École Normale Supérieure

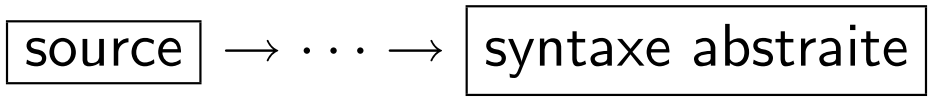
Langages de programmation et compilation

Jean-Christophe Filliâtre

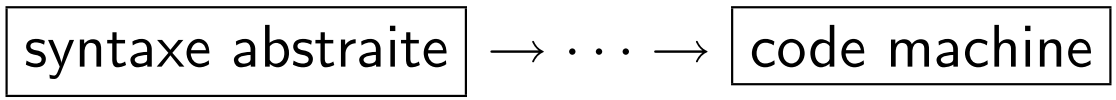
Cours 7 / 13 novembre 2008

Rappels

on a terminé la phase d'**analyse**



on considère maintenant la phase de **synthèse**



Rappels

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

si le langage source est muni d'une sémantique $\longrightarrow_s$ et le langage machine d'une sémantique $\longrightarrow_m$, 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

Objectif

dans le cours d'aujourd'hui, on se focalise sur la programmation **impérative**, en considérant un fragment de Pascal comme exemple

la programmation **fonctionnelle** et la programmation **objet** feront l'objet de cours spécifiques

Mini-Pascal

on considère le fragment suivant de Pascal

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

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

$$
\begin{array}{rcl}
F & \to & x\texttt{: integer} \\
& | & \texttt{var } x\texttt{: integer}
\end{array}
$$

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

Exemple

```
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.
```

Un peu de vocabulaire

dans la **déclaration** de procédure

```
procedure p(x1, ..., xn) ;
  var v1,...,vm ;
  ...
```

les variables `x1,...,xn` sont appelées **paramètres formels** de `p`
et les variables `v1,...,vm` sont appelées les **variables locales** de `p`

dans l'**appel** de procédure

```
p(e1, ..., en)
```

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

Niveau des déclarations

les procédures pouvant être imbriquées, on introduit quelques définitions

Définition (niveau)

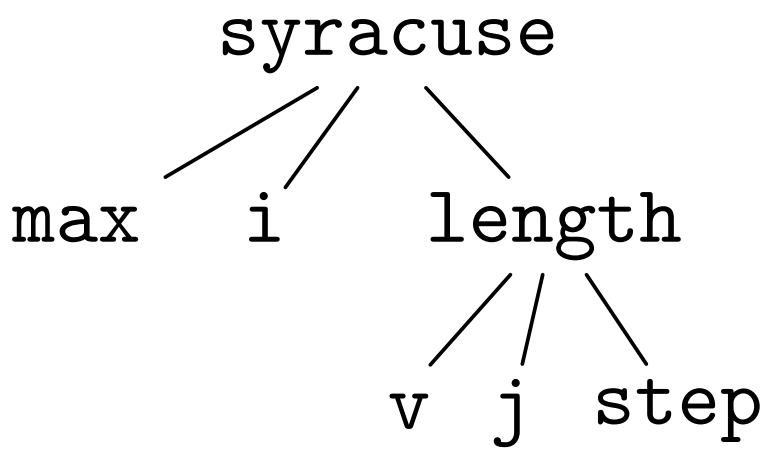
*Le **niveau** d'une déclaration (de variable ou de procédure) est le nombre de procédures sous lesquelles elle est déclarée. Le programme principal a le niveau 0.*

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.
```



Portée

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

Portée

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)

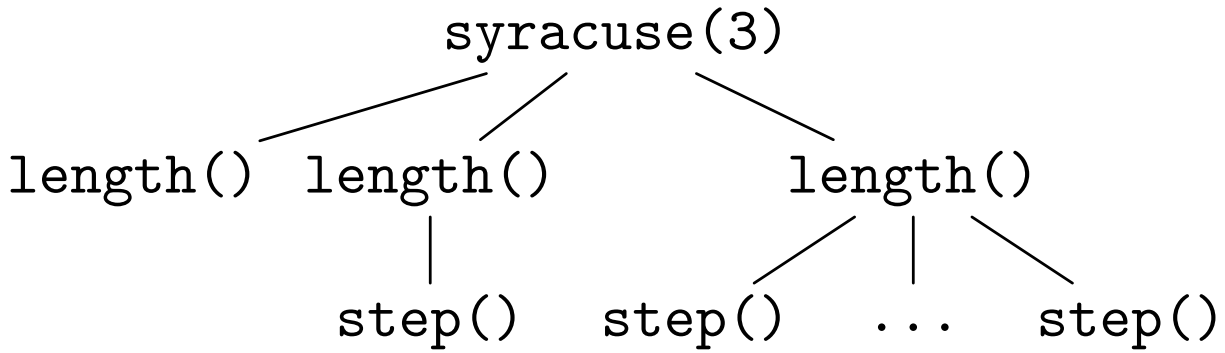
```
program syracuse ;

procedure syracuse(max1 : integer) ;
var i1 : integer ;
   procedure length() ;
   var v2,j2 : integer ;
      procedure step() ;
      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 step() ; writeln(v2)
   end ;
begin
   i1 := 1 ;
   while i1 <= max1 do begin length() ; i1 := i1+1 end
end ;

