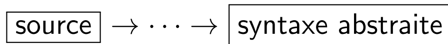
École Normale Supérieure

Langages de programmation et compilation

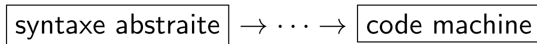
Jean-Christophe Filliâtre

Cours 8 / 23 novembre 2015

on a terminé la phase d'**analyse**



on considère maintenant la phase de **synthèse**



stratégie d'évaluation et passage des paramètres

dans la **déclaration** d'une fonction

```
function f(x1, ..., xn) =
  ...
```

les variables `x1,...,xn` sont appelées **paramètres formels** de `f`

et dans l'**appel** de cette fonction

```
f(e1, ..., en)
```

les expressions `e1,...,en` sont appelées **paramètres effectifs** de `f`

dans un langage comprenant des modifications en place, une affectation

```
e1 := e2
```

modifie un emplacement mémoire désigné par l'expression `e1`

l'expression `e1` est limitée à certaines constructions,
car des affectations comme

```
42 := 17
true := false
```

n'ont en général pas de sens

on parle de **valeur gauche** pour désigner les expressions légales à gauche d'une affectation

la stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués

on peut la définir à l'aide d'une sémantique formelle (cf cours 3)

le compilateur se doit de respecter la stratégie d'évaluation

en particulier, la stratégie d'évaluation **peut** spécifier

- à quel moment les paramètres effectifs d'un appel sont évalués
- l'ordre d'évaluation des opérandes et des paramètres effectifs

certains aspects de l'évaluation peuvent cependant rester **non spécifiés**

cela laisse alors de la latitude au compilateur, notamment pour effectuer des optimisations (par exemple en ordonnant les calculs comme il le souhaite)

on distingue notamment

- l'**évaluation stricte** : les opérandes / paramètres effectifs sont évalués avant l'opération / l'appel

  exemples : C, C++, Java, OCaml, Python, mini-ML du cours 3

- l'**évaluation paresseuse** : les opérandes / paramètres effectifs ne sont évalués que si nécessaire

  exemples : Haskell, Clojure
  mais aussi les connectives logiques `&&` et `||` de la plupart des langages

un langage impératif adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source

par exemple, le programme OCaml

```
let r = ref 0
let id x = r := !r + x; x
let f x y = !r
let () = print_int (f (id 40) (id 2))
```

affiche 42 car les deux arguments de `f` ont été évalués

la non-terminaison est également un effet

ainsi, le programme

```
let rec loop () = loop ()
let f x y = x + 1
let v = f 41 (loop ())
```

ne termine pas, bien que l'argument `y` n'est pas utilisé

un langage purement applicatif, en revanche, peut adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur (on parle de **transparence référentielle**)

en particulier, il peut faire le choix d'une évaluation paresseuse

le programme Haskell

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

termine (après avoir affiché a)

la sémantique précise également le **mode de passage** des paramètres lors d'un appel

on distingue notamment

- l'**appel par valeur** (*call by value*)
- l'**appel par référence** (*call by reference*)
- l'**appel par nom** (*call by name*)
- l'**appel par nécessité** (*call by need*)

(on parle aussi parfois de **passage** par valeur, par référence, etc.)

de **nouvelles** variables représentant les paramètres formels reçoivent les **valeurs** des paramètres effectifs

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // affiche 41
```

les paramètres formels désignent les **mêmes** valeurs gauches que les paramètres effectifs

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // affiche 42
```

les paramètres effectifs sont **substitués** aux paramètres effectifs, textuellement, et donc évalués seulement si nécessaire

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // affiche 25
  // 1+2 est évalué deux fois
  // 2+2 est évalué deux fois
  // 1/0 n'est jamais évalué
```

les paramètres effectifs ne sont évalués que si nécessaire,
mais **au plus une fois**

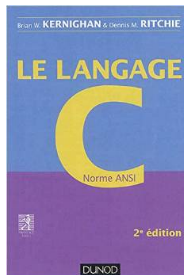
```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // affiche 25
  // 1+2 est évalué une fois
  // 2+2 est évalué une fois
  // 1/0 n'est jamais évalué
```

quelques mots sur le langage C

le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite

on peut le considérer inversement comme un assembleur de haut niveau

un ouvrage toujours d'actualité :
Le langage C
de Brian Kernighan et Dennis Ritchie



le langage C est muni d'une stratégie d'évaluation stricte,
avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

- on trouve des types de base tels que `char`, `int`, `float`, etc. (mais pas de booléens)

- un type τ`*` des pointeurs vers des valeurs de type τ

  si *p* est un pointeur de type τ`*`, alors `*`*p* désigne la valeur pointée par *p*, de type τ

  si *e* est une valeur gauche de type τ, alors `&`*e* est un pointeur sur l'emplacement mémoire correspondant, de type τ`*`

- des enregistrements, appelés *structures*, tels que

