IFT3065/IFT6232 Compilation

http://www.iro.umontreal.ca/~feeley/cours/ift3065-ift6232/

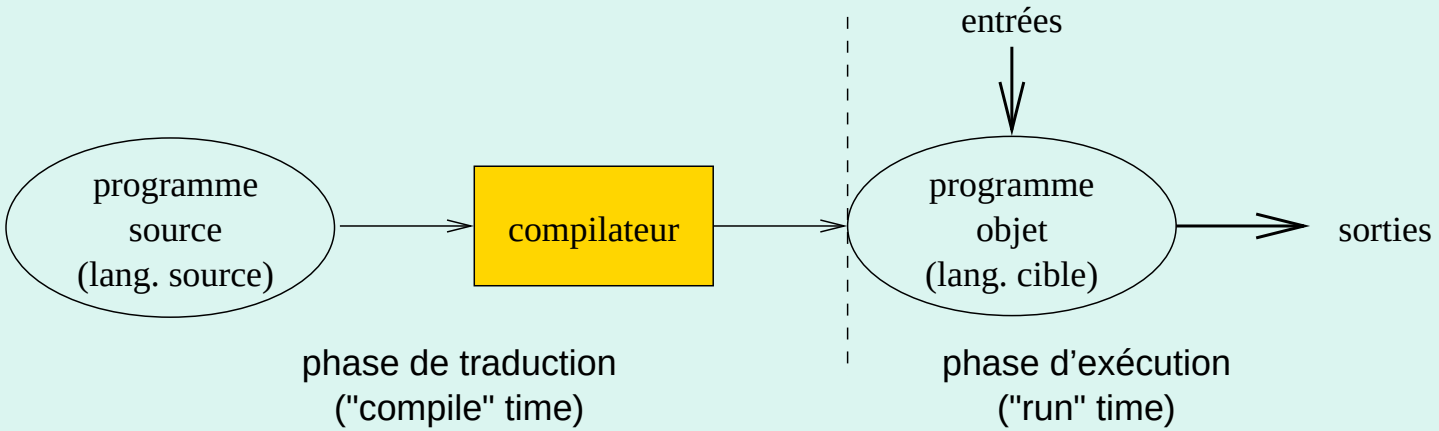
Outils de développement

- Le développement de logiciel nécessite plusieurs outils:
  - **Editeur de texte**: édition du code source (p.ex. **emacs**, **vi**)
  - **Compilateur**: production de code exécutable (p.ex. **gcc**, **javac**)
  - **Déboggeur**: élimination des bugs (p.ex. **gdb**, **gprof**, **valgrind**)
  - **Editeur de liens**: fusion des codes exécutables (p.ex. **ld**, **link**)
  - **Gestionnaire de source**: gestion des versions du code source (p.ex. **git**, **subversion**, **RCS**, **CVS**)
  - **Système de "build"**: automatisation de la construction d'un logiciel (p.ex. **make**, **configure**, **scons**)
- Ces outils sont intégrés dans les **IDE** ("Integrated Development Environment"), p.ex. Microsoft Visual-C++
- Ce cours étudie principalement les **compilateurs**

Implantation des langages de programmation (1)

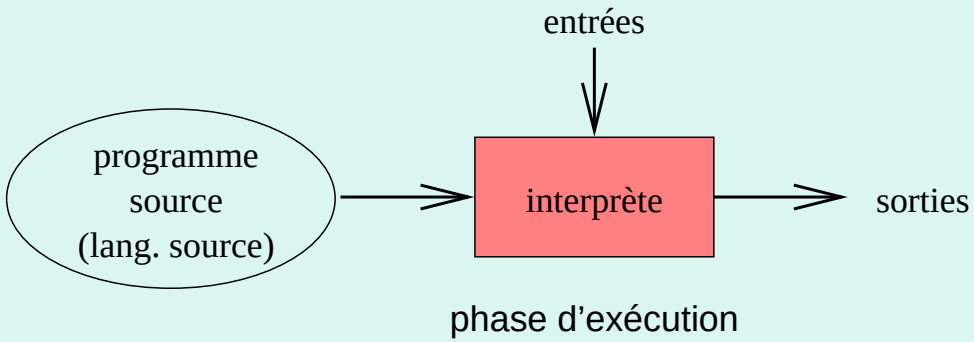
La **compilation** et l'**interprétation** sont les deux approches principales pour implanter un langage de programmation

- Approche par **compilation**: 2 phases
  1. **traduction** du programme source en un programme objet **équivalent** en langage machine
  2. **exécution** du programme objet par la machine



Implantation des langages de programmation (2)

- Approche par **interprétation**: 1 seule phase
  1. **exécution** du programme source lors de sa lecture



- **L'exécution est plus lente** si le programme est exécuté plusieurs fois ou s'il contient des boucles (répétition du décodage)
- Cependant un interprète est **plus prompt, compact, flexible et portable** (les shells de commande, p.ex. sh, sont souvent implantés par des interprètes)

Implantation des langages de programmation (3)

C++ compilé comparé au “shell script” interprété :

Fichier: "somme.cc"

```
#include <iostream>
using namespace std;
main ()
{ long s = 0;
  for (int i=1; i<=10000000; i++)
    s = s + i;
  cout << s << endl;
}
```

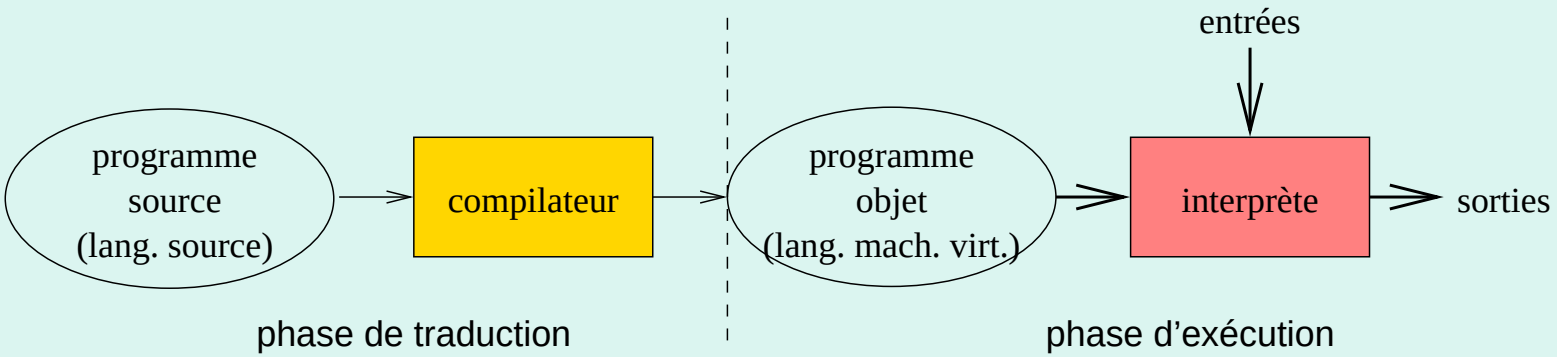
Fichier: "somme-script"

```
#! /bin/sh
(( s = 0 ))
for (( i=1 ; i<=10000000 ; i++ )) do
  (( s = s + i ))
done
echo $s
```

```
$ time g++ somme.cc -o somme
        0.27 real         0.21 user         0.04 sys
$ time ./somme
50000005000000
        0.03 real         0.02 user         0.00 sys
$ time ./somme-script
50000005000000
      153.37 real       141.36 user        11.53 sys
$ ls -l somme somme-script
-rwxr-xr-x  1 feeley  feeley  9688 Jan 11 17:30 somme
-rwxr-xr-x  1 feeley  feeley    91 Jan 11 17:33 somme-scrip
```

Implantation des langages de programmation (4)

- Un interprète implante en quelquesorte une “**machine virtuelle**” (son langage machine = langage source de l’interprète)
- **Approche hybride**: compilation vers machine virtuelle (p.ex. JVM)

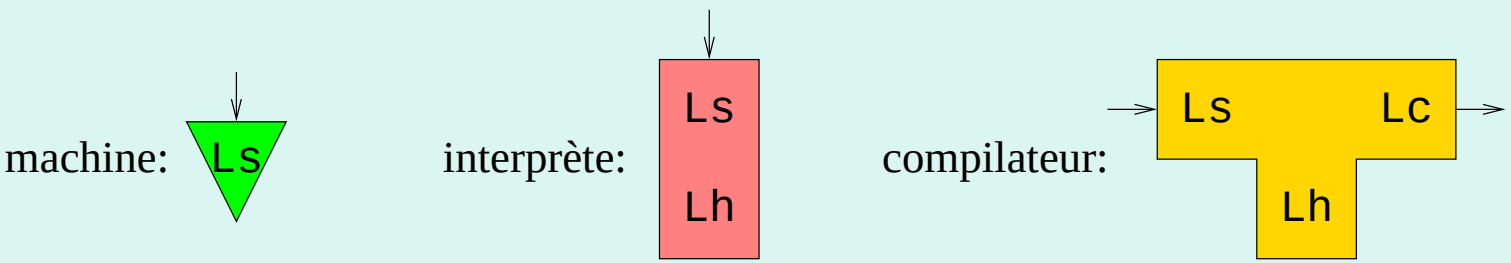


- C’est **très portable** (il suffit de porter l’interprète qui est normalement relativement simple) et **moyennement rapide** (évite une grande partie du décodage)

Machine, interprète et compilateur (1)

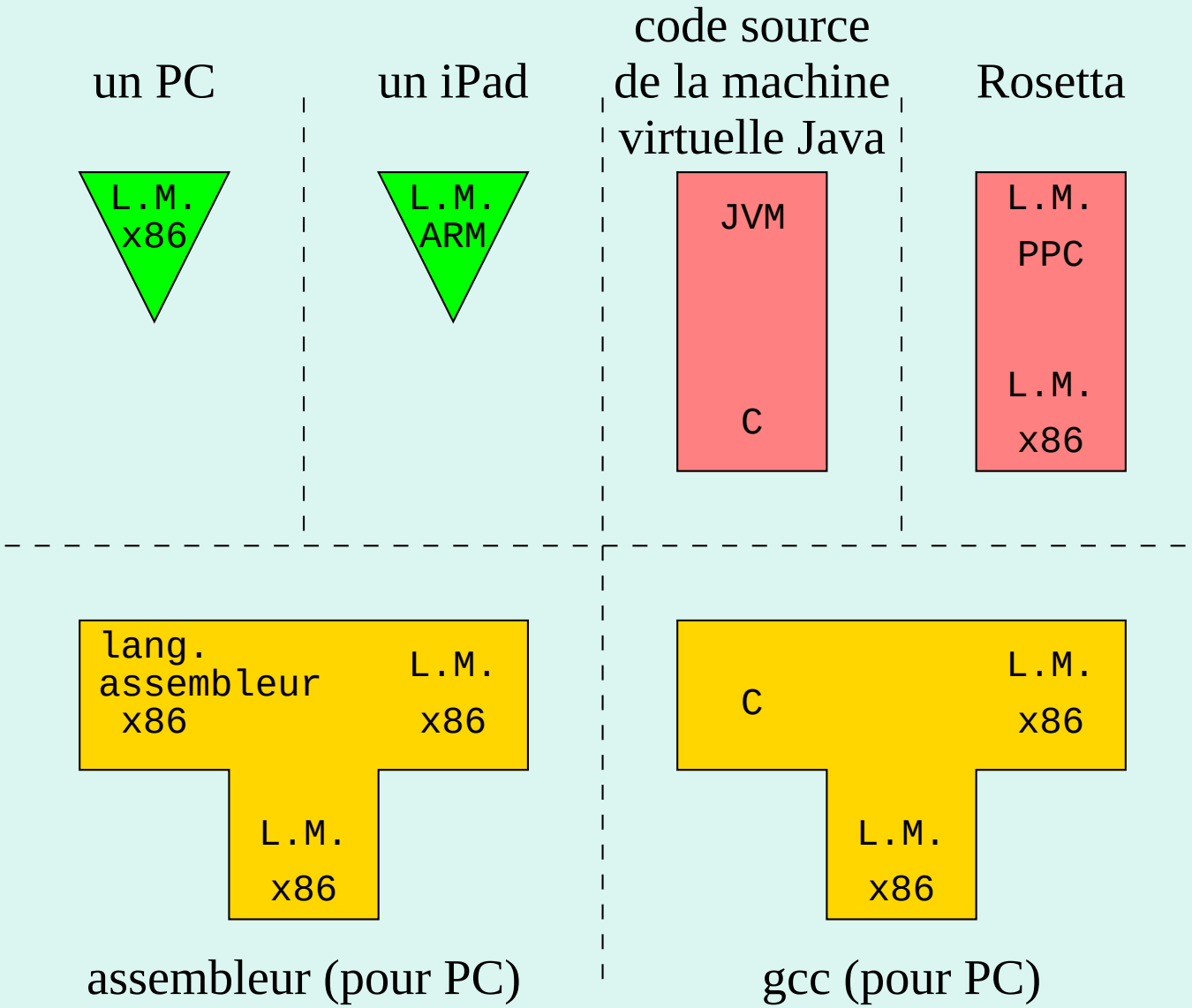
Définitions:

- Une **machine** est capable d'exécuter un programme exprimé dans son langage source **Ls**
- Un **interprète** est un programme exprimé en langage hôte **Lh** capable d'exécuter un programme exprimé dans le langage source **Ls**
- Un **compilateur** est un programme exprimé en langage hôte **Lh** capable de traduire un programme en langage source **Ls** en un programme en langage cible **Lc**
- Notation schématique :