begin syracuse(100) end.
```

Arbre d'activation

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 $p(e_1, \ldots, e_n)$ et les sous-nœuds correspondent aux appels directement effectués par $p(e_1, \ldots, e_n)$.*

exemple :

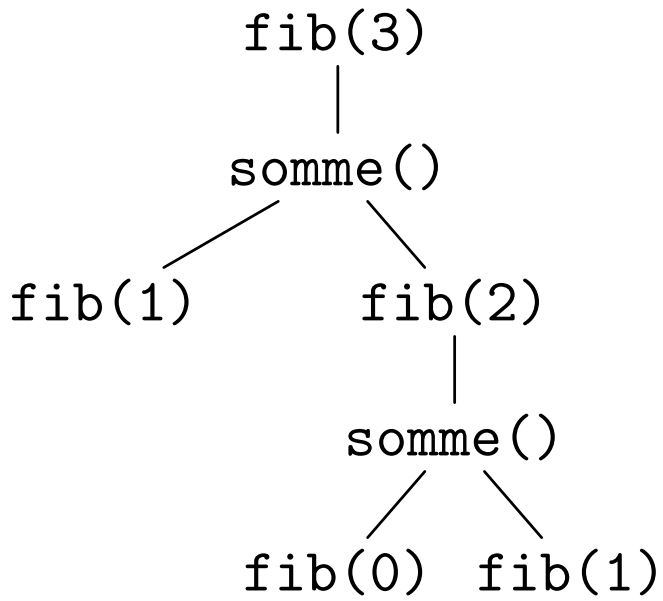


Arbre d'activation

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 := 1 else somme()
end ;

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



Procédure active

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 □