```
struct L { int head; struct L *next; };
```

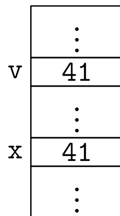
si e a le type `struct L`, on note `e.head` l'accès au champ

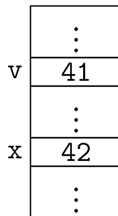
en C, une valeur gauche est de la forme

- `x`, une variable
- `*e`, le déréférencement d'un pointeur
- `e.x`, l'accès à un champ de structure,
  si e est elle-même une valeur gauche

- `t[e]`, qui n'est autre que `*(t+e)`
- `e->x`, qui n'est autre que `(*e).x`

```c
void f(int x) {
  x = x+1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```

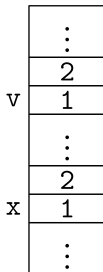


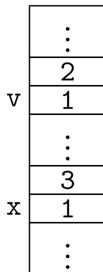


l'appel par valeur implique que les structures sont **copiées** lorsqu'elles sont passées en paramètres ou renvoyées

les structures sont également copiées lors des affectations de structures, *i.e.* des affectations de la forme `x = y`, où `x` et `y` ont le type `struct S`

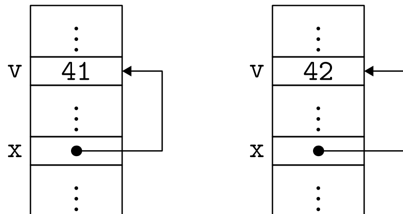
```
struct S { int a; int b; };

void f(struct S x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut toujours 2
}
```





on peut **simuler** un passage par référence en passant un pointeur explicite

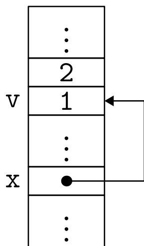
```
void incr(int *x) {
  *x = *x + 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v vaut maintenant 42
}
```

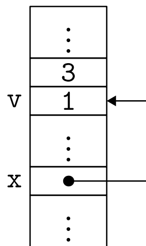


mais ce n'est qu'un passage de pointeur **par valeur**

pour éviter le coût des copies, on passe des pointeurs sur les structures le plus souvent

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b = x->b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b vaut maintenant 3
}
```





la manipulation explicite de pointeurs peut être dangereuse

considérons le programme

```
int* p() {
  int x;
  return &x;
}
```

il renvoie un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de `p`), et qui sera très probablement réutilisé rapidement par un autre tableau d'activation

on parle de référence fantôme (*dangling reference*)

on peut déclarer un tableau ainsi :

```
int t[10];
```

la notation `t[i]` n'est que du sucre syntaxique pour `*(t+i)` où

- `t` désigne un pointeur sur le début d'une zone contenant 10 entiers
- `+` désigne une opération d'*arithmétique de pointeur* (qui consiste à ajouter à `t` la quantité `4i` pour un tableau d'entiers 32 bits)

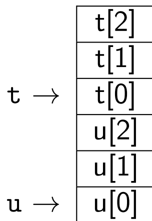
le premier élément du tableau est donc `t[0]` c'est-à-dire `*t`

quand on passe un tableau en paramètre, on ne fait que passer le pointeur (par valeur, toujours)

on ne peut affecter des tableaux, seulement des pointeurs

ainsi, on ne peut pas écrire

```
void p() {
  int t[3];
  int u[3];
  t = u;
}
```



car `t` et `u` sont des tableaux (alloués sur la pile) et l'affectation de tableaux n'est pas autorisée

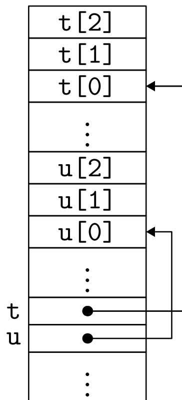
en revanche on peut écrire

```
void q(int t[3], int u[3]) {
  t = u;
}
```

car c'est exactement la même chose que

```
void q(int *t, int *u) {
  t = u;
}
```

et l'affectation de pointeurs est autorisée



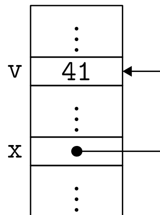
quelques mots sur le langage C++

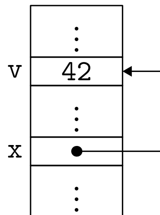
en C++, on trouve (entre autres) les types et constructions du C,
avec une stratégie d'évaluation stricte

le mode de passage est **par valeur** par défaut

mais on trouve aussi un passage **par référence**
indiqué par le symbole & au niveau de l'argument formel

```
void f(int &x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut maintenant 42
}
```



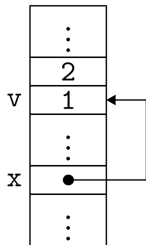


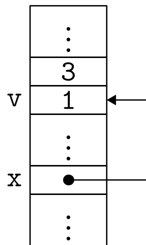
en particulier, c'est le compilateur qui

- a pris l'adresse de `v` au moment de l'appel
- a déréférencé l'adresse `x` dans la fonction `f`

on peut passer une structure par référence