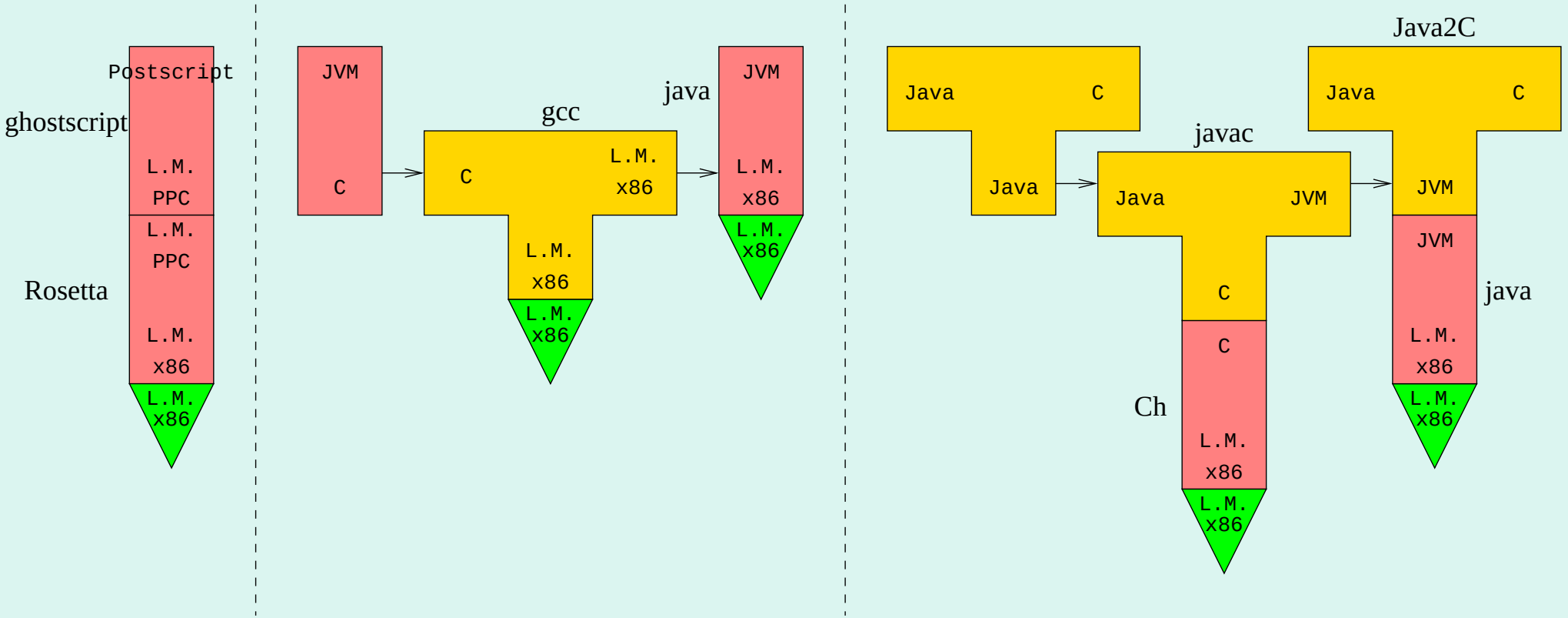
Machine, interprète et compilateur (2)

- Quelques exemples



Machine, interprète et compilateur (3)

- Un interprète ou compilateur **exécutable** peut être obtenu en combinant des machine(s), interprète(s) et compilateur(s)
- Par exemple



Variété de compilateur

Certaines combinaisons ont des usages particuliers:

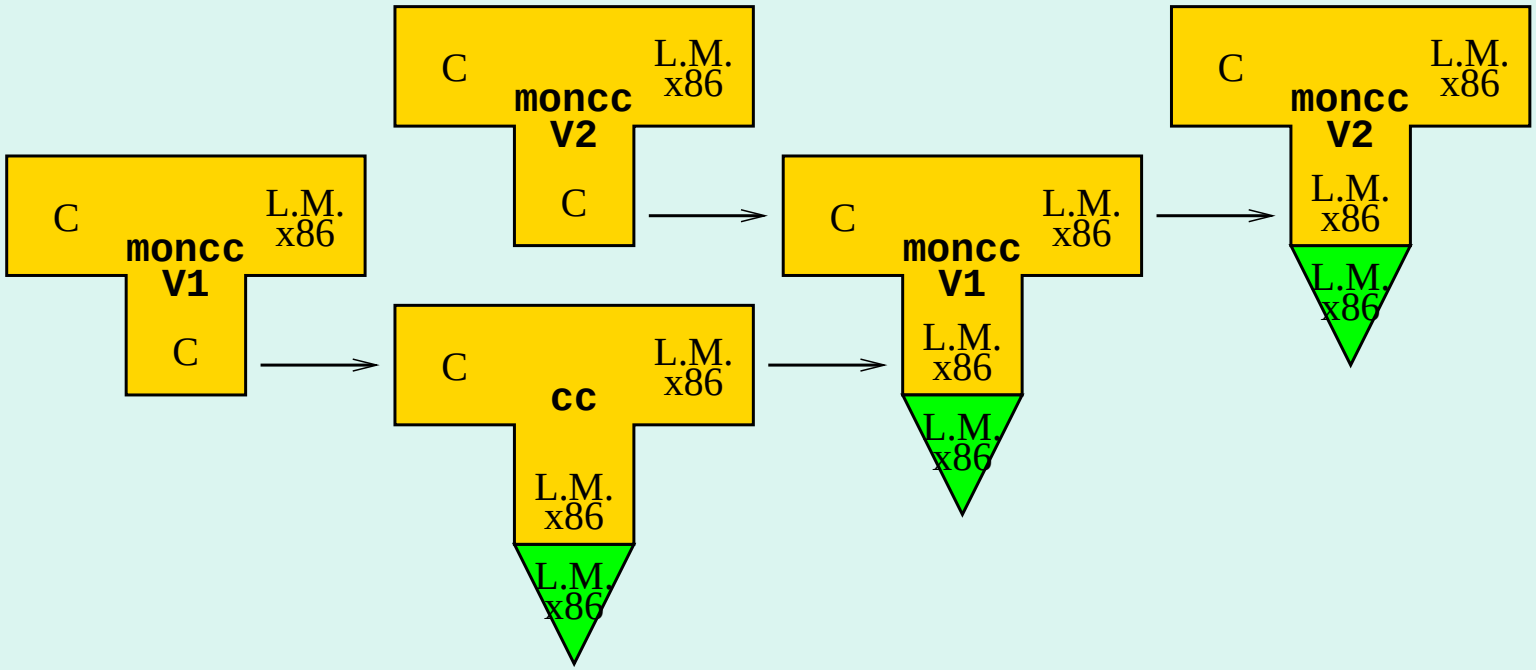
Ls	Lc	nom et utilité
	L. M. $=$ Lh	**compilateur natif** (p.ex. **gcc** typique)
	L. M. $\neq$ Lh	"**crosscompiler**" (p.ex. **javac**) systèmes avec peu de ressources
L. ass.	L. M.	**assembleur** (p.ex. **as**) programmation assembleur
L. M.	L. ass.	**désassembleur** (p.ex. **ndisasm**) déboggeur de bas niveau
L. M.	L. haut niv.	**décompilateur** (p.ex. **JAD**) déboggeur, piratage et maintenance de logiciels désuets
	$\subseteq$ Ls	**préprocesseur** (p.ex. **cpp**, **C++**)
Lh		**compilateur autogène** (p.ex. **gcc**) amorçage d'un compilateur

L’amorçage d’un compilateur (1)

- Problème de l’**amorçage** (“bootstrap”) d’un compilateur: comment créer un nouveau compilateur pour un langage **Ls**?
- Le compilateur peut être écrit dans n’importe quel langage hôte **Lh** pour lequel on a un compilateur ou interprète exécutable
- Évidemment c’est mieux si **Lh** est un langage de haut niveau (p.ex. Java, Scheme ou C) car si **Lh** est de bas niveau (p.ex. langage machine/assembleur) cela prendra beaucoup d’efforts de développement (mais c’est ce qui a été fait pour FORTRAN en 20 années-homme!)

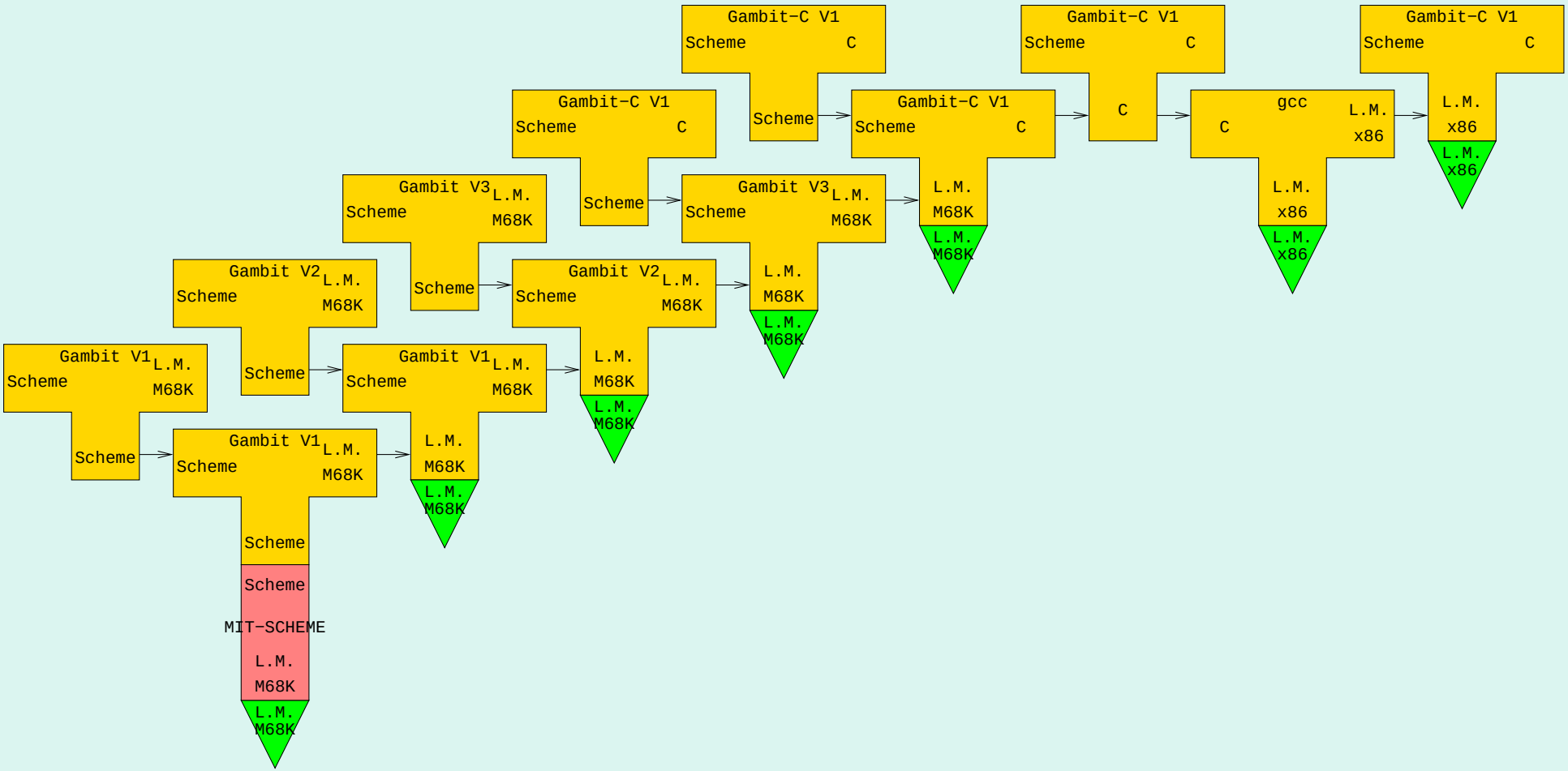
L'amorçage d'un compilateur (2)

- S'il existe déjà un interprète ou compilateur pour **Ls** il est pratique d'écrire un **compilateur autogène** c'est-à-dire où **Ls** = **Lh**
- Exemple d'un nouveau compilateur **moncc** pour le langage C



L'amorçage d'un compilateur (3)

- Exemple du compilateur Gambit pour Scheme

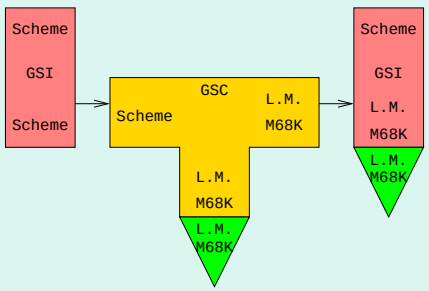


L’amorçage d’un compilateur (4)

- Avantages
  1. **Autosuffisance** (pas besoin d’un autre compilateur ou interprète pour la maintenance et extensions)
  2. Amélioration de la qualité du code généré par le compilateur a un impact sur la **performance du compilateur** (p.ex. réduction du temps de compilation)
- Désavantages
  1. Un bug dans le compilateur peut être très difficile à trouver et réparer (car il est peut être le résultat d’une compilation erronnée)
  2. Les outils périphériques existants (déboggeurs, etc) sont possiblement incompatibles (il faut donc les récrire)

Interprète méta-circulaire

- Le cas analogue pour les interprètes, c'est-à-dire **Ls** = **Lh**, se nomme **interprète méta-circulaire**
- La compilation d'un interprète méta-circulaire permet d'obtenir un interprète exécutable
- Exemple de l'interprète Gambit

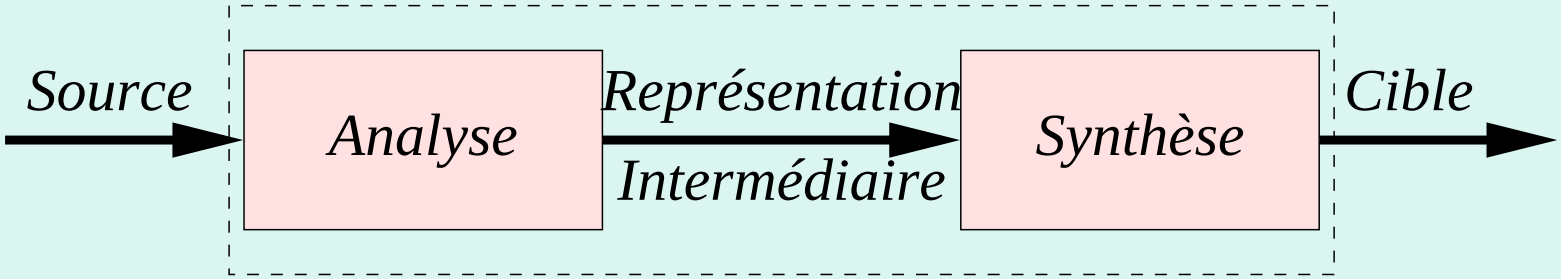


- De plus, un interprète méta-circulaire peut servir à décrire et éclaircir la sémantique d'un langage de programmation et le fonctionnement des interprètes (méta-circulaires ou non)

Défis de la compilation

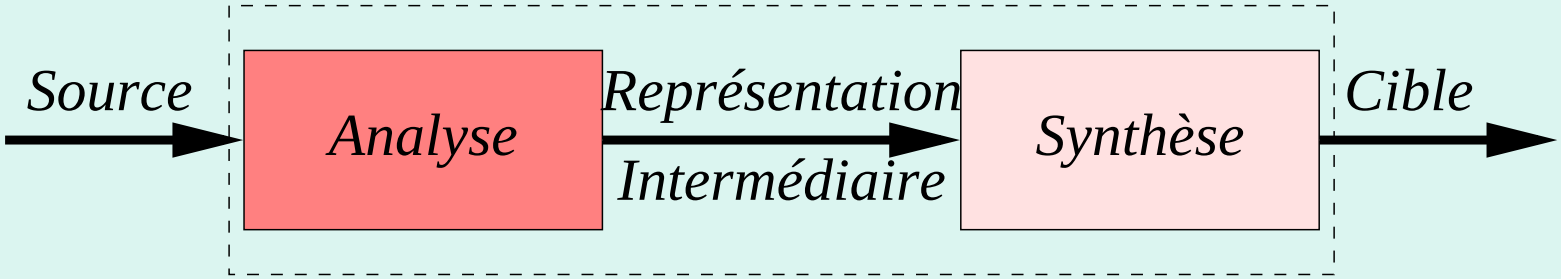
- Compilateur = programme qui traduit un **programme source** en un **programme cible** qui est **équivalent**
- Traduire entre langages ayant des **modèles d'exécution** très différents ("semantic gap")
  - **langage de haut niveau** $\Longrightarrow$ langage machine (jeu d'instruction, parallélisme, architecture)
- Garantir l'**équivalence sémantique** de la traduction
- Qualité de la traduction :
  - **Grande vitesse de compilation**
  - **Grande vitesse d'exécution du code généré**
  - **Petite taille du code**
  - **Petite consommation énergétique**