Organisation des variables en mémoire

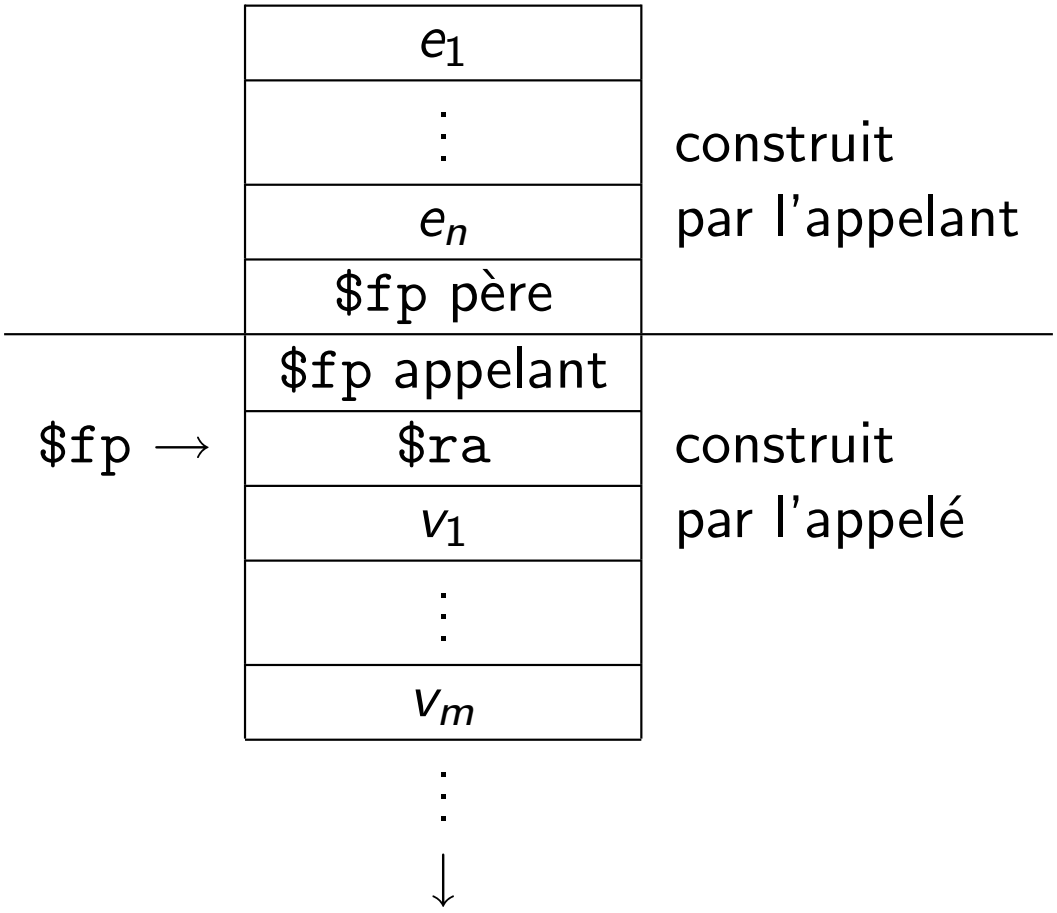
il faut choisir un emplacement mémoire pour chaque variable et être capable de **calculer** cet emplacement à l'exécution

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

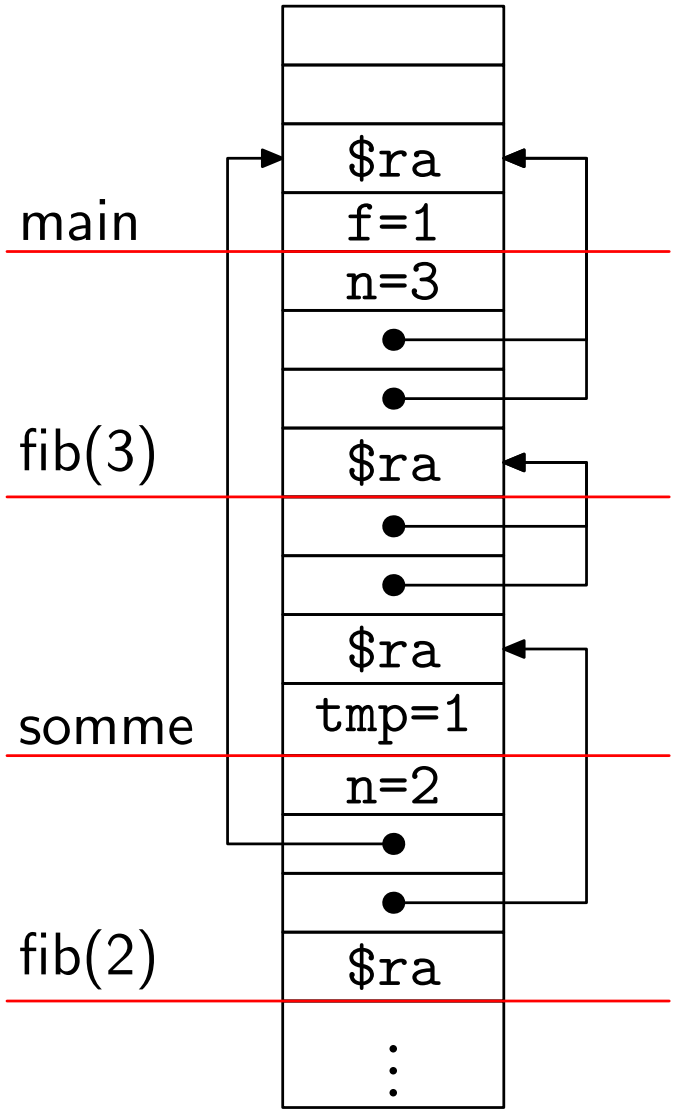
Tableau d'activation

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



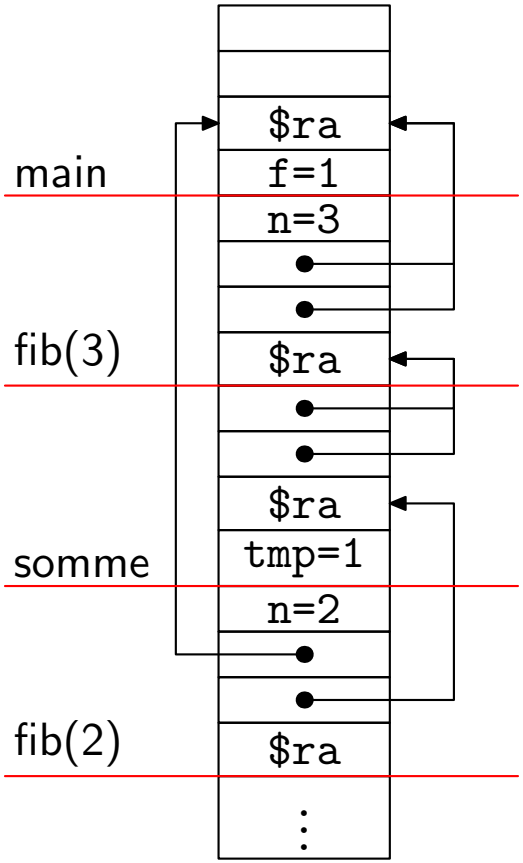
Tableaux d'activation : exemple

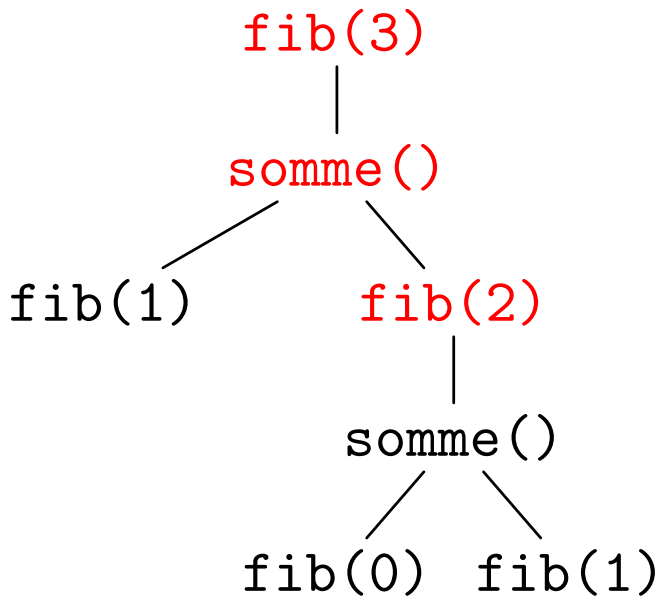
```
program fib ;