```
struct S { int a; int b; };

void f(struct S &x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut maintenant 3
}
```





on peut passer un pointeur par référence

par exemple pour ajouter un élément dans un arbre

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if      (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

quelques mots sur le langage OCaml

OCaml est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, etc.) alloué sur le tas en général

les valeurs gauches sont les éléments de tableaux

```
a.(2) <- true
```

et les champs mutables d'enregistrements

```
x.age <- 42
```

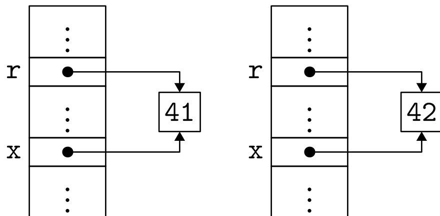
rappel : une référence est un enregistrement

```
type 'a ref = { mutable contents: 'a }
```

et les opérations := et ! sont définies par

```
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

```
let f x =
  x := !x + 1

let main () =
  let r = ref 41 in
  f r
  (* !r vaut maintenant 42 *)
```



c'est toujours un passage par valeur,
d'une valeur qui est un pointeur (vers une valeur mutable)

on peut **simuler l'appel par nom** en OCaml, en remplaçant les arguments par des fonctions

ainsi, la fonction

```
let f x y =
  if x = 0 then 42 else y + 1
```

peut être réécrite en

```
let f x y =
  if x () = 0 then 42 else y () + 1
```

et appelée comme ceci

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

plus subtilement, on peut aussi **simuler l'appel par nécessité** en OCaml

on commence par introduire un type pour représenter les calculs paresseux

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

et une fonction qui évalue un tel calcul si ce n'est pas déjà fait

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

on définit alors la fonction f comme ceci

```
let f x y =
  if force x = 0 then 42 else force y + 1
```

et on l'utilise ainsi

```
let v = f (ref (Frozen (fun () -> 0)))
          (ref (Frozen (fun () -> failwith "oups")))
```

note : la construction lazy d'OCaml fait quelque chose de semblable (un peu plus subtilement)

compilation d'un fragment de Pascal

parce qu'on y trouve à la fois

- des fonctions imbriquées
- du passage par valeur et par référence

on considère le fragment suivant de Pascal

$$
\begin{array}{rcl}
E & \rightarrow & n \\
  & | & x \\
  & | & E \texttt{ + } E \mid E \texttt{ - } E \\
  & | & E \texttt{ * } E \mid E \texttt{ / } E \\
  & | & \texttt{- } E
\end{array}
\qquad
\begin{array}{rcl}
C & \rightarrow & E \texttt{ = } E \mid E \texttt{ <> } E \\
  & | & E \texttt{ < } E \mid E \texttt{ <= } E \mid E \texttt{ > } E \mid E \texttt{ >= } E \\
  & | & C \texttt{ and } C \\
  & | & C \texttt{ or } C \\
  & | & \texttt{not } C
\end{array}
$$

$$
\begin{array}{rcl}
S & \rightarrow & x \texttt{:= } E \\
  & | & \texttt{if } C \texttt{ then } S \\
  & | & \texttt{if } C \texttt{ then } S \texttt{ else } S \\
  & | & \texttt{while } C \texttt{ do } S \\
  & | & p(E, \dots, E) \\
  & | & B
\end{array}
\qquad
\begin{array}{rcl}
D & \rightarrow & \texttt{var } x, \dots, x\texttt{: integer;} \\
  & | & \texttt{procedure } p(F; \dots; F)\texttt{;} \\
  & & \quad D \dots D\ B\texttt{;} \\
F & \rightarrow & x\texttt{: integer} \\
  & | & \texttt{var } x\texttt{: integer}
\end{array}
$$

$$
B \rightarrow \texttt{begin } S\texttt{;} \dots \texttt{;} S \texttt{ end}
\qquad
P \rightarrow \texttt{program } x\texttt{; } D \dots D\ B\texttt{.}
$$

```
program fact;

var f : integer;

procedure fact(n : integer);
begin
  f := 1;
  while n > 1 do begin
    f := n * f;
    n := n - 1
  end
end;

begin
  fact(10);
  writeln(f)  { affiche 3628800 }
end.
```

```
program fib;
var f : integer;

procedure fib(n : integer);
   procedure somme();
   var tmp : integer;
   begin fib(n-2); tmp := f;
         fib(n-1); f := f + tmp end;
begin
   if n <= 1 then f := n else somme()
end;

begin fib(10); writeln(f) end.  { affiche 55 }
```

exemple 3

```
program syracuse;

procedure syracuse(max : integer);
var i : integer;
   procedure length();
   var v,j : integer;
      procedure step();
      begin
        v := v+1; if j = 2*(j/2) then j := j/2 else j := 3*j+1
      end;
   begin
     v := 0; j := i; while j <> 1 do step(); writeln(v)
   end;
begin
   i := 1;
   while i <= max do begin length(); i := i+1 end
end;

begin syracuse(100) end.  { affiche 0 1 7 2 5 ... }
```

les procédures pouvant être imbriquées, on introduit quelques définitions

Définition (niveau)