Structure d'un compilateur



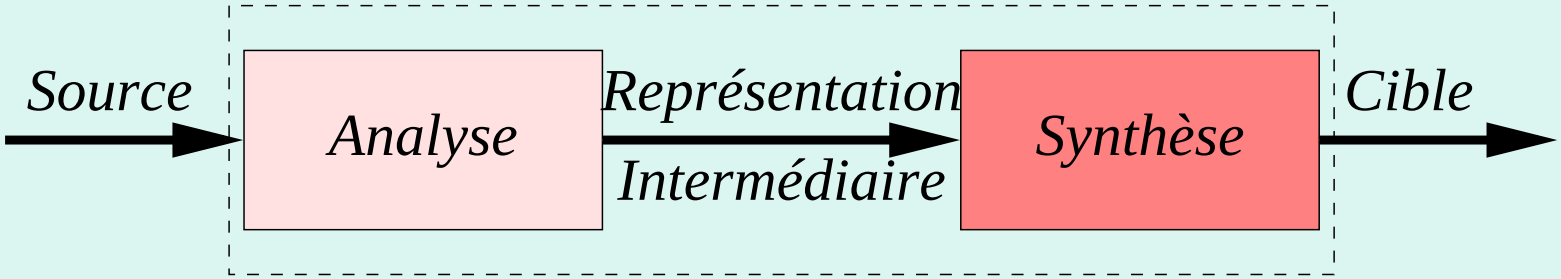
- Deux phases de compilation principales
  - **Analyse** : vérifie la validité du source et en extrait des propriétés
  - **Synthèse** : reconstitue le programme dans le langage cible
- La **représentation intermédiaire** sert à véhiculer les informations requises par la phase de synthèse

Phase d'analyse



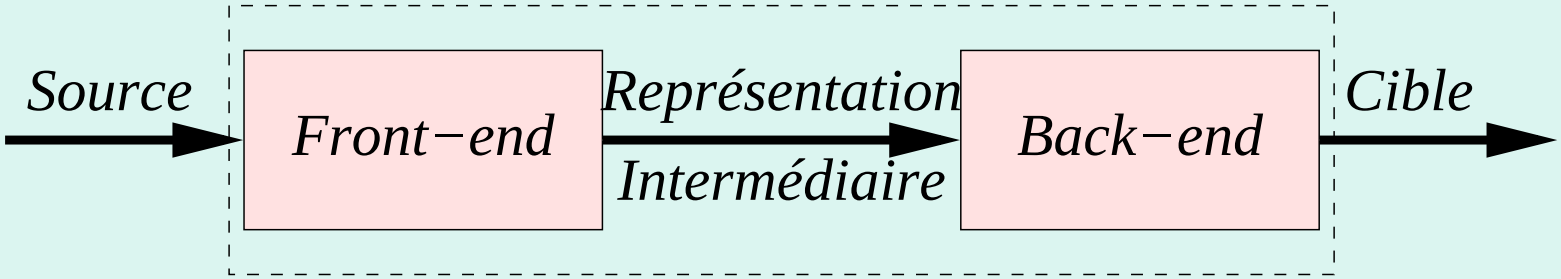
- **Analyse lexicale** : lire programme source caractère par caractère, grouper en **symboles**, éliminer blancs/commentaires, produire listing
- **Analyse syntaxique** : vérifier syntaxe, construire la représentation intermédiaire (IR), construire table de symboles
- **Analyse sémantique** : déduire/vérifier les types, annoter la représentation intermédiaire, effectuer autres vérifications sémantiques

Phase de synthèse



- **Simplification** : transformer les formes plus complexes du langage en formes plus simples (e.g. boucle `for` en boucle `while`)
- **Optimisation** : transformer le programme pour améliorer sa vitesse d'exécution ou l'espace requis (e.g. remplacer `x*8` par `x<<3`)
- **Génération de code** : produire le code dans le langage cible

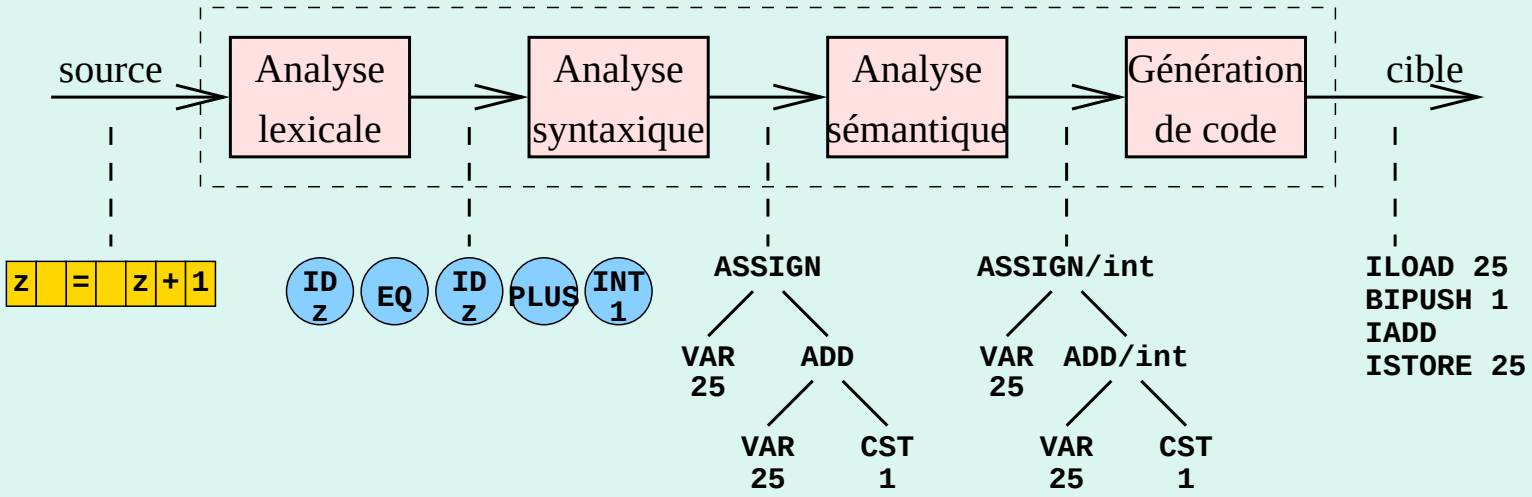
Front-end et back-end



- On décompose parfois la structure du compilateur en un **“front-end”** et un **“back-end”**
- **Front-end** : partie qui est **indépendante** du Lc
- **Back-end** : partie qui est **dépendante** du Lc
- Attrait : **recibler** le compilateur en changeant seulement le back-end
- Si on a M langages source et N langages cible : $M + N$ composantes à développer plutôt que $M * N$ compilateurs **(idée jamais réalisée en pratique!)**

Compilateur = pipeline

- Compilateur = pipeline de petits compilateurs

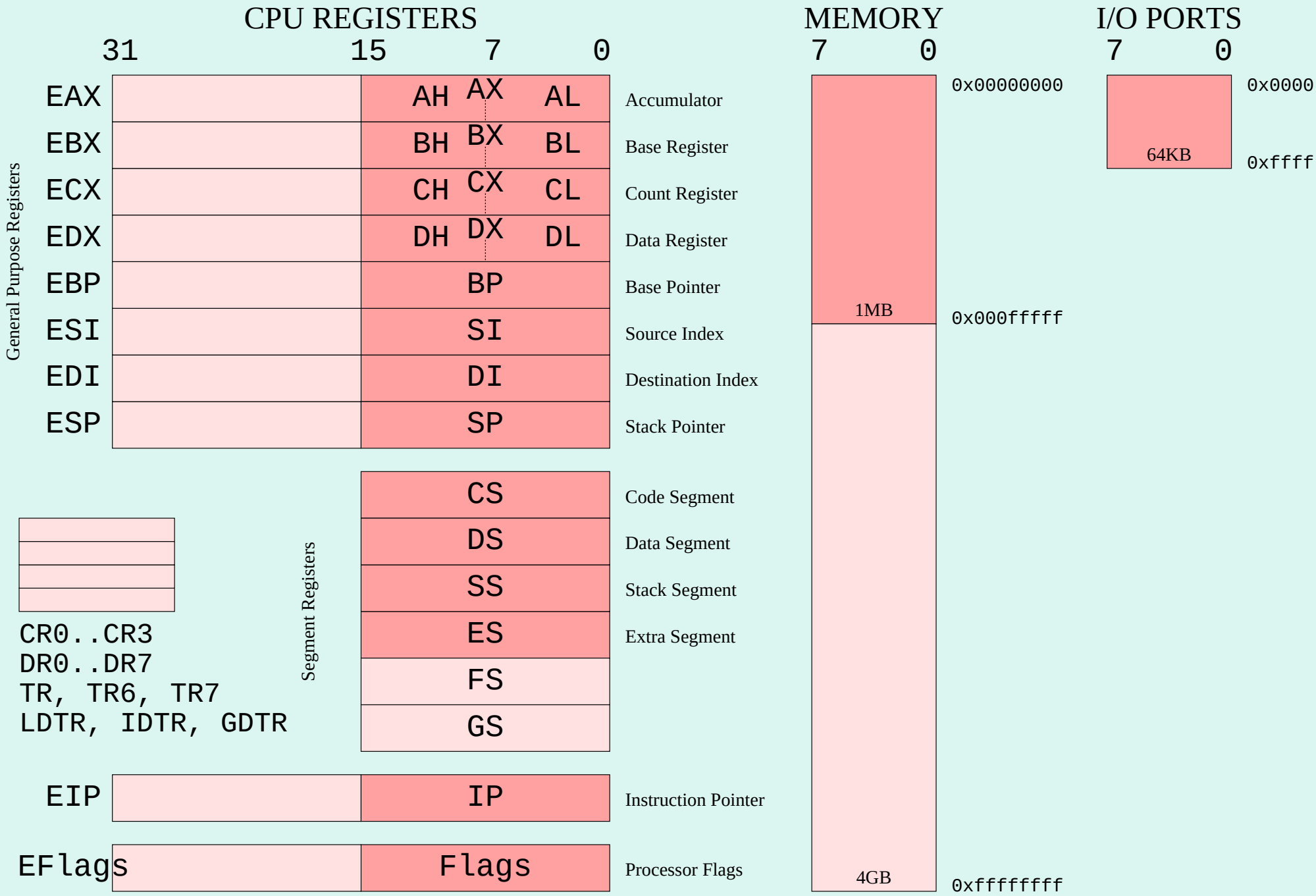


- Tout étage est lui-même un petit compilateur (avec ses propres **langage source** et **langage cible**)
- Jadis les étages étaient des programmes indépendants
  - Exemple : **cc** = **cpp** + **ccom** + **as**
  - Cela réduit la mémoire requise à la compilation
- Maintenant ces étages sont souvent **concurrents**

Machines cibles

- **Machines réelles** (en matériel): x86-32, x86-64, ARM, Thumb, SPARC, PowerPC, MIPS, M68K, ...
  - Grande vitesse d’exécution
- **Machines virtuelles** (en logiciel): JVM, P-code, Lua-VM, Parrot, LLVM, ...
  - Code compact
  - Simplicité de la compilation
  - Portabilité grâce à un interprète portable

Architecture Intel 8086/80386



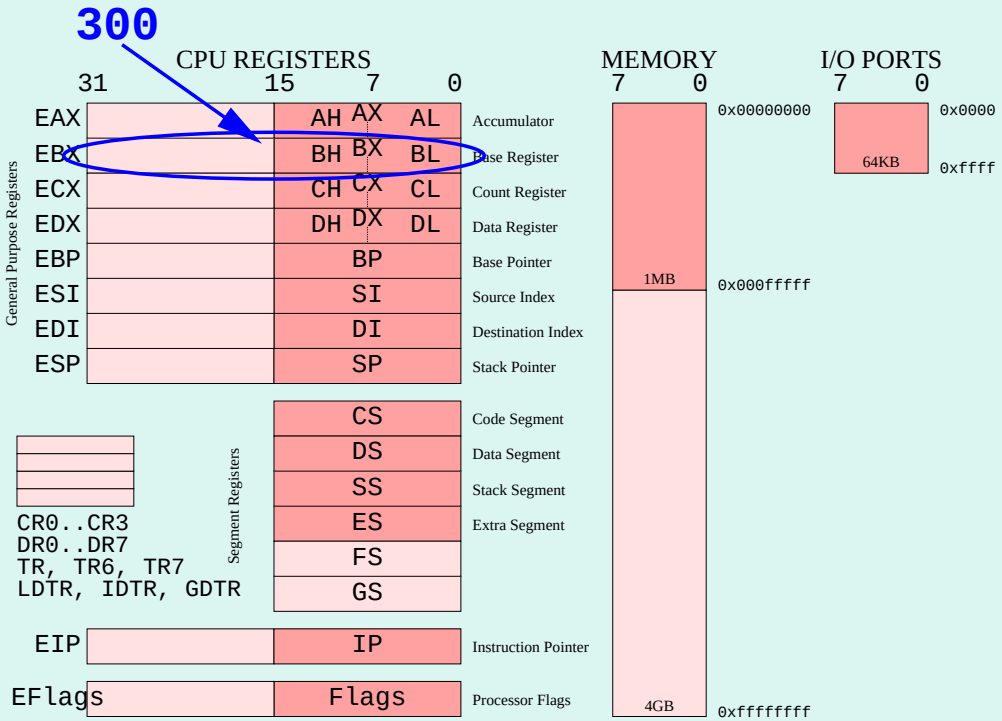
Modes d'adressage de la famille x86

- **Registre**: contenu d'un registre du processeur
  - **Syntaxe:** *%nom*
  - **Exemples:** `%eax  %cl  %es`
- **Immédiat**: valeur numérique constante
  - **Syntaxe:** $*expression_constante*
  - **Exemples:** `$10  $prix  $1000+2*5  $0x20`
- **Direct**: contenu d'une case mémoire à une adresse fixe
  - **Syntaxe:** *expression_constante*
  - **Exemples:** `10  prix  1000+2*5  0x20`
- **Indirect**: contenu d'une case mémoire à une adresse calculée
  - **Syntaxe:** *expr_constante*`(`*reg_base*`,`*reg_index*`,`*1*`|`*2*`|`*4*`|`*8*`)`
  - **Exemples:** `4(%ebp,%eax,8)  9(%esp)  (%di)`