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 := 1 else somme()
end ;

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



Tableaux et arbre d'activation

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





Identificateurs

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 !

Facilité

on suppose disposer d'une facilité pour écrire du code MIPS directement dans le code Ocaml, avec la notation ‘...‘ et un échappement {}, et une concaténation ++

exemple : la fonction suivante empile la valeur d'un registre `r`

```
let pushr r =
  ‘ sub $sp, $sp, 4
    sw  {r}, 0($sp) ‘
```

de même la fonction suivante dépile une valeur et stocke le résultat dans `r`

```
let popr r =
  ‘ lw  {r}, 0($sp)
    add $sp, $sp, 4 ‘
```

Expressions arithmétiques

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 -> mips
```

à l'issue de l'exécution, le résultat de l'expression doit se trouver dans `$a0`

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

Expressions arithmétiques

on commence par les constantes entières

```
let rec int_expr lvl = function
  | Econst n ->
      `li  $a0, {n}`
```

et les opérations arithmétiques

```
  | Ebinop (Badd, e1, e2) ->
      int_expr lvl e1 ++ pushr $a0 ++
      int_expr lvl e2 ++
      popr $t1 ++
      `add $a0, $t1, $a0`
  | Ebinop (Bsub, e1, e2) ->
      ...
```

Efficacité

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

```
        li   $a0, 1
        sub  $sp, $sp, 4
        sw   $a0, 0($sp)
        li   $a0, 2
        lw   $t1, 0($sp)
        add  $sp, $sp, 4
        add  $a0, $t1, $a0
```

alors même que l'on dispose de 32 registres

Expressions arithmétiques

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);
      `move $t1, $fp` ++
      iter (lvl - l) `lw $t1, 8($t1)` ++
      `lw $a0, {ofs}($t1)`
```

avec

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

Expressions booléennes

de même, les expressions booléennes sont compilées avec une fonction

```
val bool_expr : int -> bool_expr -> mips
```

d'une manière très analogue

```
let rec bool_expr lvl = function
  | Bcmp (Beq, e1, e2) ->
      int_expr lvl e1 ++ pushr $a0 ++
      int_expr lvl e2 ++
      popr $t1 ++
      `seq $a0, $t1, $a0`

  | ... (* 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`)

Instructions

les instructions sont compilées avec une fonction

```
val stmt : int -> stmt -> mips
```

```
let rec stmt lvl = function
  | Swriteln e ->
      int_expr lvl e ++
      `li  $v0, 1
       syscall
       li  $v0, 4
       la  $a0, newline
       syscall          `
```

Instructions

```
| 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
```

Appel de procédure

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 code s -> code ++ int_expr lvl s ++ pushr $a0)
      nop el ++
    `move $t1, $fp` ++
    iter (lvl - l) `lw  $t1, 8($t1)` ++ pushr T1 ++
    `jal  {p}` ++
    popn (1 + List.length el)
```

Affectation

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)

Affectation

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 ++
    `move $t1, $fp` ++
    iter (lvl - l) `lw  $t1, 8($t1)` ++
    `sw  $a0, {ofs}($t1)`
```

Modes de passage des paramètres

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)

Exemple

```
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 89 }
end.
```

Passage par référence

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)

Passage par référence

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
```

Passage par référence

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);
      `move $t1, $fp` ++
      iter (lvl - l) `lw  $t1, 8($t1)` ++
      `add  $a0, $t1, {ofs}` ++
      if br then `lw  $a0, 0($a0)` else nop
```

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

Passage par référence

il faut aussi modifier le calcul des valeurs droites !

```
  | Evar { level = l; offset = ofs; by_reference = br } ->
      assert (l <= lvl);
      `move $t1, $fp` ++
      iter (lvl - l) `lw  $t1, 8($t1)` ++
      `lw  $a0, {ofs}($t1)` ++
      if br then `lw  $a0, 0($a0)` else nop
```

Passage par référence

ainsi que celui de l'affectation !

```
| Sassign ({level=l ; offset=ofs ; by_reference=br}, e) ->
    int_expr lvl e ++
    `move $t1, $fp` ++
    iter (lvl - l) `lw  $t1, 8($t1)` ++
    (if br then `lw   $t1, {ofs}($t1)`
           else `add  $t1, $t1, {ofs}`) ++
    `sw  $a0, 0($t1)`
```

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

Compilation des déclarations

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
```

Compilation des déclarations

on compile une déclaration avec

```
val decl : decl -> mips
```

```
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
```

Compilation d'une procédure

```
let frame_size dl = (* 4 fois le nombre de variables de dl *)

let procedure p =
  let fs = 8 + frame_size p.locals in
  `{p.name.pident} :
```

```
        sub  $sp, $sp, {fs}       # alloue la frame
        sw   $fp, {fs - 4}($sp)   # sauve $fp
        sw   $ra, {fs - 8}($sp)   # sauve $ra
        add  $fp, $sp, {fs - 8}   # affecte $fp
```

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

```
  `     lw   $ra, 0($fp)        # restaure $ra
        lw   $fp, 4($fp)        # restaure $fp
        add  $sp, $sp, {fs}     # désalloue la frame
        jr   $ra                `
```