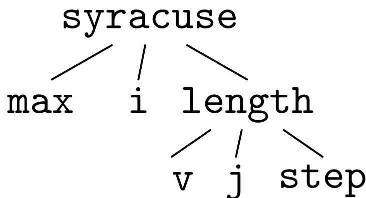
Le ***niveau*** *d'une déclaration (de variable ou de procédure) est le nombre de procédures sous lesquelles elle est déclarée. Le programme principal a le niveau 0.*

Définition (père, ancêtre, frères)

On dit que p est le ***père*** *d'un identificateur y si y est déclaré dans p. On dit que p est un* ***ancêtre*** *de y si p est soit y soit le père d'un ancêtre de y. On dit que deux identificateurs sont* ***frères*** *s'ils ont le même père.*

exemple

```
program syracuse;

procedure syracuse(max : integer);
var i : integer;
   procedure length();
   var v,j : integer;
      procedure step();
      begin
        ...
      end;
   begin
     ...
   end;
begin
   ...
end;

begin syracuse(100) end.
```



la **portée** est usuelle : si le corps d'une procédure p mentionne un identificateur alors celui-ci est soit une déclaration locale de p, soit un ancêtre de p (y compris p lui-même), soit le frère d'un ancêtre de p

l'analyse de portée est réalisée avant ou pendant le typage

la syntaxe abstraite conserve une trace de cette analyse ; pour chaque identificateur du programme, on conserve notamment son niveau de déclaration

arbres de syntaxe abstraite issus de l'analyse syntaxique :

```
type pint_expr =
  | PEconst of int
  | PEvar   of string
  | PEbinop of binop * pint_expr * pint_expr
  ...
```

(les identificateurs sont des **chaînes** de caractères)

arbres de syntaxe abstraite issus du typage :

```
type ident = { ident : string; level : int; ... }

type int_expr =
  | Econst of int
  | Evar   of ident
  | Ebinop of binop * int_expr * int_expr
```

(les identificateurs sont maintenant du type `ident`)

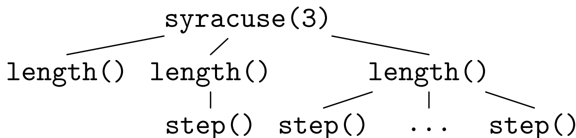
portée (exemple)

```
program syracuse;

procedure syracuse0(max1 : integer);
var i1 : integer;
   procedure length1();
   var v2,j2 : integer;
      procedure step2();
      begin
        v2 := v2+1; if j2 = 2*(j2/2) then j2 := j2/2 else j2 := 3*j2+1
      end;
   begin
     v2 := 0; j2 := i1; while j2 <> 1 do step2(); writeln(v2)
   end;
begin
   i1 := 1;
   while i1 <= max1 do begin length1(); i1 := i1+1 end
end;

begin syracuse0(100) end.
```

Définition (arbre d'activation)

*Pour une exécution donnée d'un programme, on définit l'***arbre d'activation** *de la manière suivante : tout nœud correspond à un appel de procédure $p(e_1, \dots, e_n)$ et les sous-nœuds correspondent aux appels directement effectués par $p(e_1, \dots, e_n)$.*

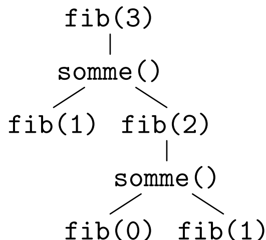
exemple :



attention : la profondeur dans l'arbre d'activation ne coïncide pas nécessairement avec le niveau de déclaration

exemple :

```
program fib;
var f : integer;

procedure fib(n : integer);
   procedure somme();
   var tmp : integer;
   begin fib(n-2); tmp := f;
         fib(n-1); f := f + tmp end;
begin
   if n <= 1 then f := n else somme()
end;

begin fib(3); writeln(f) end.
```



Définition (procédure active)

On dit qu'une procédure est **active** *si on n'a pas encore fini d'exécuter le corps de cette procédure.*

Proposition

Lorsqu'une procédure p est active, alors tous les ancêtres de p sont des procédures actives.

preuve : par récurrence sur la profondeur de p dans l'arbre d'activation

c'est vrai pour le programme principal (n'a que lui-même comme ancêtre)