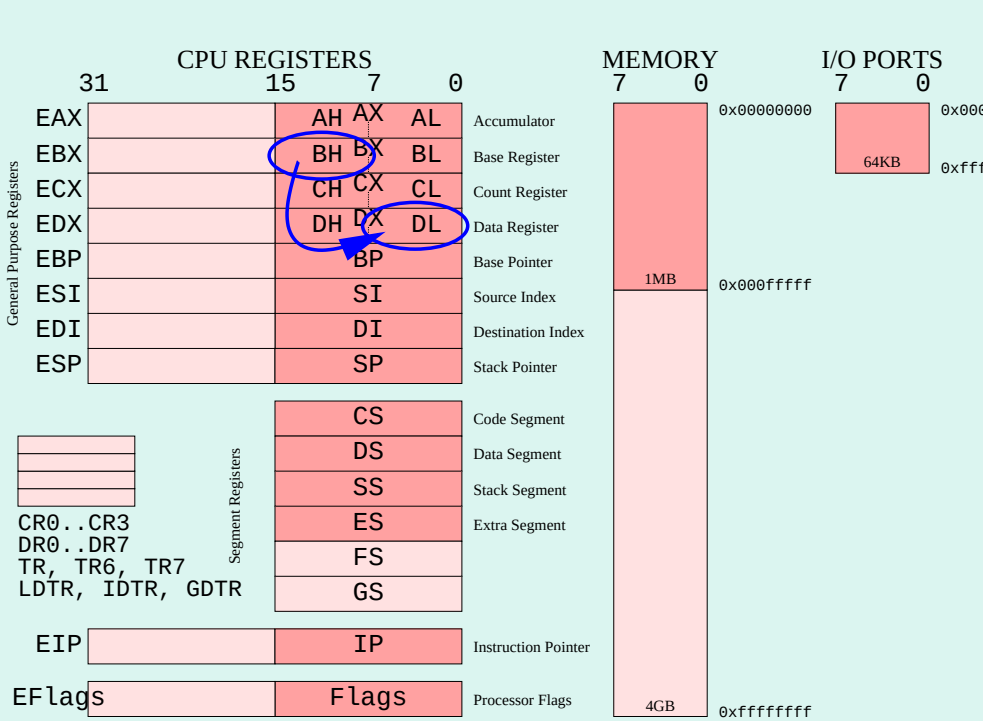
Modes immédiat et registre

- Exemples avec instruction: `mov` *source*, *dest*

`mov $300,%ebx`

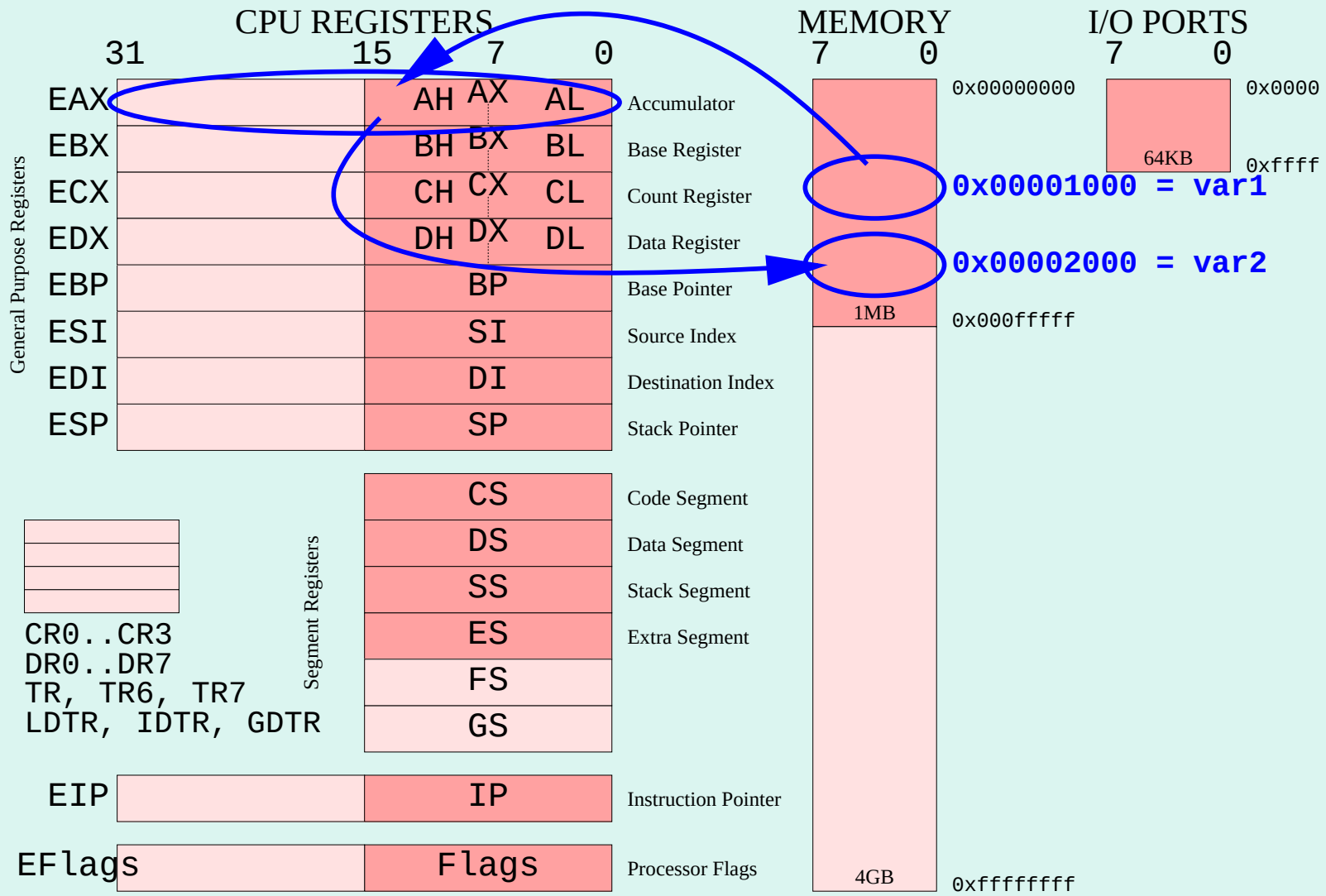


`mov %bh,%dl`



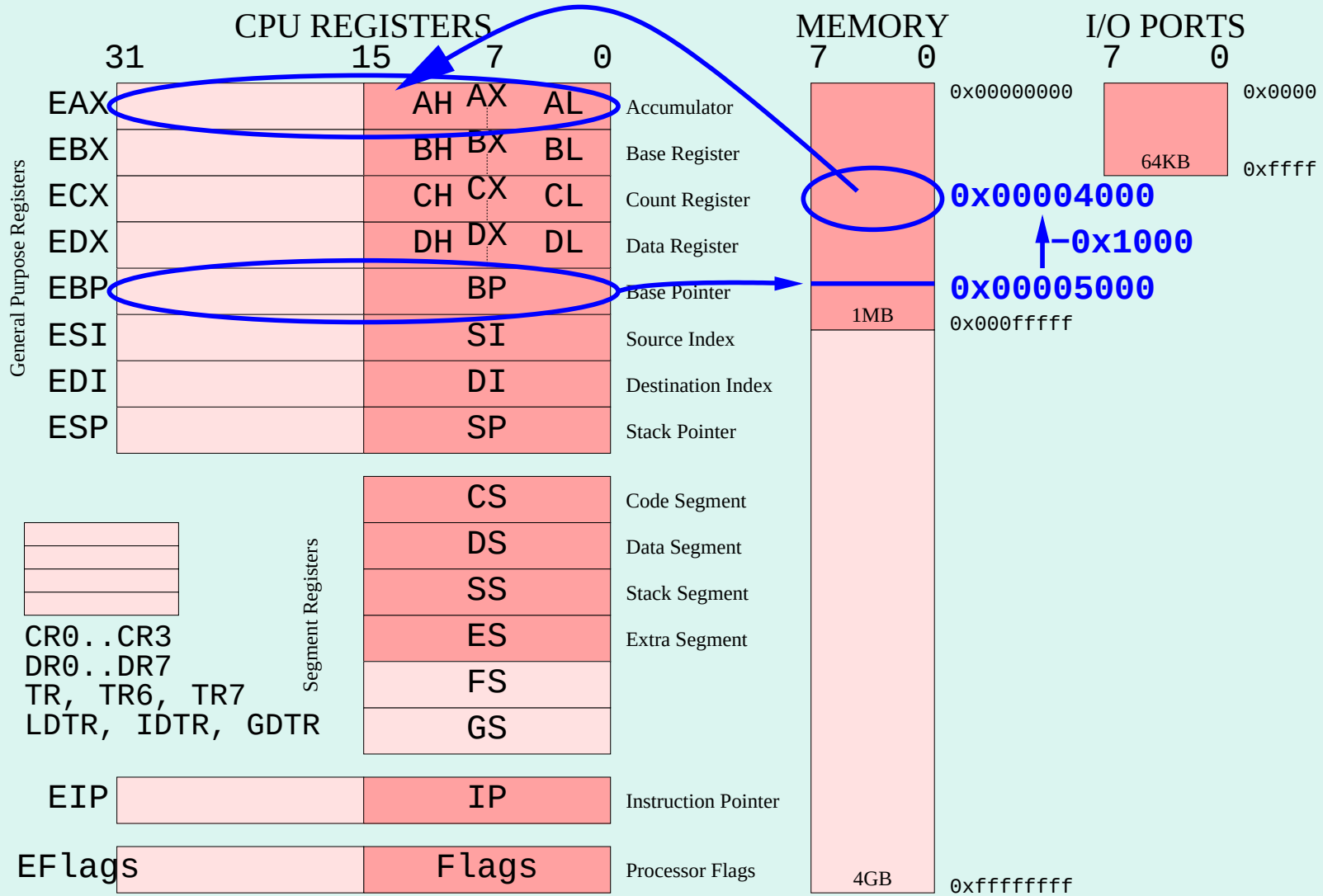
Mode direct


```
mov var1,%eax
mov %eax,var2
```



Mode indirect

`mov -0x1000(%ebp),%eax`



Instructions principales (1)

`mov` *source*, *dest*	*dest* ← *source*
`add` *source*, *dest*	*dest* ← *dest* + *source*
`sub` *source*, *dest*	*dest* ← *dest* - *source*
`inc` *dest*	*dest* ← *dest* + 1
`dec` *dest*	*dest* ← *dest* - 1
`neg` *dest*	*dest* ← - *dest*
`and` *source*, *dest*	*dest* ← *dest* & *source*
`or` *source*, *dest*	*dest* ← *dest* | *source*
`xor` *source*, *dest*	*dest* ← *dest* ^ *source*
`not` *dest*	*dest* ← ~ *dest*

- Pour forcer une opération d'une certaine taille : ajouter suffixe (`b` = 8 bits, `w` = 16 bits, `l` = 32 bits)

```
movb   $10,20(%ebp)
```

Instructions principales (2)

`pushl` *source*	`sub $4,%esp` ; `movl` *source*`,(%esp)`
`popl` *dest*	`movl (%esp),`*dest* ; `add $4,%esp`
`jmp` *étiquette*	`mov $`*étiquette*`,%ip`
`jz` *étiquette*	`mov $`*étiquette*`,%ip` (si `flags.Z = 1`)
`jnz` *étiquette*	`mov $`*étiquette*`,%ip` (si `flags.Z = 0`)
`cmp` *source*, *dest*	*dest* - *source* (`flags` ← `codes`)
`call` *étiquette*	`push %ip` ; `mov $`*étiquette*`,%ip`
`ret`	`pop %ip`

Exemple 1

- Somme des nombres de 0 à 10

```
1. [b90a000000]    mov   $10,%ecx
2. [b800000000]    mov   $0,%eax
3. [           ] loop:
4. [01c8       ]   add   %ecx,%eax
5. [49         ]   dec   %ecx
6. [75fb       ]   jnz   loop
```

Exemple 2

- Fonction “max” et appel

```
1.  [        ] max:                          |           |
2.  [89e5    ]   mov   %esp,%ebp             +-----------+
3.  [8b4608  ]   mov   8(%ebp),%eax          | adr ret |<--%esp
4.  [8b5e04  ]   mov   4(%ebp),%ebx          +-----------+
5.  [39d8    ]   cmp   %ebx,%eax             |    22     |
6.  [7702    ]   ja    eax_is_max            +-----------+
7.  [89d8    ]   mov   %ebx,%eax             |    11     |
8.  [        ] eax_is_max:                   +-----------+
9.  [c3      ]   ret                         |           |  memoire
10.                                          <-32 bits->  haute
11.
12.
13.
14. [6a0b      ]   pushl $11   <-- appel de max(11,22)
15. [6a16      ]   pushl $22
16. [e8xxxxxxxx]   call  max
17. [83c408    ]   add   $8,%esp
```

Etudier gcc

- Nous pouvons en apprendre beaucoup sur le fonctionnement d'un compilateur en étudiant le **comportement d'un compilateur existant**
- Nous allons brièvement étudier le comportement du **compilateur C de GNU (gcc)** pour architecture Intel x86-32
- Nous allons ensuite concevoir un petit compilateur pour **reproduire son comportement**
- En d'autres termes... du **"reverse engineering"** de compilateur!