Le programme principal

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

```
let prog p =
  let fs = 12 + frame_size p.locals in
  let code_main = stmt 0 p.body in
  let code_procs = decls p.locals in
```

```
  `main:
        sub  $sp, $sp, {fs}       # alloue la frame
        add  $fp, $sp, {fs - 12}  # affecte $fp
        sw   $ra, 0($fp)          # sauve $ra
  ` ++ code_main ++
  `     lw   $ra, 0($fp)          # restaure $ra
        add  $sp, $sp, {fs}       # désalloue la frame
        jr   ra
  ` ++ code_procs
```

démo

Encore plus efficace

on peut améliorer l'accès aux variables

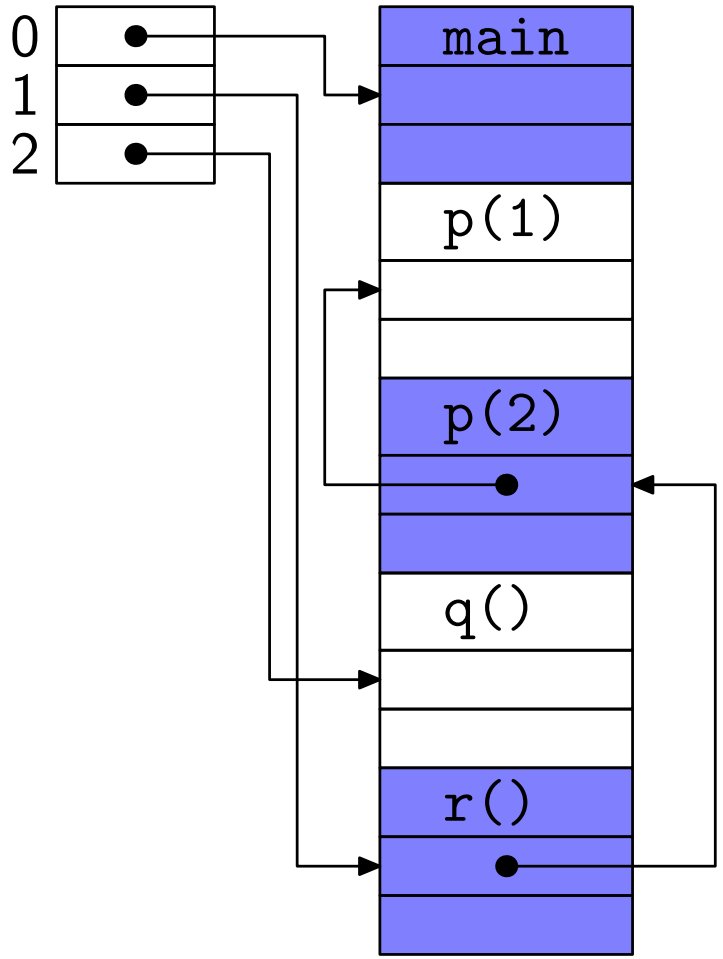
l'idée est de conserver dans une table un pointeur vers le **dernier** tableau d'activation de chaque niveau ; et les tableaux d'un même niveau forment une liste chaînée

- 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

Exemple

```
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 alloué à la base de la pile par exemple

Correction de la compilation

Correction de la compilation

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

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

$$n_1 + n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 + n_2$$
$$n_1 - n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 - n_2$$

Correction de la compilation

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

$$
\begin{array}{rcl}
m & ::= & \texttt{li}\ r, n \\
& | & \texttt{add}\ r, r, n \mid \texttt{add}\ r, r, r \mid \texttt{sub}\ r, r, n \mid \texttt{sub}\ r, r, r \\
& | & \texttt{lw}\ r, r \mid \texttt{sw}\ r, r \mid \\
r & ::= & \texttt{\$a0} \mid \texttt{\$t1} \mid \texttt{\$sp}
\end{array}
$$

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

$$
\begin{array}{rcl}
R & ::= & \{\texttt{\$a0} \mapsto n; \texttt{\$t1} \mapsto n; \texttt{\$sp} \mapsto n\} \\
M & ::= & \mathbb{N} \to \mathbb{Z}
\end{array}
$$

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 $m_1, \ldots, m_k$ comme la clôture transitive

Correction de la compilation

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

$$
\begin{aligned}
R, M, \texttt{li } r, n &\xrightarrow{m} R\{r \mapsto n\}, M \\
R, M, \texttt{add } r_1, r_2, n &\xrightarrow{m} R\{r_1 \mapsto R(r_2) + n\}, M \\
R, M, \texttt{add } r_1, r_2, r_3 &\xrightarrow{m} R\{r_1 \mapsto R(r_2) + R(r_3)\}, M \\
R, M, \texttt{lw } r_1, r_2 &\xrightarrow{m} R\{r_1 \mapsto M(R(r_2))\}, M \\
R, M, \texttt{sw } r_1, r_2 &\xrightarrow{m} R, M\{R(r_2) \mapsto R(r_1)\}
\end{aligned}
$$

Correction de la compilation

on souhaite montrer que si

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

et si

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

alors $R'(\$a0) = n$

on procède par récurrence structurelle sur e

Correction de la compilation

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

$$\begin{cases} R'(\$\mathtt{a0}) = n \\ R'(\$\mathtt{sp}) = R(\$\mathtt{sp}) \\ \forall a \geq R(\$\mathtt{sp}),\ M'(a) = M(a) \end{cases}$$

Correction de la compilation

- cas $e = n$

  on a $e \xrightarrow{\star} n$ et $code(e) = \texttt{li \$a0}, n$ 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

$$
\begin{aligned}
code(e) = \quad & code(e_1) \\
& \texttt{sub \$sp, \$sp, 4} \\
& \texttt{sw \$a0, \$sp} \\
& code(e_2) \\
& \texttt{lw \$t1, \$sp} \\
& \texttt{add \$sp, \$sp, 4} \\
& \texttt{add \$a0, \$t1, \$a0}
\end{aligned}
$$

Correction de la compilation

	R, M	
$code(e_1)$	R_1, M_1	par hypothèse de récurrence $R_1(\$a0) = n_1$ et $R_1(\$sp) = R(\$sp)$ $\forall a \geq R(\$sp),\ M_1(a) = M(a)$
`sub $sp,$sp,4` `sw $a0,$sp`	R'_1, M'_1	$R'_1 = R_1\{\$sp \mapsto R(\$sp) - 4\}$ $M'_1 = M_1\{R(\$sp) - 4 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	par hypothèse de récurrence $R_2(\$a0) = n_2$ et $R_2(\$sp) = R(\$sp) - 4$ $\forall a \geq R(\$sp) - 4,\ M_2(a) = M'_1(a)$

`lw $t1,$sp`
`add $sp,$sp,4`
`add $a0,$t1,$a0`
	R', M_2	$R'(\$a0) = n_1 + n_2$
		$R'(\$sp) = R(\$sp) - 4 + 4 = R(\$sp)$
		$\forall a \geq R(\$sp),$
		$M_2(a) = M'_1(a) = M_1(a) = M(a)$

Tableaux

Tableaux statiques

commençons par des **tableaux statiques**, c'est-à-dire dont les dimensions sont connues à la compilation

en Pascal, on peut introduire des tableaux à une dimension