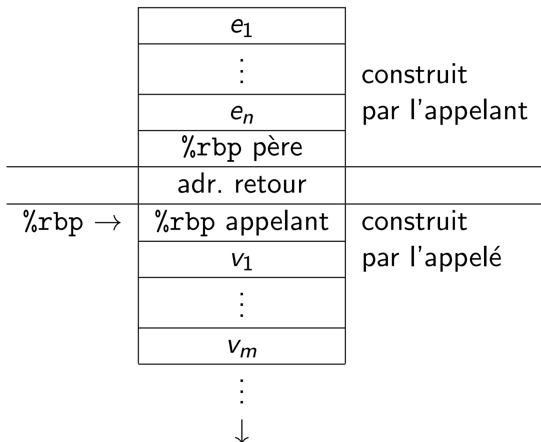
si p a été activée par une procédure q alors q est toujours active (par définition) et tous les ancêtres de q sont actifs par hypothèse de récurrence ; or les règles de visibilité impliquent que soit q est le père de p, soit p est le frère d'un ancêtre de q, et dans les deux cas tous les ancêtres de p sont bien actifs □

il faut choisir un emplacement mémoire pour chaque variable et être capable de **calculer** cet emplacement à l'exécution

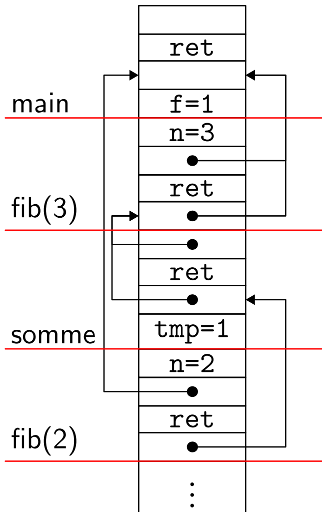
on procède ainsi : à chaque procédure active correspond une portion de la **pile** appelée **tableau d'activation**, qui contient

- ses paramètres effectifs
- ses variables locales
- un pointeur vers le tableau d'activation de sa procédure père (qui existe en vertu du résultat précédent !)
- un pointeur vers le tableau d'activation de la procédure appelante
- l'adresse de retour

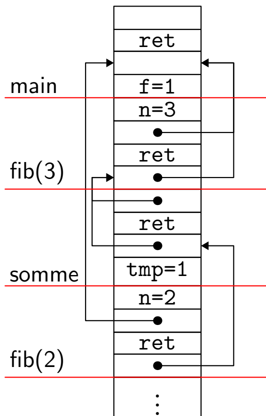
tableau d'activation correspondant à un appel $p(e_1, \dots, e_n)$ d'une procédure $p(x_1, \dots, x_n)$; `var` $v_1, \dots, v_m$; `begin` ... `end`

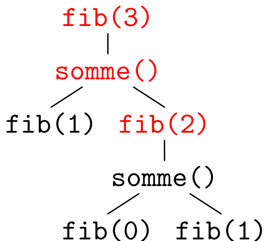


```
program fib;

var f : integer;

procedure fib(n : integer);
   procedure somme();
   var tmp : integer;
   begin
     fib(n-2); tmp := f;
     fib(n-1); f := f + tmp
   end;
begin
   if n <= 1 then f := n else somme()
end;

begin fib(3); writeln(f) end.
```



à chaque instant, les tableaux d'activation sur la pile correspondent à un chemin depuis la racine dans l'arbre d'activation





on étend le type `ident` pour indiquer la position de la variable dans le tableau d'activation :

```
type ident = { ident : string; level : int; offset : int }
```

on suppose que cette information a été calculée
(par exemple en même temps que l'analyse de portée)

allons-y !

pour produire du code X86-64, on utilise le module OCaml `X86_64` fourni sur la page du cours

avec ce module, on écrit par exemple

```
movq (imm 42) (reg rax)
```

pour construire l'instruction assembleur

```
movq $42, %rax
```

et on concatène les morceaux d'assembleur avec une opération ++

on suit un schéma de compilation simpliste, utilisant la pile pour stocker les résultats intermédiaires (on verra plus tard comment utiliser efficacement les registres)

écrivons une fonction `int_expr` qui compile une expression arithmétique

```
val int_expr : int -> int_expr -> X86_64.text
```

à l'issue de l'exécution, le résultat de l'expression doit se trouver dans `%rdi`

l'entier passé en argument est le niveau auquel se situe l'expression, soit $n+1$ si l'expression se trouve dans le corps d'une procédure de niveau n

on commence par les constantes entières

```
let rec int_expr lvl = function
  | Econst n ->
      movq (imm n) (reg rdi)
```

et les opérations arithmétiques

```
  | Ebinop (Badd, e1, e2) ->
      int_expr lvl e1 ++
      pushq (reg rdi) ++
      int_expr lvl e2 ++
      popq rsi ++
      addq (reg rsi) (reg rdi)
  | Ebinop (Bsub, e1, e2) ->
      ...
```

bien entendu, c'est extrêmement naïf ; le code pour 1+2 est

```
        movq  $1, %rdi
        pushq %rdi
        movq  $2, %rdi
        popq  %rsi
        addq  %rsi, %rdi
```