Etudier gcc (2)

- Programme source (**test1.c**) :

```
/* File: "test1.c" */

int main( int argc, char *argv[] )
{
  return 20;
}
```

- `% gcc -O1 -fomit-frame-pointer -S test1.c`

- Crée le programme cible **test1.s**

Etudier gcc (3)

- Programme cible (**test1.s**) :

```
        .text
.globl _main
_main:
        movl    $20, %eax
        ret
```

- Assemblage, édition de liens et exécution :

```
% gcc -o test1.exe test1.s
% ./test1.exe
% echo $?
20
```

Etudier gcc (4)

- Constatations :

1. L'étiquette globale **"_main"** correspond à la fonction **"main"** (c'est le "name mangling")

2. L'instruction **"movl $20,** *reg***"** stocke la valeur 20 dans le registre *reg*

3. Le protocole d'appel de fonction utilise le registre **"%eax"** pour stocker le résultat d'une fonction ayant un résultat de type "int" (probablement int = 32 bits)

4. Le protocole d'appel de fonction doit se servir d'un "call" pour appeler "main", i.e. **"call _main"**

Etudier gcc (5)

- “Tout projet même trivial devrait avoir un makefile”

```
# File: makefile

.SUFFIXES:
.SUFFIXES: .o .c .s .exe

GCC_OPTS= -O1 -fomit-frame-pointer

all: test1.s test1.exe

.c.s:
        gcc $(GCC_OPTS) -S -o $*.s $*.c

.s.exe:
        gcc -o $*.exe $*.s

clean:
        rm -f *.s *.exe
```

- Maintenant :

```
% make
gcc -O1 -fomit-frame-pointer -S -o test1.s test1.c
gcc -o test1.exe test1.s
```

Etudier gcc (6)

- “Tout projet même trivial devrait être sous git/cvs”

```
% mkdir monprojet

% cd monprojet

% git init
Initialized empty Git repository in /u/feeley/monpr

... creer les fichiers test1.c et makefile ...

% git add makefile test1.c

% git commit -a -m "premier commit"
[master (root-commit) c96816a] premier commit
 2 files changed, 23 insertions(+), 0 deletions(-)
 create mode 100644 makefile
 create mode 100644 test1.c

... modifier test1.c ...

% git commit -a -m "retourner 99"
[master f2b7f30] retourner 99
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Etudier gcc (7)


```
% git log
commit f2b7f30db7b1c70fdec52a04d380dbd1d8448bcb
Author: Marc Feeley <feeley@iro.umontreal.ca>
Date:   Thu Jan 5 10:11:42 2012 -0500

    retourner 99

commit c96816a9d2c1eff5f1a0b601493c5382208984f9
Author: Marc Feeley <feeley@iro.umontreal.ca>
Date:   Thu Jan 5 10:10:16 2012 -0500

    premier commit

% git diff f2b7 c968
diff --git a/test1.c b/test1.c
index 9088c34..4c627af 100644
--- a/test1.c
+++ b/test1.c
@@ -2,5 +2,5 @@

 int main( int argc, char *argv[] )
 {
-  return 99;
+  return 20;
 }
```

Etudier gcc (8)

- Programme source (**test2.c**) :

```
/* File: "test2.c" */

int main( int argc, char *argv[] )
{
  return argc;
}
```

- Programme cible (**test2.s**) :

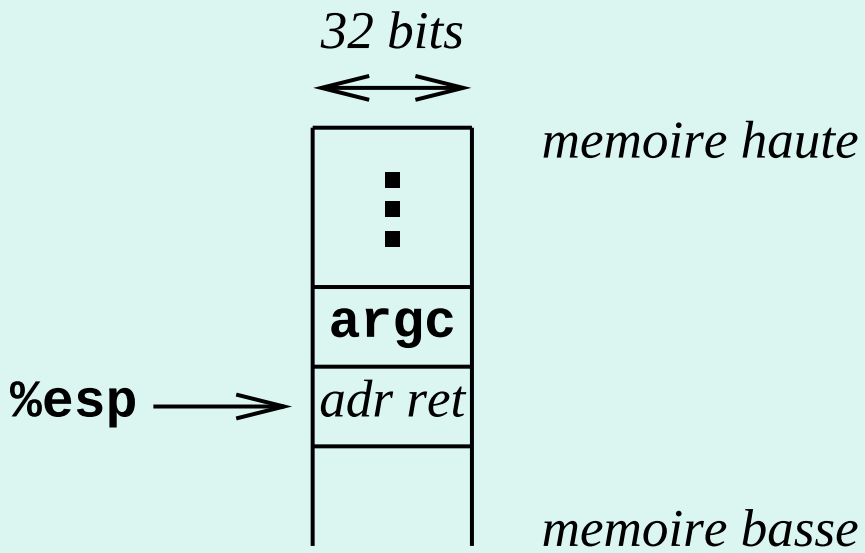
```
        .text
.globl _main
_main:
        movl    4(%esp), %eax
        ret
```

```
% ./test2.exe a b
% echo $?
3
```

Etudier gcc (9)

- Constatations :

1. Le paramêtre “argc” est sur la pile d’exécution juste au dessus de l’adresse de retour empilée par l’instruction **“call _main”** :



2. Les paramêtres sont empilés **du dernier au premier**, et la pile **grandit vers le bas**

Etudier gcc (10)

- Conception d'un compilateur simple pour expressions numériques littérales (fichier avec extension ".x")

```
# File: makefile

.SUFFIXES:
.SUFFIXES: .o .c .s .x .exe

GCC_OPTS= -O1 -fomit-frame-pointer

all: test3.s test3.exe

.c.s:
        gcc $(GCC_OPTS) -S -o $*.s $*.c

.x.s:
        gsi comp.scm < $*.x > $*.s

.s.exe:
        gcc -o $*.exe $*.s

clean:
        rm -f *.s *.exe
```

Etudier gcc (11)

- Programme source (**test3.x**) :

```
25
```

- Programme cible (**test3.s**) :

```
        .text
.globl _main
_main:
        movl    $25, %eax
        ret
```

comp.scm


```
;;; Parsing

(define (parse)
  (read))

;;; Translation of AST to machine code

(define (translate ast)
  (comp-function "_main" ast))

(define (comp-function name expr)
  (gen-function name (comp-expr expr)))

(define (comp-expr expr) ;; generate code to compute expr in %eax
  (cond ((number? expr)
         (gen-literal expr))
        (else
         (error "comp-expr can only compile literal expressions"))))

;;; Code generation for x86-32 using GNU as syntax

(define gen string-append)

(define (gen-function name code)
  (gen "    .text\n"
       ".globl " name "\n"
       name ":\n"
       code
       "    ret\n"))

(define (gen-literal n)
  (gen "    movl    $" (number->string n) ", %eax\n"))

(display (translate (parse)))
```

Expressions binaires

- Étendre le compilateur pour accepter les **opérateurs binaires**: +, -, *, /

- Pour simplifier utiliser la **syntaxe de Scheme** pour les expressions, i.e.

$$20/6 \Longrightarrow (/\ 20\ 6)$$

- Chaque (sous-)expression est calculée dans le registre **“%eax”**

- Utiliser la **pile d’exécution** pour sauver les résultats intermédiaires

Exemple

- Programme source (**test4.x**) :

```
(/ 20 6)
```

- Programme cible (**test4.s**) :

```
          .text
.globl _main
_main:
          movl    $6, %eax
          addl    $-4, %esp
          movl    %eax, (%esp)
          movl    $20, %eax
          idivl   (%esp), %eax
          addl    $4, %esp
          ret
```

Changements à comp.scm


```
(define (comp-expr expr) ;; generate code to compute expr in %eax

  (cond ((number? expr)
         (gen-literal expr))

        ((and (list? expr)
              (= (length expr) 3)
              (member (list-ref expr 0) '(+ - * /)))
         (gen-bin-op
          (case (list-ref expr 0)
            ((+) "add")
            ((-) "sub")
            ((*) "imul")
            ((/) "idiv"))
          (comp-expr (list-ref expr 2))
          (comp-expr (list-ref expr 1))))

        (else
         (error "comp-expr cannot handle expression"))))

(define (gen-bin-op oper opnd1 opnd2)
  (gen opnd1
       "    addl    $-4, %esp\n"
       "    movl    %eax, (%esp)\n"
       opnd2
       "    " oper "l    (%esp), %eax\n"
       "    addl    $4, %esp\n"))
```

Paramètre argc

- Étendre le compilateur pour permettre d’accéder à **argc**

Changements incorrects à comp.scm


```
(define (comp-expr expr) ;; generate code to compute expr in %eax

  (cond ((number? expr)
         (gen-literal expr))

        ((equal? expr 'argc)
         (gen-parameter 1))

        ...))

(define (gen-parameter i)
  (gen "    movl    " (number->string (* 4 i)) "(%esp), %eax\n"))
```

Exemple

- Programme source (**test4.x**) :

```
(+ argc argc)
```

- Programme cible (**test4.s**) :

```
        .text
.globl _main
_main:
        movl    4(%esp), %eax
        addl    $-4, %esp
        movl    %eax, (%esp)
        movl    4(%esp), %eax
        addl    (%esp), %eax
        addl    $4, %esp
        ret
```

- Il faut tenir compte des changements à **“%esp”** (le **“frame size”**)

Changements corrects à comp.scm


```
(define (comp-function name expr)
  (gen-function name (comp-expr expr 0)))

(define (comp-expr expr fs)

  (cond ((number? expr)
         (gen-literal expr))

        ((equal? expr 'argc)
         (gen-parameter (+ fs 1)))

        ((and (list? expr)
              (= (length expr) 3)
              (member (list-ref expr 0) '(+ - * /)))
         (gen-bin-op
          (case (list-ref expr 0)
            ((+) "add")
            ((-) "sub")
            ((*) "imul")
            ((/) "idiv"))
          (comp-expr (list-ref expr 2) fs)
          (comp-expr (list-ref expr 1) (+ fs 1))))

        (else
         (error "comp-expr cannot handle expression"))))
```

Code cible correct

- Programme cible (**test4.s**) :

```
        .text
.globl _main
_main:
        movl    4(%esp), %eax
        addl    $-4, %esp
        movl    %eax, (%esp)
        movl    8(%esp), %eax
        addl    (%esp), %eax
        addl    $4, %esp
        ret
```

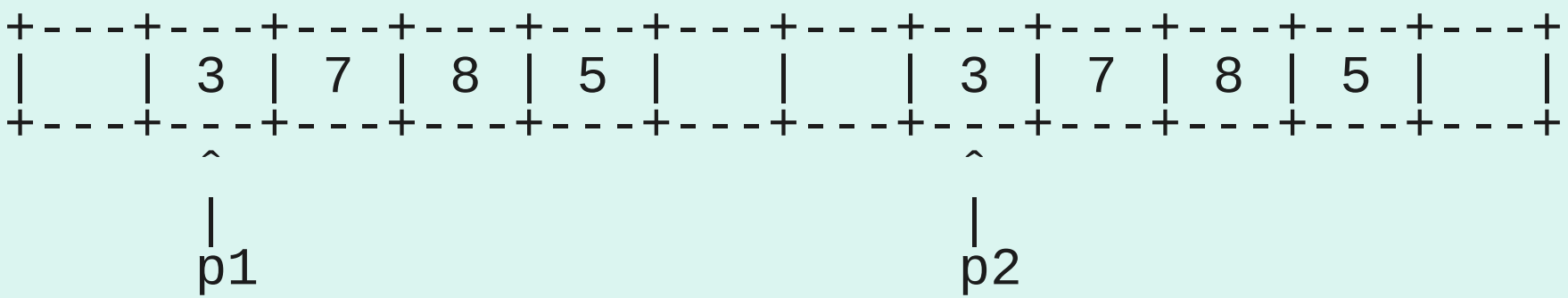
- Plusieurs améliorations sont possibles :
  - Utiliser instructions “pushl” et “popl”
  - Allocation de registre
  - “Constant folding”
  - “Strength reduction” (incr, shift)

Exemple complexe (1)

- Copier une zone de mémoire de n entiers de 32 bits

- Code possible en C :

```
1. typedef int i32;
2.
3. void copy( i32 *p1, i32 *p2, int n )
4. {
5.   do {
6.        *p2++ = *p1++;
7.      } while (--n != 0);
8. }
```



Exemple complexe (2)


```
1.                .text
2.             .globl _copy
3.             _copy:
4.  [55    ]       pushl %ebp                  |          |
5.  [89e5  ]       movl  %esp, %ebp          +----------+
6.  [50    ]       pushl %eax                |  %edi'   | <-- %esp
7.  [51    ]       pushl %ecx                +----------+
8.  [56    ]       pushl %esi                |  %esi'   |
9.  [57    ]       pushl %edi                +----------+
10. [8b7508]       movl  8(%ebp), %esi       |  %ecx'   |
11. [8b7d0c]       movl  12(%ebp), %edi      +----------+
12. [8b4d10]       movl  16(%ebp), %ecx      |  %eax'   |
13.            loop:                         +----------+
14. [8b06  ]       movl  (%esi), %eax        |  %ebp'   | <-- %ebp
15. [8907  ]       movl  %eax, (%edi)        +----------+
16. [83c604]       addl  $4, %esi            | adr ret  |
17. [83c704]       addl  $4, %edi            +----------+
18. [49    ]       dec   %ecx                |    p1    |
19. [75f3  ]       jnz   loop                +----------+
20. [5f    ]       popl  %edi                |    p2    |
21. [5e    ]       popl  %esi                +----------+
22. [59    ]       popl  %ecx                |    n     |
23. [58    ]       popl  %eax                +----------+
24. [c9    ]       leave                     |          |  memoire
25. [c3    ]       ret                      <-32 bits->  haute
```

Définition formelle de la syntaxe (1)

- Méthode classique = définir une **grammaire** pour le langage
- La grammaire spécifie exactement les programmes (ou “phrases”) qui sont **acceptables syntaxiquement**
- Analogie avec langue naturelle :