```
var t : array [2..10] of integer ;
```

ou à plusieurs

```
var u : array [-5..5, 1..10] of integer ;
```

ou même des tableaux de tableaux

```
var v : array [0..11] of array [0..31] of integer ;
```

Taille des données

la taille d'une donnée (en octets) se calcule aisément par récurrence sur son type :

$$\begin{aligned} \mathit{taille}(\texttt{integer}) &= 4 \\ \mathit{taille}(\texttt{array}\ [l_1..u_1, \ldots, l_k..u_k]\ \texttt{of}\ \tau) &= \prod_{i=1}^{k}(u_i - l_i + 1) \times \mathit{taille}(\tau) \end{aligned}$$

exemple : le tableau

```
var u : array [-5..5, 1..10] of integer ;
```

occupe 440 octets en mémoire

Rangement des éléments d'un tableau

pour ranger les éléments d'un tableau multi-dimensionnel, on a *a priori* le choix entre un rangement par lignes ou par colonnes

cependant, si on déclare

```
var u : array [-5..5, 1..10] of integer;
```

on peut souhaiter que `u[i]` soit vu comme un élément de type

```
array [1..10] of integer
```

et il est donc souhaitable de privilégier un rangement par lignes

Calcul d'adresse

soit un tableau

$$t : \texttt{array}\ [l_1..u_1, \ldots, l_k..u_k]\ \texttt{of}\ \tau$$

rangé en mémoire à l'adresse *base*

l'emplacement mémoire a de l'élément $t[i_1, \ldots, i_k]$ est alors

$$\begin{aligned} a = \mathit{base} \quad & + \quad (i_1 - l_1) \times \mathit{taille}(\texttt{array}\ [l_2..u_2, \ldots, l_k..u_k]\ \texttt{of}\ \tau) \\ & + \quad (i_2 - l_2) \times \mathit{taille}(\texttt{array}\ [l_3..u_3, \ldots, l_k..u_k]\ \texttt{of}\ \tau) \\ & \quad \vdots \\ & + \quad (i_{k-1} - l_{k-1}) \times \mathit{taille}(\texttt{array}\ [l_k..u_k]\ \texttt{of}\ \tau) \\ & + \quad (i_k - l_k) \times \mathit{taille}(\tau) \end{aligned}$$

Calcul d'adresse

posons $d_i \stackrel{\text{def}}{=} u_i - l_i + 1$

alors

$$\begin{aligned} a = base \quad &+ \quad (i_1 - l_1) \times d_2 \times \cdots \times d_k \times taille(\tau) \\ &+ \quad (i_2 - l_2) \times d_3 \times \cdots \times d_k \times taille(\tau) \\ &\vdots \\ &+ \quad (i_{k-1} - l_{k-1}) \times d_k \times taille(\tau) \\ &+ \quad (i_k - l_k) \times taille(\tau) \end{aligned}$$

soit

$$\begin{aligned} a \quad = \quad & (\ldots(i_1 \times d_2 + i_2) \times d_3 + i_3) \cdots + i_k) \times taille(\tau) + \\ & (base - (\ldots(l_1 \times d_2 + l_2) \times d_3 + l_3) \cdots + l_k) \times taille(\tau)) \end{aligned}$$

le second terme peut être calculé à la compilation ; le premier est calculé dynamiquement, avec la méthode de Horner pour limiter les multiplications

Passage de tableaux en paramètres

le principe de passage des paramètres ne change pas pour les tableaux statiques

on note cependant que

- les dimensions d'un tableau paramètre formel doivent être indiquées
  ```
  procedure p(t : array[2..5] of integer) ;
  ```
  pour permettre le calcul de l'adresse à l'intérieur de `p`
- les arguments n'occupent plus tous la même place dans le tableau d'activation (elle dépend maintenant du type)
- un tableau passé par valeur doit être copié, ce qui n'est plus une opération atomique
- un élément de tableau est une valeur gauche ; exemple
  ```
  var t : array[1..10] of integer ;
  procedure p(var x : integer) ; begin x := 42 end ;
  begin p(t[1]) end.
  ```

Tableaux dynamiques

imposer une taille statique aux tableaux est un contrainte (c'était un défaut de la conception originel de Pascal)

en pratique, on veut pouvoir déclarer des tableaux dont la taille n'est connue qu'à l'exécution, tels que