alors même que l'on dispose de 16 registres

le cas intéressant est celui d'une **variable** x

soit `l` son niveau et `ofs` sa position dans le tableau d'activation

pour trouver le tableau d'activation de x, il faut **suivre** `lvl - l` **fois** le pointeur vers le tableau d'activation du père

```
| Evar { level = l; offset = ofs } ->
    assert (l <= lvl);
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    movq (ind ~ofs rsi) (reg rdi)
```

avec

```
let rec iter n code =
  if n=0 then nop else code ++ iter (n - 1) code
```

de même, les expressions booléennes sont compilées avec une fonction

```
val bool_expr : int -> bool_expr -> X86_64.text
```

d'une manière très analogue

```
let rec bool_expr lvl = function
  | Bcmp (Beq, e1, e2) ->
      int_expr lvl e1 ++ pushq (reg rdi) ++
      int_expr lvl e2 ++ popq rsi ++
      cmpq (reg rdi) (reg rsi) ++
      sete (reg dil) ++ movzbq (reg dil) rdi
  | ...
      (* laissé en exercice *) ...
```

attention : les opérateurs `and` et `or` doivent être évalués paresseusement *i.e.* e_2 n'est pas évaluée dans e_1 `and` e_2 (resp. e_1 `or` e_2) si e_1 vaut `false` (resp. `true`)

les instructions sont compilées avec une fonction

```
val stmt : int -> stmt -> X86_64.text
```

```
let rec stmt lvl = function
  | Swriteln e ->
      int_expr lvl e ++ call "print_int"
```

avec

```
print_int:
        movq %rdi, %rsi
        movq $.Sprint_int, %rdi
        movq $0, %rax
        call printf
        ret
.data
.Sprint_int:
        .string "%d\n"
```

```
| Sif (e, s1, s2) ->
    (* laissé en exerice *)

| Swhile (e, s) ->
    (* laissé en exerice *)

| Sblock sl ->
    List.fold_left
      (fun code s -> code ++ stmt lvl s) nop sl
```

pour un appel à une procédure `p` de niveau `l`, il faut

1. empiler les arguments
2. empiler le pointeur vers le tableau d'activation du père : pour cela, il suffit de suivre `lvl - l` fois le pointeur vers le tableau du père
3. appeler le code situé à l'étiquette `p`
4. dépiler les arguments et le pointeur vers le tableau du père

```
| Scall ({pident = p; plevel = l}, el) ->
    List.fold_left
      (fun acc e -> acc ++ int_expr lvl e ++ pushq (reg rdi))
      nop el ++
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    pushq (reg rsi) ++
    call p ++ addq (imm (8 + 8 * List.length el)) (reg rsp)
```

reste l'**affectation** `x := e`

le membre gauche est ici réduit à une variable `x`

d'une manière générale, le membre gauche d'une affectation doit être une **valeur gauche**, c'est-à-dire une expression qui désigne un emplacement mémoire

ainsi 3+1 `:= e` n'aurait pas sens,
pas plus que `f(x) := e` dans un langage avec fonctions

il est important de noter que la signification de l'identificateur `x` n'est pas la même à gauche et à droite de `:=`
(c'est pourquoi on parle de valeur gauche et de valeur droite)

comme pour la valeur droite, on suit `lvl - l` fois le pointeur vers le tableau d'activation du père

```
| Sassign ({ level = l; offset = ofs }, e) ->
    int_expr lvl e ++
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    movq (reg rdi) (ind ~ofs rsi)
```

pour l'instant, on a passé les paramètres **par valeur**

i.e. le paramètre formel est une **nouvelle variable** qui prend comme valeur initiale celle du paramètre effectif

en Pascal, le qualificatif `var` permet de spécifier un passage **par référence**

dans ce cas le paramètre formel désigne la **même variable** que le paramètre effectif, qui doit donc être une variable (une valeur gauche, de manière plus générale)

```
program test;

procedure fact(n : integer; var f : integer);
begin
   f := 1;
   while n > 1 do begin
      f := n * f;
      n := n - 1
   end
end;

var x : integer;

begin
   fact(10, x);
   writeln(x)    { affiche 3628800 }
end.
```

pour prendre en compte le passage par référence, on étend encore le type `ident` pour indiquer s'il s'agit d'une variable passée par référence

```
type by_reference = bool

type ident =
  { ident : string;
    level : int;
    offset : int;
    by_reference : by_reference }
```

