
     and size_ret = Hashtbl.find ret_fun df.nom_cfun
     and retcode = ref nop
     and pos = snd(fpmax) in begin
      for i = 0 to (size_ret/8 - 1) do
         retcode :=
               popq rax ++
               movq (reg rax) (ind ~ofs:( pos - i*8) rbp) ++ !retcode
      done;
     codefuns ++
      label df.nom_cfun ++
      pushq (reg rbp) ++
      movq (reg rsp) (reg rbp) ++
      pushn (fst(fpmax)) ++
      codebloc ++
      !retcode ++
      popn (fst(fpmax)) ++
      popq rbp ++
      ret
     end)
```



Production de code : Compiler

Cfichier -> program x86_64

```
let compile_program f  =
     let f = alloc f in
     let (codemain, codefuns) = List.fold_left compile_fun  (nop, nop) f
      in

          { text = glabel "main" ++ codemain ++
          movq (imm 0) (reg rax) ++
          ret ++
          codefuns;
          data =     !segment_donnes  }
```

Merci de votre attention