```
procedure p(n : integer) ;
var t : array [1..n*n] of integer ;
begin ... end ;
```

```
procedure tri(n : integer ; var t : array [1..n] of integer) ;
...
```

seul le nombre de dimensions est encore statique

Tableaux dynamiques

deux problèmes se posent

- la taille occupée en mémoire par le tableau n'est plus connue à la compilation ; en particulier, un tableau ne peut plus être rangé dans le tableau d'activation comme toute autre variable
- le calcul de l'adresse d'un élément $t[i_1, \ldots, i_k]$ nécessite les l_i et les d_i, qui ne sont plus connus statiquement

Tableaux dynamiques

la solution : le tableau d'activation contient un **descripteur de tableau**, qui a la forme suivante

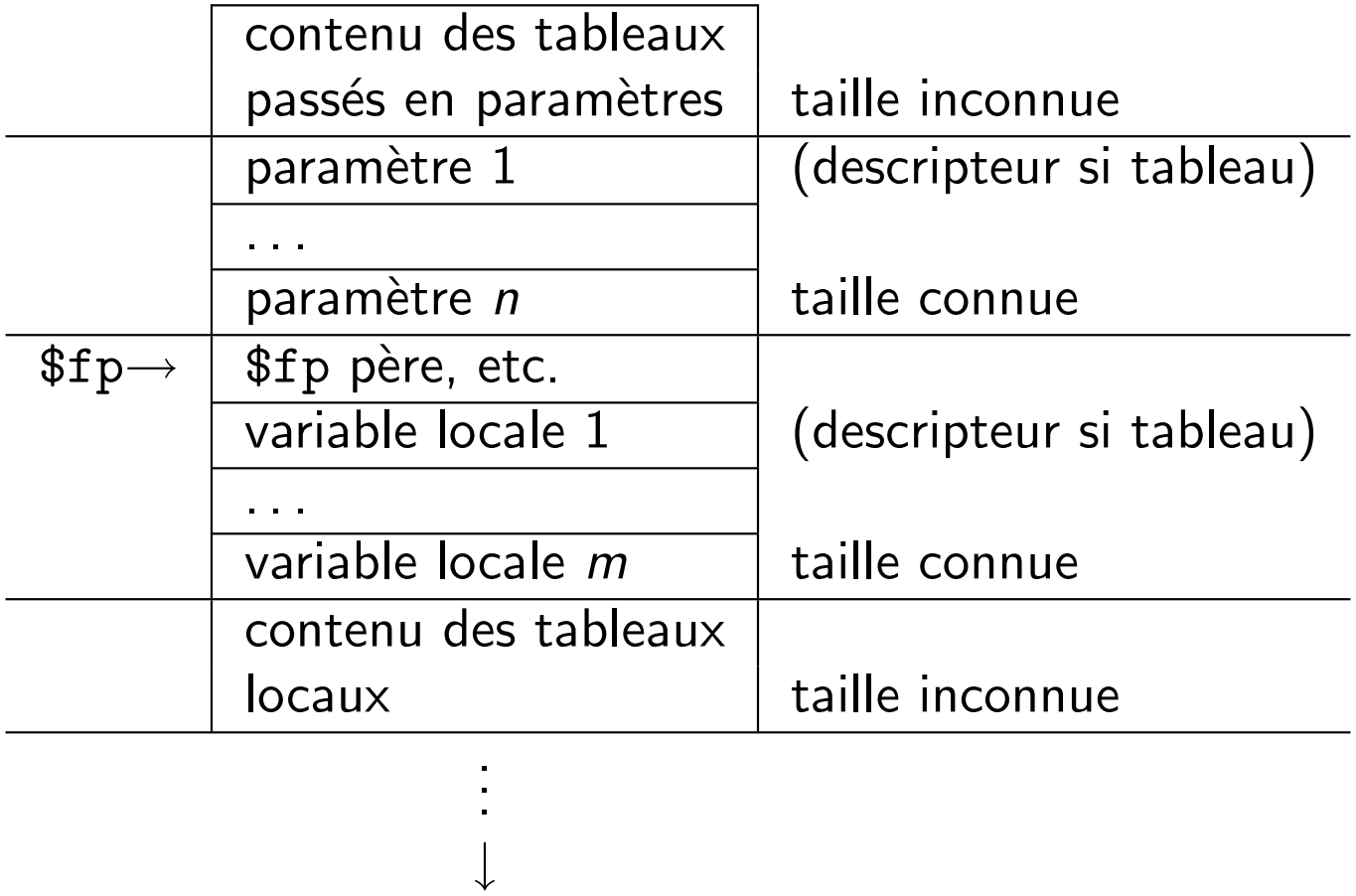
$base$
taille totale du tableau
$base - (\ldots(l_1 \times d_2 + l_2) \times d_3 + l_3) \cdots + l_k) \times taille(\tau)$
d_2
d_3
$\vdots$
d_k
$taille(\tau)$

en particulier, la taille de ce descripteur est connue à la compilation (elle ne dépend que du nombre de dimensions)

(note : si on veut tester les débordements d'indices, il faut également conserver les valeurs des l_i et u_i dans ce descripteur)

Tableaux dynamiques

on peut continuer de stocker les tableaux en pile mais il faut conserver des emplacements connus statiquement pour les paramètres et les variables locales ; le tableau d'activation prendre la forme suivante



Enregistrements

Enregistrements

en Pascal, on introduit ainsi un enregistrement