(note : vaut toujours `false` pour les variables locales)

dans un appel tel que `p(e)` le paramètre effectif `e` n'est plus typé ni compilé de la même manière selon qu'il s'agit d'un paramètre passé par valeur ou par référence

lorsque le paramètre est passé par référence, le typage va donc

1. vérifier qu'il s'agit bien d'une variable (valeur gauche)
2. indiquer qu'elle doit être passée par référence

une façon de procéder consiste à ajouter une construction de « calcul de valeur gauche » dans la syntaxe des expressions

```
type int_expr =
  ...
  | Eaddr of ident
```

et à remplacer, le cas échéant, le paramètre effectif `e` par `Eaddr e`

il faut ajouter le code correspondant dans `int_expr` :

```
let rec int_expr lvl = function
  | Eaddr { level = l; offset = ofs; by_reference = br } ->
      assert (l <= lvl);
      movq (reg rbp) (reg rdi) ++
      iter (lvl - l) (movq (ind ~ofs:16 rdi) (reg rdi)) ++
      addq (imm ofs) (reg rdi) ++
      if br then movq (ind rdi) (reg rdi) else nop
```

note : le cas `br = true` correspond au cas d'une variable elle-même passée par référence

il faut aussi modifier le calcul des valeurs droites :

```
  | Evar { level = l; offset = ofs; by_reference = br } ->
      assert (l <= lvl);
      movq (reg rbp) (reg rsi) ++
      iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
      movq (ind ~ofs rsi) (reg rdi) ++
      if br then movq (ind rdi) (reg rdi) else nop
```

ainsi que celui de l'affectation :

```
| Sassign ({level=l; offset=ofs; by_reference=br}, e) ->
    int_expr lvl e ++
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    (if br then movq (ind ~ofs rsi) (reg rsi)
     else addq (imm ofs) (reg rsi)) ++
    movq (reg rdi) (ind rsi)
```

en revanche, il n'y a rien à modifier dans l'appel (grâce à la nouvelle construction Eaddr)

il reste à compiler les déclarations

```
type pident = { pident : string; plevel : int }

type procedure =
  { name    : pident;
    formals : (string * by_reference) list;
    locals  : decl list;
    body    : stmt; }

and decl =
  | Var       of string list
  | Procedure of procedure
```

on compile une déclaration avec

```
val decl : decl -> X86_64.text
```

```
let rec decl = function
  | Var _       -> nop
  | Procedure p -> procedure p ++ decls p.locals

and decls dl =
  List.fold_left (fun code d -> code ++ decl d) nop dl
```

```
let frame_size dl = (* 8 fois le nombre de variables de dl *)

let procedure p =
  let fs = 8 + frame_size p.locals in
```

```
p:
       subq $fs, %rsp           # alloue le tableau
       movq %rbp, fs-8(%rsp)    # sauve %rbp
       leaq fs-8(rsp), %rbp     # positionne rbp
```

```
  ++ stmt (p.name.plevel + 1) p.body ++
```

```
       movq %rbp, %rsp          # désalloue le tableau
       popq %rbp                # restaure %rbp
       ret                      # retour à l'appelant
```

il suffit de considérer le programme comme une procédure de niveau -1

```
let prog p =
  let fs = 8 + frame_size p.locals in
```

```
  main:
       subq $fs, %rsp          # alloue le tableau
       leaq fs-8(rsp), %rbp    # positionne rbp
```

```
  ++ stmt 0 p.body ++
```

```
       addq $fs, %rsp          # désalloue le tableau
       movq $0, %rax           # code de retour 0
       ret
```

```
  ++ decls p.locals
```

démo

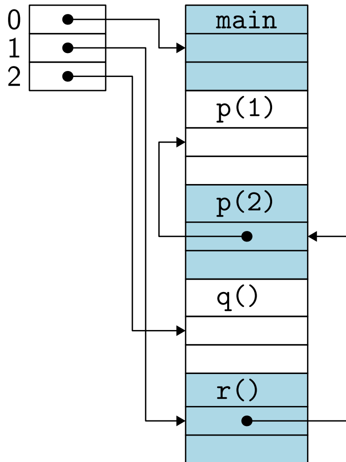
on peut améliorer l'accès aux variables

l'idée est de conserver dans une table un pointeur vers le **dernier** tableau d'activation de chaque niveau ; et les tableaux d'un même niveau forment une liste chaînée

- quand on appelle une procédure d'un niveau n
  - on la fait pointer vers la dernière procédure de niveau n
  - elle devient la dernière procédure de niveau n
- quand on revient de l'appel, on dépile

l'accès à une variable de niveau n se fait maintenant en temps constant

```
program foo;

procedure r();
begin ... end;

procedure p(n : integer);
  procedure q();
  begin r() end;
begin
  if n = 1 then p(2) else q()
end;

begin p(1) end.
```



la table a une taille connue à la compilation (niveau maximal) et peut être allouée à la base de la pile par exemple

correction de la compilation

le compilateur doit respecter la **sémantique** du langage (correction)

si le langage source est muni d'une sémantique $\rightarrow_s$ et le langage machine d'une sémantique $\rightarrow_m$, et si l'expression e est compilée en $C(e)$ alors on doit avoir un « diagramme qui commute » :

$$\begin{array}{ccc} e & \xrightarrow{\star}_s & v \\ \downarrow & & \approx \\ C(e) & \xrightarrow{\star}_m & v' \end{array}$$

où $v \approx v'$ exprime que les valeurs v et v' coïncident

considérons uniquement les expressions arithmétiques sans variable

$$e ::= n \mid e + e \mid e - e$$

et montrons la correction de la compilation

on se donne une sémantique à réductions pour le langage source

$$\begin{array}{rcl} v & ::= & n \\ E & ::= & \square \mid E + e \mid v + E \mid E - e \mid v - E \end{array}$$