```
ce chat est noir        OK
est noir chat ce        pas OK

x := 0 ;                OK
0 ; := x                pas OK
```

- Note: un programme est composé d’une séquence de **symboles** (identificateurs, mot réservés, ponctuation, etc)

Définition formelle de la syntaxe (2)

- Déf: **Vocabulaire** (Σ) = ensemble de symboles qu'on peut retrouver dans une phrase

  Ex: Σ={0,1} Σ={+,-,a,...,z} Σ={if,and,:=,<,...}

- Déf: **Phrase** = séquence possiblement vide de symboles tirés de Σ

  Ex: Σ={0,1} phrase1=00101 phrase2= (vide)

- Déf: **Langage** = ensemble de phrases, possiblement infini

  Ex1: Σ={0,1} L1={00,01,10,11}
  Ex2: Σ={0,1} L2={1,10,100,1000,...}

Définition formelle de la syntaxe (3)

- Déf: **Grammaire hors-contexte** en format BNF (Backus-Naur Form) = ensemble de **catégories** et **productions**
  - **Catégorie** = nom qui désigne un type de fragment de phrase, par convention entouré de $<$ ... $>$

    Ex: $<$*expression*$>$ $<$*entier*$>$ $<$*vide*$>$
  - **Production** = règle de la forme

    $<$*cat*$>$ `::=` x_1 x_2 ... x_n

    où $<$*cat*$>$ est une catégorie et x_i est une catégorie ou symbole pour tout i

    Ex: $<$*X*$>$ `::=` `0` $<$*Y*$>$ `1`

Définition formelle de la syntaxe (4)

- Par convention la première production indique la **catégorie de départ**

- Exemple: grammaire G1 =

  ⟨*bin*⟩ ::= 0
  ⟨*bin*⟩ ::= 1
  ⟨*bin*⟩ ::= ⟨*bin*⟩ ⟨*bin*⟩

Définition formelle de la syntaxe (5)

- Déf: **Dérivation directe** (notation $X \rightarrow Y$)
  Soit $x_1 \ldots x_n$ une chaîne et
  1. x_i est une catégorie ou symbole pour tout i
  2. x_j = $<C>$
  3. il existe une production $<C> ::= y_1 \ldots y_m$

  alors

  $x_1 \ldots x_{j-1} <C> x_{j+1} \ldots x_n \rightarrow x_1 \ldots x_{j-1} y_1 \ldots y_m x_{j+1} \ldots x_n$

- Déf: **Dérivation** (d'une phrase) = une séquence de dérivations directes à partir de la catégorie de départ dont le résultat est la phrase en question
  Ex: $<bin> \rightarrow <bin> <bin> \rightarrow <bin>$ `0` $\rightarrow$ `1 0`

Définition formelle de la syntaxe (6)

- Déf: **L(G)** = le langage défini par la grammaire G = ensemble de toutes les phrases dérivables par G

- L(G) = { p | $<\textit{départ}> \rightarrow^{*}$ p }

- Par exemple: L(G1) = { `0`, `1`, `00`, `01`, `10`, `11`, ... } car

  $<\textit{bin}> \rightarrow$ `0`

  $<\textit{bin}> \rightarrow$ `1`

  $<\textit{bin}> \rightarrow <\textit{bin}>\,<\textit{bin}> \rightarrow <\textit{bin}>$ `0` $\rightarrow$ `0 0`

  $<\textit{bin}> \rightarrow <\textit{bin}>\,<\textit{bin}> \rightarrow <\textit{bin}>$ `1` $\rightarrow$ `0 1`

  $<\textit{bin}> \rightarrow <\textit{bin}>\,<\textit{bin}> \rightarrow <\textit{bin}>$ `0` $\rightarrow$ `1 0`

  etc

Définition formelle de la syntaxe (7)

- Exemple: grammaire G2 =

<*flottant*> ::= <*entier*> . <*entier*>

<*entier*> ::= <*chiffre*>
<*entier*> ::= <*chiffre*> <*entier*>

<*chiffre*> ::= 0
<*chiffre*> ::= 1
<*chiffre*> ::= 2
<*chiffre*> ::= 3
<*chiffre*> ::= 4
<*chiffre*> ::= 5
<*chiffre*> ::= 6
<*chiffre*> ::= 7
<*chiffre*> ::= 8
<*chiffre*> ::= 9

Définition formelle de la syntaxe (8)

- Rappel: grammaire G2 =

  <flottant> ::= *<entier>* . *<entier>*

  <entier> ::= *<chiffre>*
  <entier> ::= *<chiffre>* *<entier>*

  <chiffre> ::= 0
  ...

- 1.5 dans L(G2)? oui:

  <flottant> → *<entier>* . *<entier>* → *<chiffre>* . *<entier>*

  → 1 . *<entier>* → 1 . *<chiffre>* → 1 . 5

  ou bien

  <flottant> → *<entier>* . *<entier>* → *<entier>* . *<chiffre>*
  → *<chiffre>* . *<chiffre>* → *<chiffre>* . 5 → 1 . 5

- .5 dans L(G2)? non

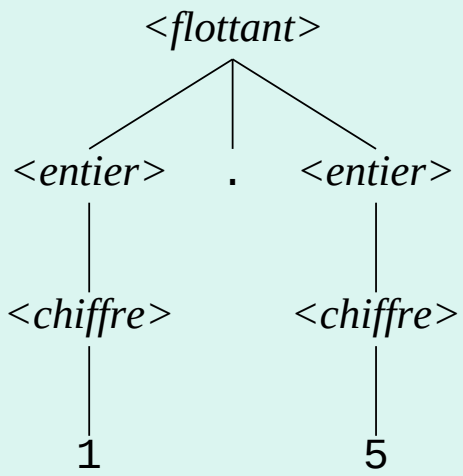
Arbre de dérivation (1)

- Utile pour représenter la **structure syntaxique** d'une phrase
- Étant donné une certaine dérivation d'une phrase P, l'arbre de dérivation correspondant aura la catégorie de départ à sa racine, des symboles aux feuilles, et chaque noeud interne est une catégorie qui a comme enfants la partie droite de la production qui l'a remplacée dans la dérivation
- Les feuilles de l'arbre de dérivation = la phrase
- L'arbre de dérivation **ne représente pas l'ordre** dans lequel les dérivations sont effectuées

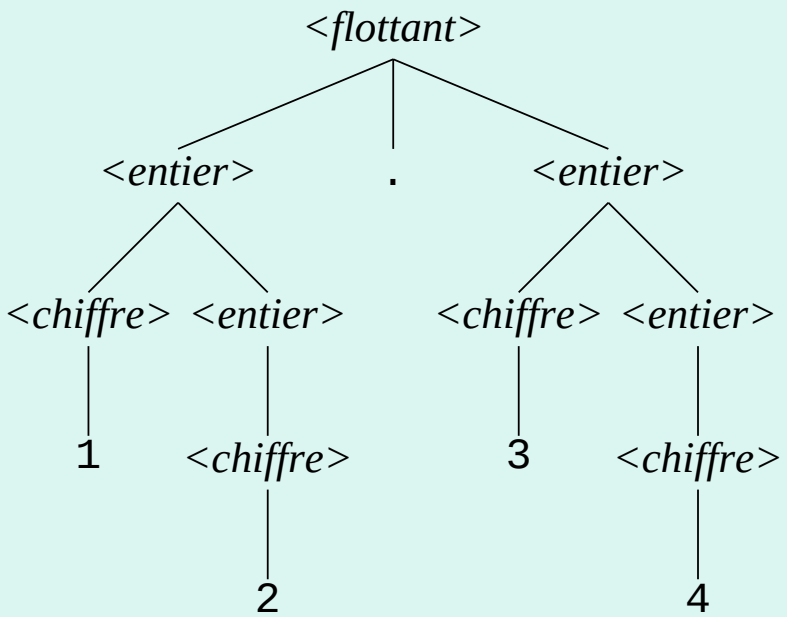
Arbre de dérivation (2)

- Exemples avec grammaire G2

`1.5`



`12.34`

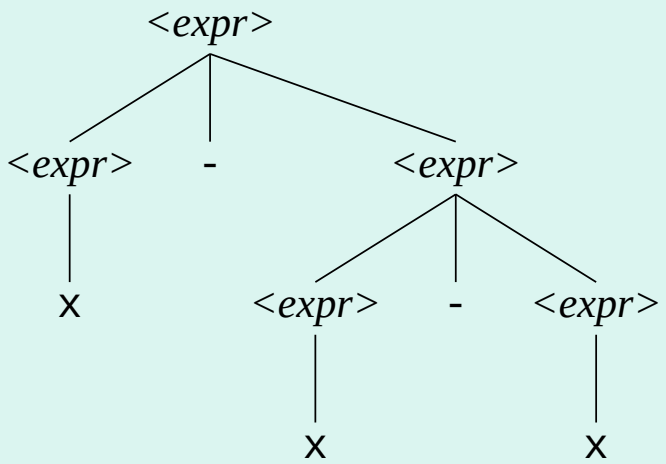


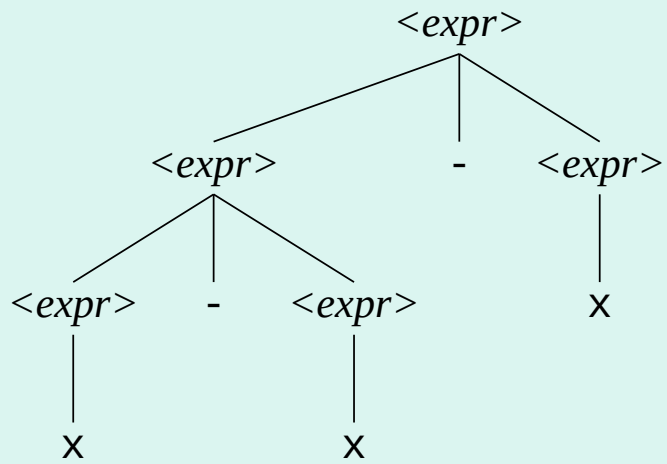
<flottant> → <entier> . <entier> → <chiffre> . <entier>
→ 1 . <entier> → 1 . <chiffre> → 1 . 5

<flottant> → <entier> . <entier> → <entier> . <chiffre>
→ <chiffre> . <chiffre> → <chiffre> . 5 → 1 . 5

Grammaires ambiguës (1)

- Déf: Une grammaire G est **ambiguë** ssi il existe une phrase P dans L(G) tel que P a **plus qu'un arbre de dérivation**
- Exemple de grammaire ambiguë: G3 =
  <expr> ::= x
  <expr> ::= *<expr>* - *<expr>*
- 2 arbres de dérivation pour: x - x - x





Grammaires ambiguës (2)

- Les grammaires ambiguës sont à éviter car normalement le compilateur utilise l'arbre de dérivation pour **attacher un sens au programme**

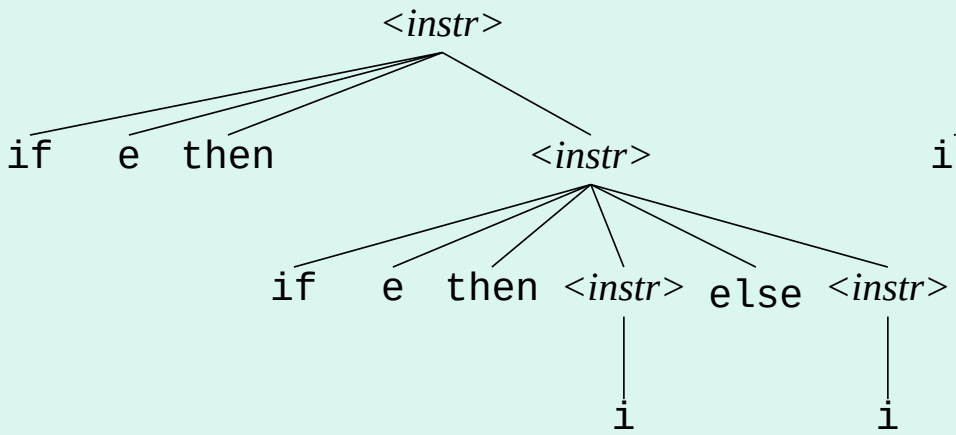
- Si la grammaire est ambiguë, le compilateur doit utiliser d'autres règles pour retirer les ambiguïtés

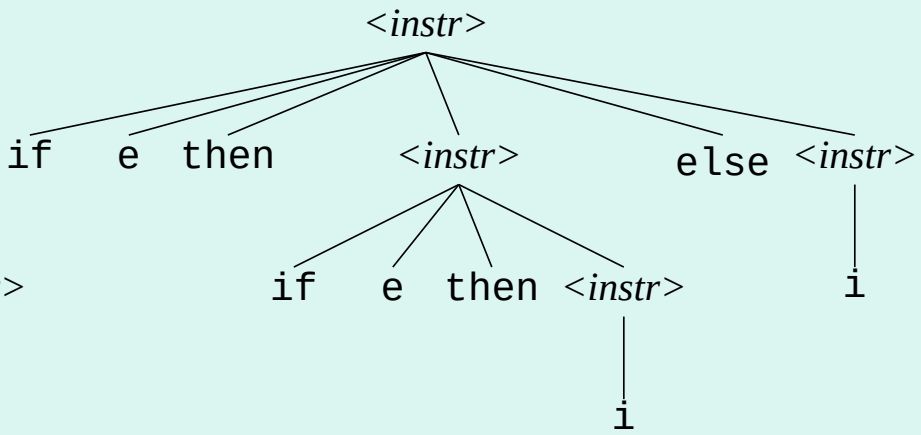
Grammaires ambiguës (3)

- Cas classique, le **“else” pendant**: gram. G4 =

  <*instr*> ::= i
  <*instr*> ::= if e then <*instr*>
  <*instr*> ::= if e then <*instr*> else <*instr*>

  if e then if e then i else i





Grammaires ambiguës (4)

- Solution typique (Algol60/Pascal/C): règle informelle “`else` va avec le `if` le plus proche”

- Algol68/Modula-2/Ada/Fortran77 évitent ce problème en demandant un `fi`/`END`/`end if`/`endif` à la fin d’un `if`

- Grammaire G5 =

<*instr*> `::=` `i`

<*instr*> `::=` `if e then` <*instr*> `fi`
<*instr*> `::=` `if e then` <*instr*> `else` <*instr*> `fi`

```
if e then if e then i else i fi fi
if e then if e then i fi else i fi
```

Autres formalismes syntaxiques (1)

- Déf: **BNF étendu (EBNF)** = BNF + expressions régulières
- Notation permise dans les productions :
  - `[ X ]` $->$ X est optionnel
  - `{ X }` $->$ répétition de X (≥ 0 fois)
  - `X | Y` $->$ choix entre X et Y
  - `( X Y Z )` $->$ groupement
- Exemple: gram. G6 =

<*flottant*> `::=` <*entier*> `.` { <*chiffre*> } [<*exp*>]

<*entier*> `::=` <*chiffre*> { <*chiffre*> }

<*chiffre*> `::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

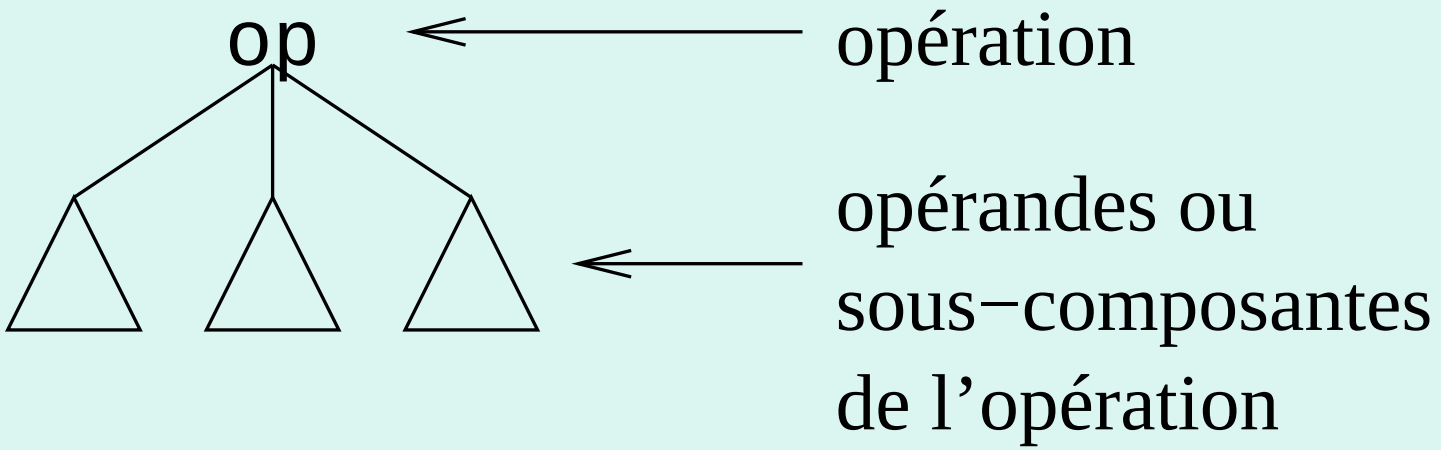
<*exp*> `::= E [ + | - ]` <*entier*>

ASA: Arbres de syntaxe abstraite (1)

- Déf: représentation d'un (fragment de) programme sous forme d'arbre qui **souligne sa structure** et **élimine les détails syntaxiques**

- En anglais **AST** ("Abstract Syntax Tree")

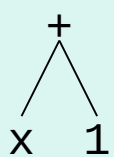
- Forme générale :



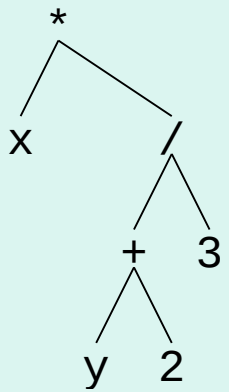
ASA: Arbres de syntaxe abstraite (2)

- Exemples tirés de Pascal :

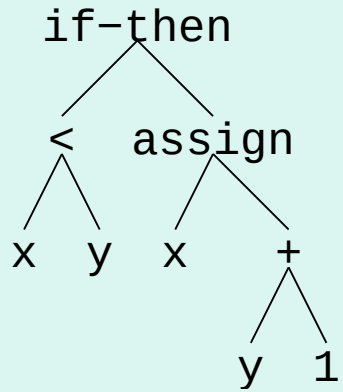
`x+1`



`x*((y+2)/3)`



`if x<y then x:=y+1`



- L'ASA ne contient pas les parenthèses, point-virgules, etc car il exprime implicitement les regroupements logiques qu'ils produisent
- Si deux fragments dans 2 langages différents ont le même ASA c'est qu'ils sont "équivalents", par exemple
  - C: `if (x<y) x=y+1;`
  - Scheme: `(if (< x y) (set! x (+ y 1)))`

Parseurs

- Un **parseur** pour un langage de programmation L est un algorithme qui lit un code source pour vérifier qu'il est dérivable par la grammaire de L
- Normalement le parseur **construit aussi l'arbre de dérivation ou de syntaxe** du code source et affiche des messages d'erreur de syntaxe
- Deux grandes classes d'algorithmes :
  - **Top-down parsing** : construction à partir de la racine de l'arbre de dérivation
    - Algorithmes : descente récursive, parseur LL, ...
  - **Bottom-up parsing** : construction à partir des feuilles de l'arbre de dérivation
    - Algorithmes : parseur CYK, parseur Earley, parseur SLR, parseur LALR, ...

Réalisation d'un parseur

- Deux approches :
  - Écrire le parseur à la main
  - Utiliser un **générateur de parseur** ("parser generator") pour le générer à partir de la grammaire

- Règle générale, on décompose la grammaire en deux :
  - **Grammaire lexicale** (vocabulaire = caractères)
  - **Grammaire syntaxique** (vocabulaire = "tokens")

Écrire un parseur à la main

- Laborieux mais offre plus de contrôle sur les actions du parseur

- Normalement on utilise la **descente récursive** :
  - requiert une grammaire de type **LL(1)** (non ambiguë, pas de récursions à gauche, c'est-à-dire de la forme $<seq> ::= \epsilon \mid <seq> <expr>$)
  - associer une fonction à chaque catégorie
  - utiliser le prochain symbole pour choisir la règle de production

- Certains langages se prêtent bien à cette approche (Lisp, Scheme, Pascal)

Exemple : expressions préfixes (1)

- GHC simplifiée en notation BNF :