```
var e : record x : integer ; y : integer end
```

de manière générale, un type d'enregistrement est de la forme

$$\tau = \texttt{record}\ x_1 : \tau_1\ ;\ x_2 : \tau_2\ ;\ \ldots x_n : \tau_n\ \texttt{end}$$

où les x_i sont appelés les champs de l'enregistrement

on accède à un champ avec la notation `e.x`, qui est une valeur gauche et une valeur droite

Enregistrements

un enregistrement est naturellement représenté par un bloc mémoire où les valeurs des différents champs sont stockées consécutivement

la taille occupée est connue à la compilation :

$$\mathit{taille}(\texttt{record}\ x_1 : \tau_1\ \texttt{;}\ x_2 : \tau_2\ \texttt{;}\ \ldots x_n : \tau_n\ \texttt{end}) = \sum_{i=1}^{n} \mathit{taille}(\tau_i)$$

note : si un champ est un tableau dynamique, alors la valeur de ce champ est le descripteur de ce tableau (dont la taille est connue statiquement)

Enregistrements

la valeur gauche d'un enregistrement est l'adresse du début de son emplacement, qui coïncide avec l'adresse de son premier champ

pour chaque champ x_i on peut calculer statiquement le **décalage** $\delta(x_i)$ de ce champ par rapport au début

$$\delta(x_i) = \sum_{j=1}^{i-1} \mathit{taille}(\tau_j)$$

la valeur gauche de $e.x_i$ est obtenue en ajoutant $\delta(x_i)$ à la valeur gauche de e

quelques mots sur le langage C

Le langage C

le langage C est un langage impératif relativement bas niveau, notamment par 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

Les types du C

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

- un type $\tau*$ des pointeurs vers des valeurs de type τ

  si p est un pointeur de type $\tau*$, 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 $\tau*$

- 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

Mode de passage

le mode de passage des paramètres en C est **par valeur**

en particulier, 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`

en pratique, on utilise surtout des pointeurs sur des structures

Tableaux

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 ici à ajouter à `t` la quantité `4i` car il s'agit d'un tableau d'entiers)

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

Valeurs gauches du C

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

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

Devinette

les programmes suivants sont-ils corrects ?

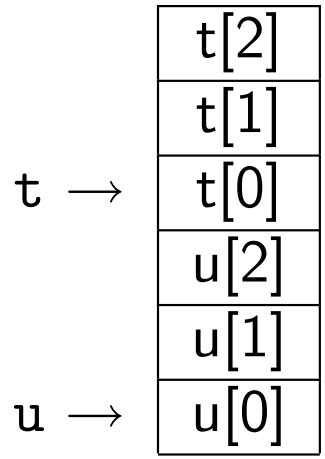
```
void p() {
  int t[3];
  int u[3];
  t = u;
}

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

Devinette

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

non, car `t` et `u` sont vraiment des tableaux, et l'affectation de tableaux n'est pas autorisée

si on pense en termes de tableau d'activation, c'est clair



en revanche on peut écrire

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

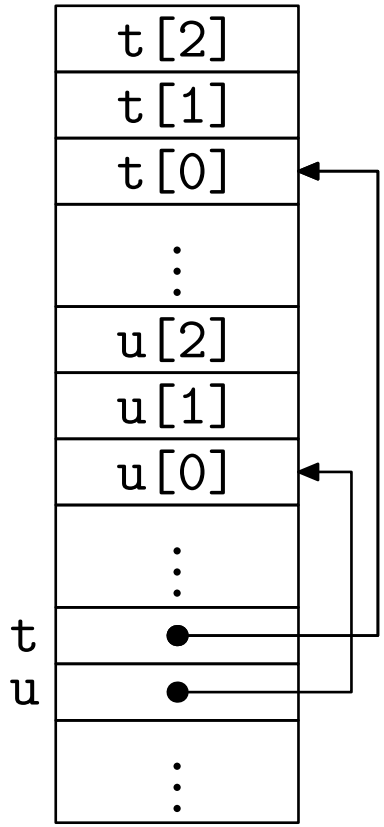
Devinette

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

oui, car c'est exactement la même chose que

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

et l'affectation de pointeurs est autorisée



Référence fantôme

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*)

La semaine prochaine

la semaine prochaine

- TD 7 le mercredi 19 novembre
- Cours 8 le jeudi 20 novembre
  - compilation des langages fonctionnels