$$\begin{array}{l} n_1 + n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 + n_2 \\ n_1 - n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 - n_2 \end{array}$$

on se donne de même une sémantique à réductions pour le langage cible

$$\begin{array}{rcl} m & ::= & \texttt{movq}\ \$n, r \\ & \mid & \texttt{addq}\ \$n, r \mid \texttt{addq}\ r, r \mid \texttt{subq}\ \$n, r \mid \texttt{subq}\ r, r \\ & \mid & \texttt{movq}\ (r), r \mid \texttt{movq}\ r, (r) \mid \\ r & ::= & \texttt{\%rdi} \mid \texttt{\%rsi} \mid \texttt{\%rsp} \end{array}$$

un état S est la donnée de valeurs pour les registres, R,
et d'un état de la mémoire, M

$$\begin{array}{rcl} R & ::= & \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\} \\ M & ::= & \mathbb{N} \to \mathbb{Z} \end{array}$$

on définit la sémantique d'une instruction m par une réduction de la forme

$$R, M, m \xrightarrow{m} R', M'$$

et d'une suite d'instructions $m_1, \dots, m_k$ comme la clôture transitive

la réduction $R, M, m \xrightarrow{m} R', M'$ est définie par

$$
\begin{array}{rcl}
R, M, \texttt{movq } \$n, r & \xrightarrow{m} & R\{r \mapsto n\}, M \\
R, M, \texttt{addq } \$n, r & \xrightarrow{m} & R\{r \mapsto R(r) + n\}, M \\
R, M, \texttt{addq } r_1, r_2 & \xrightarrow{m} & R\{r_2 \mapsto R(r_1) + R(r_2)\}, M \\
 & \text{idem pour } \texttt{subq} & \\
R, M, \texttt{movq } (r_1), r_2 & \xrightarrow{m} & R\{r_2 \mapsto M(R(r_1))\}, M \\
R, M, \texttt{movq } r_1, (r_2) & \xrightarrow{m} & R, M\{R(r_2) \mapsto R(r_1)\}
\end{array}
$$

on souhaite montrer que si

$$e \xrightarrow{\star} n$$

et si

$$R, M, code(e) \xrightarrow{m}{}^{\star}\ R', M'$$

alors $R'(\texttt{\%rdi}) = n$

on procède par récurrence structurelle sur e

on établit un résultat plus fort (**invariant**), à savoir :

si $e \xrightarrow{\star} n$ et $R, M, code(e) \xrightarrow{m}^{\star} R', M'$ alors

$$\begin{cases} R'(\texttt{\%rdi}) = n \\ R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) \\ \forall a \geq R(\texttt{\%rsp}),\ M'(a) = M(a) \end{cases}$$

- cas $e = n$

  on a $e \xrightarrow{\star} n$ et $code(e) = \texttt{movq \$}n\texttt{, \%rdi}$ et le résultat est immédiat

- cas $e = e_1 + e_2$

  on a $e \xrightarrow{\star} n_1 + e_2 \xrightarrow{\star} n_1 + n_2 \xrightarrow{\star} n$ avec $n = n_1 + n_2$, et

$$
\begin{array}{rl}
code(e) = & code(e_1) \\
& \texttt{addq \$-8, \%rsp} \\
& \texttt{movq \%rdi, (\%rsp)} \\
& code(e_2) \\
& \texttt{movq (\%rsp), \%rsi} \\
& \texttt{addq \$8, \%rsp} \\
& \texttt{addq \%rsi, \%rdi}
\end{array}
$$

	R, M	
$code(e_1)$	R_1, M_1	par hypothèse de récurrence $R_1(\texttt{\%rdi}) = n_1$ et $R_1(\texttt{\%rsp}) = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp}),\ M_1(a) = M(a)$
`addq $-8, %rsp` `movq %rdi, (%rsp)`	R'_1, M'_1	$R'_1 = R_1\{\texttt{\%rsp} \mapsto R(\texttt{\%rsp}) - 8\}$ $M'_1 = M_1\{R(\texttt{\%rsp}) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	par hypothèse de récurrence $R_2(\texttt{\%rdi}) = n_2$ et $R_2(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8$ $\forall a \geq R(\texttt{\%rsp}) - 8,\ M_2(a) = M'_1(a)$

`movq (%rsp), %rsi`
`addq $8, %rsp`
`addq %rsi, %rdi`
	R', M_2	$R'(\texttt{\%rdi}) = n_1 + n_2$
		$R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8 + 8 = R(\texttt{\%rsp})$
		$\forall a \geq R(\texttt{\%rsp}),$
		$M_2(a) = M'_1(a) = M_1(a) = M(a)$

- TD cette semaine
  - aide au projet
- cours la semaine prochaine
  - compilation des langages fonctionnels