```
<expr> ::= <simp> | <list>
<simp> ::= "x"
<list> ::= "(" <seq> ")"
<seq>  ::= ϵ | <expr> <seq>
```

Exemple : expressions préfixes (2)

- Parseur à descente récursive :

```
char next() { return ...next symbol... }
void advance() { ...skip next symbol... }

void expr() { if (next()=='x') simp(); else list(); }

void simp() { if (next()=='x') advance(); else error(); }

void list() { if (next()=='(') {
                 advance();
                 seq();
                 if (next()==')') advance(); else error();
               } else error();
             }

void seq() { if (next()=='(' || next()=='x') {
                expr();
                seq();
              }
            }
```

Grammaires LL(1)

- Les grammaires LL(1) ont besoin d'un seul symbole d'anticipation pour choisir la règle de production à utiliser

- Si on a la règle de production

  $<C>$ ::= α | β

  alors on aura la fonction de parseur pour $<C>$:

```
void C()
{
  if (next() in PREDICT(<C> ::= α))
    ... parser α ...;
  else if (next() in PREDICT(<C> ::= β))
    ... parser β ...;
  else
    /* impossible */;
}
```

Ensembles first et follow

- Soit
  - *first*(α) = { s | $\alpha \rightarrow^* s\beta$ }
  - *follow*(<C>) = { s | $\exists$ <X> ::= α et $\alpha \rightarrow^* \beta$ <C> $s\gamma$ }
  - *nullable*(α) ssi $\alpha \rightarrow^* \epsilon$

- Alors

  PREDICT(<C> ::= α) = *first*(α), si *nullable*(α) = faux

  PREDICT(<C> ::= α) = *first*(α) $\cup$ *follow*(<C>), sinon

- Si on a une règle de production : <C> ::= α | β

  et *PREDICT*(<C> ::= α) $\cap$ *PREDICT*(<C> ::= β) $\neq \emptyset$

  alors la grammaire n'est pas LL(1)

Calcul des ensembles first et follow


```
function nullable( X1 X2 ... Xk ) = nul[ X1 ] and ... and nul[ Xk ]

function first( X1 X2 ... Xk ) =
  if nul[ X1 ] = false then fir[ X1 ]
                       else fir[ X1 ] U first( X2 ... Xk )

function follow( <C> ) = fol[ <C> ]


for each symbol S     : nul[  S  ] = false ; fir[  S  ] = {S}
for each category <C> : nul[ <C> ] = false ; fir[ <C> ] = {} ; fol[ <C> ] = {}

repeat

  for each production  <C> ::= X1 ... Xk

    if nullable( X1 ... Xk ) then nul[ <C> ] = true

    for each i from 1 to k

      if nullable( X1 ... Xi-1 ) then fir[ <C> ] = fir[ <C> ] U fir[ Xi ]

      if nullable( Xi+1 ... Xk ) then fol[ Xi ] = fol[ Xi ] U fol[ <C> ]

      for each j from i+1 to k
        if nullable( Xi+1 ... Xj-1 ) then fol[ Xi ] = fol[ Xi ] U fir[ Xj ]

until nul, fir and fol did not change in this iteration
```

Exemple (1)


```
Grammaire :    <X> ::=  <Y>  |  a

               <Y> ::=       |  b

               <Z> ::=   c   |  <X> <Y> <Z>


Itérations de l'algorithme :

         ==========nul==========  ==========fir==========  =======fol=======
          <X>   <Y>   <Z>  a b c   <X>   <Y>   <Z>  a b c   <X>   <Y>   <Z>
init     false false false false                      a b c
iter 1   false true  false false    a     b     c    a b c   b,c    c
iter 2   true  true  false false   a,b    b   a,b,c  a b c  a,b,c a,b,c
iter 3   true  true  false false   a,b    b   a,b,c  a b c  a,b,c a,b,c

Donc :

PREDICT( <X> ::= <Y>          ) = {a,b,c}  <-- conflit
PREDICT( <X> ::= a            ) = {a}      <--'

PREDICT( <Y> ::=              ) = {a,b,c}  <-- conflit
PREDICT( <Y> ::= b            ) = {b}      <--'

PREDICT( <Z> ::= c            ) = {c}      <-- conflit
PREDICT( <Z> ::= <X> <Y> <Z>  ) = {a,b,c}  <--'

La grammaire n'est *pas* LL(1)
```

Exemple (2)


```
Grammaire :    <X> ::=          a

               <Y> ::=       |  b

               <Z> ::=   c   |  <X> <Y> <Z>


Itérations de l'algorithme :

        ==========nul==========  ==========fir==========  =======fol=======
         <X>   <Y>   <Z>  a b c   <X>   <Y>   <Z>  a b c   <X>   <Y>   <Z>
init    false false false false                      a b c
iter 1  false true  false false    a     b     c    a b c   b,c    c
iter 2  false true  false false    a     b    a,c   a b c   b,c    c
iter 3  false true  false false    a     b    a,c   a b c   b,c    c

Donc :

PREDICT( <X> ::= a           ) = {a}

PREDICT( <Y> ::=             ) = {c}
PREDICT( <Y> ::= b           ) = {b}

PREDICT( <Z> ::= c           ) = {c}
PREDICT( <Z> ::= <X> <Y> <Z> ) = {a}

La grammaire est LL(1)
```

Générateur de parseurs

- Un générateur de parseur est un compilateur dont le source est la spécification de la grammaire du langage et la cible est le parseur pour ce langage

- Plusieurs choix :
  - Grammaires régulières : lex, flex, JLex, Silex, ...
  - Grammaires hors-contexte : yacc, bison, ANTLR, LALR-SCM, ...

Exemple : expressions infixes

- GHC en notation BNF :

```
<expression> ::= <expr-add>

<expr-add> ::= <expr-mul>
             | <expr-add> "+" <expr-mul>
             | <expr-add> "-" <expr-mul>

<expr-mul> ::= <primary>
             | <expr-mul> "*" <primary>
             | <expr-mul> "/" <primary>

<primary> ::= <integer> | <ident> | "(" <expression> ")"

<integer> ::= <digit> | <integer> <digit>

<ident> ::= <letter> | <ident> ( <letter> | <digit> | "_" )

<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

<letter> ::= "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"
           | "k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"
           | "u"|"v"|"w"|"x"|"y"|"z"
```

Yacc

- “Yet Another Compiler Compiler”
- Requiert une grammaire de type **LALR(1)** (non ambiguë, ...)
- Fichier de spécification (p.e. “parseur.y”) :

```
... section déclarations ...

%%

... section règles de grammaire ...

%%

... section code C ...
```

```
% yacc parseur.y    # parseur.y ==> y.tab.c
% gcc -o parseur.exe y.tab.c
% ./parseur.exe
```

Fichier parseur.y (partie 1)


```
%{
#include <stdio.h>
#include <stdlib.h>

void yyerror( const char *msg )
{ printf( "error: %s\n", msg );
  exit( 1 );
}

int yylex( void )
{ int c = getchar();
  if (c == EOF || c == '\n') c = 0; /* YYEOF */
  return c;
}

%}

%start expression
```

Fichier parseur.y (partie 2)


```
%%

expression: expr_add ;

expr_add: expr_mul
        | expr_add '+' expr_mul
        | expr_add '-' expr_mul
        ;

expr_mul: primary
        | expr_mul '*' primary
        | expr_mul '/' primary
        ;

primary: integer | ident | '(' expression ')' ;

integer: digit | integer digit ;

ident: letter | ident letter | ident digit | ident '_' ;

digit: '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

letter: 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'
      | 'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'
      | 'u'|'v'|'w'|'x'|'y'|'z';

%%

int main( void )
{ yyparse(); return 0; }
```

Parseur en action


```
% yacc parseur.y    # parseur.y ==> y.tab.c
% gcc -o parseur.exe y.tab.c
% ./parseur.exe
12*(3+x-4)
% echo $?
0
% ./parseur.exe
12*-3
error: syntax error
% echo $?
1
```

Lex

- Pour alléger la grammaire on la sépare en une **grammaire lexicale** (qui définit les tokens) et une **grammaire syntaxique** (dont les terminaux sont ces tokens)
- “Lexical analyzer generator”
- Définit les tokens à l’aide d’un **langage régulier**
- Fichier de spécification (p.e. “scanneur.l”) :

```
... section déclarations ...
%%
... section règles de grammaire ...
%%
... section code C ...
```

Fichier scanneur.l


```
digit [0-9]
letter [a-zA-Z]

%%
{digit}+                                return INTEGER;
{letter}({letter}|{digit}|_)*           return IDENT;
"+"                                     return '+';
"-"                                     return '-';
"*"                                     return '*';
"/"                                     return '/';
"("                                     return '(';
")"                                     return ')';
\n                                      /* ignore end of line */;
[ \t]+                                  /* ignore whitespace */;
%%
```

Fichier parseur.y (partie 1)


```
%{
#include <stdio.h>
#include <stdlib.h>

int yywrap( void )
{ return 1; }

void yyerror( const char *msg )
{ printf( "error: %s\n", msg ); exit( 1 ); }

%}

%token INTEGER IDENT

%start expression
```

Fichier parseur.y (partie 2)


```
%%

expression: expr_add ;

expr_add: expr_mul
        | expr_add '+' expr_mul
        | expr_add '-' expr_mul
        ;

expr_mul: primary
        | expr_mul '*' primary
        | expr_mul '/' primary
        ;

primary: INTEGER | IDENT | '(' expression ')' ;

%%

#include "lex.yy.c"

int main( void )
{ yyparse(); return 0; }
```

Parseur en action


```
% lex scanneur.l     # scanneur.l ==> lex.yy.c
% yacc parseur.y     # parseur.y ==> y.tab.c
% gcc -o parseur.exe y.tab.c
% ./parseur.exe
12*(3+x-4)
% echo $?
0
% ./parseur.exe
x=x+1
=error: syntax error
% echo $?
1
```

Validation vs construction d'ASA

- Le parseur généré ne fait que **valider** que le programme source respecte la grammaire
- Mais la représentation intermédiare que nous recherchons est celle des ASAs
- Pour créer un ASA il faut utiliser les **actions sémantiques** de yacc et lex
- Le concept des actions sémantiques vient de la théorie des **grammaires attribuées**

Grammaires attribuées

- Une **grammaires attribuées** est une grammaire qui spécifie des **attributs** (ou propriétés) qui sont attachés à chaque catégorie et des règles permettant de **calculer** ces attributs pour chaque règle de production

- Le concepteur d'une grammaire a libre choix des attributs qu'il désire utiliser

- La notation normalement utilisée pour désigner l'attribut A de la catégorie C est $C.A$

- Les attributs et leurs règles de calcul deviennent donc une **technique pour attacher une sémantique** au programme

Exemple de grammaire attribuée (1)

- On désire attacher à la catégorie <*integer*> (et <*digit*>) l'attribut *valeur*, qui est sa valeur numérique

- GHC sans attributs :

```
<integer> ::=
      <digit>
    | <integer> <digit>


<digit> ::=
      "0"
    | "1"
    | "2"
    | "3"
    | "4"
    | "5"
    | "6"
    | "7"
    | "8"
    | "9"
```

Exemple de grammaire attribuée (2)

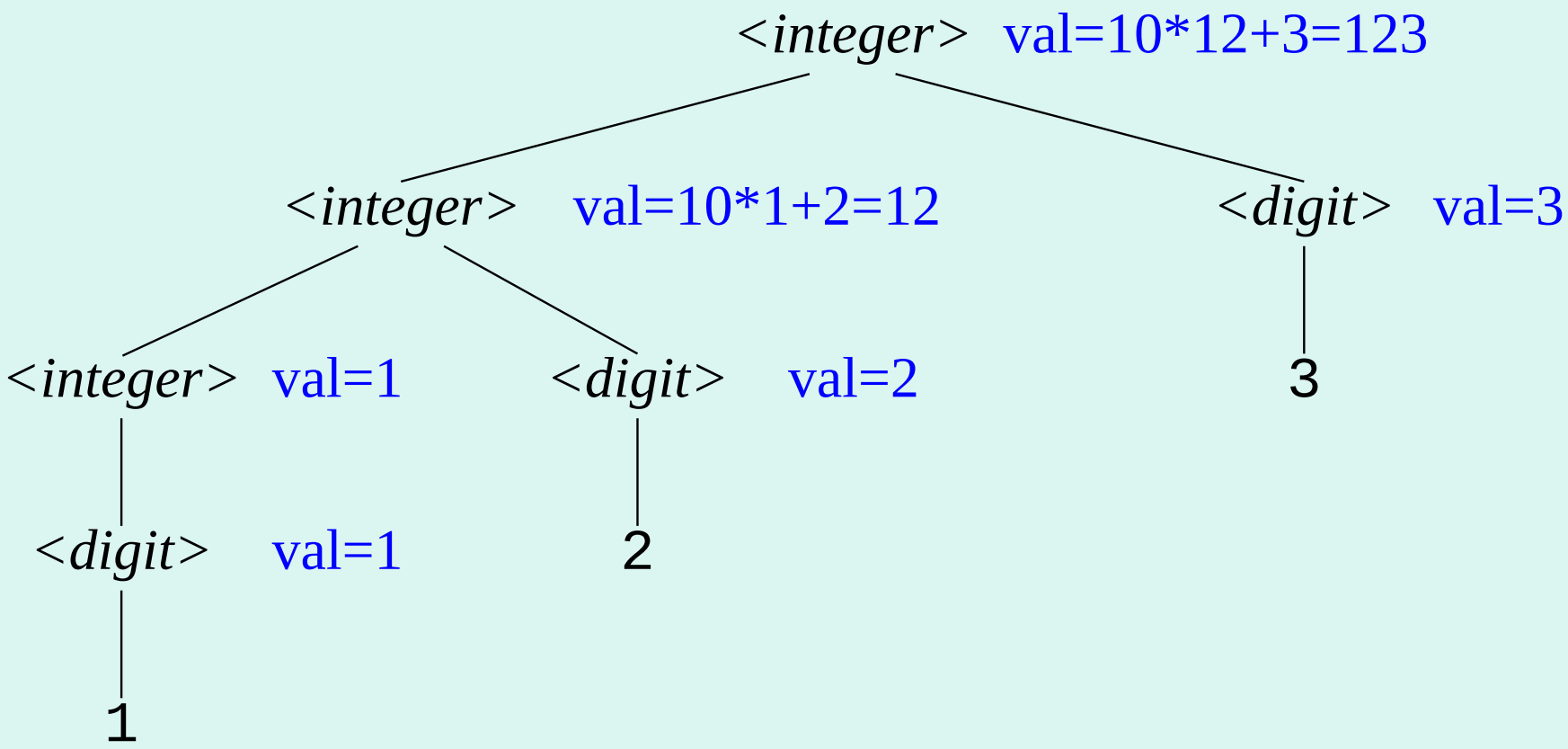
- GHC avec attributs :

```
<integer> ::=
    <digit>1                    <integer>.val = <digit>1.val
  | <integer>1 <digit>2         <integer>.val = 10*<integer>1.val
                                                + <digit>2.val

<digit> ::=
    "0"                         <digit>.val = 0
  | "1"                         <digit>.val = 1
  | "2"                         <digit>.val = 2
  | "3"                         <digit>.val = 3
  | "4"                         <digit>.val = 4
  | "5"                         <digit>.val = 5
  | "6"                         <digit>.val = 6
  | "7"                         <digit>.val = 7
  | "8"                         <digit>.val = 8
  | "9"                         <digit>.val = 9
```

Exemple de grammaire attribuée (3)

- Le calcul des attributs se fait en utilisant l'arbre de dérivation et les règles de calcul des attributs

- Exemple pour la chaîne **123** :



Exemple de grammaire attribuée (4)

- Plus exactement le calcul des attribut correspond à la **résolution d'un système d'équations** produit lors de l'analyse syntaxique

- Exemple pour la chaîne **123** :

$<integer>_a$.val = 10*$<integer>_b$.val + $<digit>_f$.val
$<integer>_b$.val = 10*$<integer>_c$.val + $<digit>_e$.val
$<integer>_c$.val = $<digit>_d$.val
$<digit>_d$.val = 1
$<digit>_e$.val = 2
$<digit>_f$.val = 3

- Les lettres a, b, ... sont des étiquettes qui identifient les noeuds internes de l'arbre de dérivation

Exemple : expressions constantes (1)

- Comment exprimer l'attribut **"expression est constante"**?
- Une expression est constante si elle a la même valeur quel que soit le contexte d'évaluation (contenu des variables)
- Besoin des attributs *cst* et *val*
- Ce sont des **attributs synthétisés** (l'attribut du parent dépend seulement des attributs des enfants)

Exemple : expressions constantes (2)


```
<expression> ::=
   <expr-add>1
    <expression>.cst = <expr-add>1.cst
    <expression>.val = <expr-add>1.val

<expr-add> ::=
   <expr-mul>1
    <expr-add>.cst = <expr-mul>1.cst
    <expr-add>.val = <expr-mul>1.val

 | <expr-add>1 "+" <expr-mul>2
    <expr-add>.cst = <expr-add>1.cst & <expr-mul>2.cst
    <expr-add>.val = if <expr-add>.cst then
                       <expr-add>1.val + <expr-mul>2.val
                     else
                       0

 | <expr-add>1 "-" <expr-mul>2
    <expr-add>.cst = <expr-add>1.cst & <expr-mul>2.cst
    <expr-add>.val = if <expr-add>.cst then
                       <expr-add>1.val - <expr-mul>2.val
                     else
                       0
```

Exemple : expressions constantes (3)


```
<expr-mul> ::=
    <primary>1
     <expr-mul>.cst = <primary>1.cst
     <expr-mul>.val = <primary>1.val

  | <expr-mul>1 "*" <primary>2
     <expr-mul>.cst = <expr-mul>1.cst & <primary>2.cst
     <expr-mul>.val = if <expr-mul>.cst then
                        <expr-mul>1.val * <primary>2.val
                      else
                        0

  | <expr-mul>1 "/" <primary>2
     <expr-mul>.cst = <expr-mul>1.cst & <primary>2.cst
                      & <primary>2.val != 0
     <expr-mul>.val = if <expr-mul>.cst then
                        <expr-mul>1.val / <primary>2.val
                      else
                        0
```

Exemple : expressions constantes (4)


```
<primary> ::=
   <integer>1
    <primary>.cst = true
    <primary>.val = <integer>1.val

 | <ident>1
    <primary>.cst = false
    <primary>.val = 0

 | "(" <expression>1 ")"
    <primary>.cst = <expression>1.cst
    <primary>.val = <expression>1.val
```

Exemple : code machine (1)

- Comment exprimer l'attribut **"code machine"**?
- Le code machine c'est la **séquence d'instructions** qui va calculer la valeur de l'expression et l'empiler sur la pile (nous supposons une machine virtuelle à pile simple avec instructions ADD, SUB, MUL, DIV, PUSH *const*, LOAD *offset*)

  2*(x+1) $\Longrightarrow$ PUSH 2 ; LOAD 5 ; PUSH 1 ; ADD ; MUL

- Besoin des attributs *code* et *fs*
- L'attribut *code* est synthétisé, mais *fs* est un **attribut hérité** (l'attribut de l'enfant dépend des attributs du parent)

Exemple : code machine (2)


```
<expression> ::=
   <expr-add>1
    <expression>.code = <expr-add>1.code
    <expr-add>1.fs = <expression>.fs

<expr-add> ::=
   <expr-mul>1
    <expr-add>.code = <expr-mul>1.code
    <expr-mul>1.fs = <expr-add>.fs

 | <expr-add>1 "+" <expr-mul>2
    <expr-add>.code = <expr-add>1.code .
                      <expr-mul>2.code .
                      "ADD"
    <expr-add>1.fs = <expr-add>.fs
    <expr-mul>2.fs = <expr-add>.fs + 1

 | <expr-add>1 "-" <expr-mul>2
    <expr-add>.code = <expr-add>1.code .
                      <expr-mul>2.code .
                      "SUB"
    <expr-add>1.fs = <expr-add>.fs
    <expr-mul>2.fs = <expr-add>.fs + 1
```

Exemple : code machine (3)


```
<expr-mul> ::=
    <primary>1
     <expr-mul>.code = <primary>1.code
     <primary>1.fs = <expr-mul>.fs

  | <expr-mul>1 "*" <primary>2
     <expr-mul>.code = <expr-mul>1.code .
                       <primary>2.code .
                       "MUL"
     <expr-mul>1.fs = <expr-mul>.fs
     <primary>2.fs = <expr-mul>.fs + 1

  | <expr-mul>1 "/" <primary>2
     <expr-mul>.code = <expr-mul>1.code .
                       <primary>2.code .
                       "DIV"
     <expr-mul>1.fs = <expr-mul>.fs
     <primary>2.fs = <expr-mul>.fs + 1
```

Exemple : code machine (4)


```
<primary> ::=
   <integer>1
    <primary>.code = "PUSH <integer>1.val"

 | <ident>1
    <primary>.code = "LOAD <primary>.fs+<ident>1.pos"

 | "(" <expression>1 ")"
    <primary>.code = <expression>1.code
    <expression>1.fs = <primary>.fs
```

Construction de l'AST (1)

- L'AST peut être vu comme un attribut synthétisé qui est calculé lors de l'analyse syntaxique
- Chaque type de noeud dans l'AST est une **classe** qui possède un **constructeur**, i.e.

new noeud(enfant$_1$, enfant$_2$, ...)

Construction de l'AST (2)


```
<expression> ::=
   <expr-add>1
    <expression>.ast = <expr-add>1.ast

<expr-add> ::=
   <expr-mul>1
    <expr-add>.ast = <expr-mul>1.ast

 | <expr-add>1 "+" <expr-mul>2
    <expr-add>.ast = new add( <expr-add>1.ast,
                              <expr-mul>2.ast )

 | <expr-add>1 "-" <expr-mul>2
    <expr-add>.ast = new sub( <expr-add>1.ast,
                              <expr-mul>2.ast )
```

Construction de l'AST (2)


```
<expr-mul> ::=
   <primary>1
    <expr-mul>.ast = <primary>1.ast

 | <expr-mul>1 "*" <primary>2
    <expr-mul>.ast = new mul( <expr-mul>1.ast,
                              <primary>2.ast )

 | <expr-mul>1 "/" <primary>2
    <expr-mul>.ast = new div( <expr-mul>1.ast,
                              <primary>2.ast )

<primary> ::=
   <integer>1
    <primary>.ast = new integer( <integer>1.val )

 | <ident>1
    <primary>.ast = new ident( <ident>1.name )

 | "(" <expression>1 ")"
    <primary>.ast = <expression>1.ast
```

Yacc et actions sémantiques

- Yacc permet d'annoter les règles de production avec des **actions sémantiques** qui calculent un seul **attribut synthétisé**

- C'est donc principalement utile pour la **construction d'un AST**

- Dans les actions sémantiques :
  - L'identifiant **$$** désigne **l'attribut du parent**
  - L'identifiant **$*i*** désigne **l'attribut de l'enfant *i***

- Le type de l'attribut est **YYSTYPE** (`int` par défaut)

Définition du type YYSTYPE


```
typedef struct node
  {
    enum { integer, ident, add2, sub2, mul2, div2 } type;
    union
      {
        struct { int val; } _integer;
        struct { char *name; } _ident;
        struct { struct node *child1, *child2; } _binop;
      };
  } node, *ast;

#define YYSTYPE ast
```

Constructeurs de noeuds


```
ast make_integer( char *tok )
{ ast p = (ast)malloc(sizeof(node));
  p->type = integer;
  p->_integer.val = atoi( tok );
  return p;
}

ast make_ident( char *tok )
{ ast p = (ast)malloc(sizeof(node));
  p->type = ident;
  p->_ident.name = (char*)malloc(strlen(tok)+1);
  strcpy( p->_ident.name, tok );
  return p;
}

ast make_binop( int type, ast child1, ast child2 )
{ ast p = (ast)malloc(sizeof(node));
  p->type = type;
  p->_binop.child1 = child1;
  p->_binop.child2 = child2;
  return p;
}
```

Fichier scanneur.l


```
digit [0-9]
letter [a-zA-Z]
identifier {letter}({letter}|{digit}|_)*

%%
{digit}+      yylval = make_integer(yytext); return INTEG
{identifier}  yylval = make_ident(yytext); return IDENT;
"+"           return '+';
"-"           return '-';
"*"           return '*';
"/"           return '/';
"("           return '(';
")"           return ')';
\n            /* ignore end of line */;
[ \t]+        /* ignore whitespace */;
%%
```

Fichier parseur.y (partie 1)


```
%{
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

int yywrap( void ) ...
void yyerror( const char *msg ) ...

typedef struct node ...

#define YYSTYPE ast

ast make_integer( char *tok ) ...
ast make_ident( char *tok ) ...
ast make_binop( int type, ast child1, ast child2 ) ...
void translate( ast p ) ...

%}

%token INTEGER IDENT

%start program
```

Fichier parseur.y (partie 2)


```
%%

program: expression                 { translate( $1 ); }
       ;

expression: expr_add                { $$ = $1; }
          ;

expr_add: expr_mul                  { $$ = $1; }
        | expr_add '+' expr_mul     { $$ = make_binop( add2, $1, $3 ); }
        | expr_add '-' expr_mul     { $$ = make_binop( sub2, $1, $3 ); }
        ;

expr_mul: primary                   { $$ = $1; }
        | expr_mul '*' primary      { $$ = make_binop( mul2, $1, $3 ); }
        | expr_mul '/' primary      { $$ = make_binop( div2, $1, $3 ); }
        ;

primary: INTEGER                    { $$ = $1; }
       | IDENT                      { $$ = $1; }
       | '(' expression ')'         { $$ = $2; }
       ;

%%

#include "lex.yy.c"

int main( void )
{ yyparse(); return 0; }
```

Fonction de traduction


```
void translate( ast p )  /* convert to prefix expression */
{
  switch (p->type)
    {
      case integer:
        printf( "%d", p->_integer.val );
        break;

      case ident:
        printf( "%s", p->_ident.name );
        break;

      default:
        printf( "(" );
        switch (p->type)
          { case add2: printf( "+" ); break;
            case sub2: printf( "-" ); break;
            case mul2: printf( "*" ); break;
            case div2: printf( "/" ); break;
          }
        printf( " " );
        translate( p->_binop.child1 );
        printf( " " );
        translate( p->_binop.child2 );
        printf( ")" );
        break;
    }
}
```

Parseur en action


```
% lex scanneur.l     # scanneur.l ==> lex.yy.c
% yacc parseur.y     # parseur.y ==> y.tab.c
% gcc -o parseur.exe y.tab.c
% ./parseur.exe
12*(3+x-4)/(((5)))
(/ (* 12 (- (+ 3 x) 4)) 5)
```

Silex et LALR-SCM

- Outils très similaires à lex et yacc, mais axés sur Scheme (pour la spécification et le code du parseur généré)

- Ces logiciels et le compilateur/interprète Gambit ont été conçus au DIRO et sont disponibles sur la page web du cours

- Exemple d'utilisation :

```
% gsi silex/silex.scm -e '(lex "scanner.l" "scanner.scm")'
% gsi -e '(load "Lalr/lalr.scm")' parser.scm
2*x+3
(+ (* 2 x) 3)
```

Exemple Silex


```
;;; File: "scanner.l"

digit  [0-9]
letter [a-zA-Z]
identifier ({letter}|_)({letter}|{digit}|_)*

%%

[ \n]+          (yycontinue)
{digit}+        (cons 'INTEGER (string->number yytext))
{identifier}    (cons 'IDENT   (string->symbol yytext))
"+"             (cons '+       #f)
"-"             (cons '-       #f)
"*"             (cons '*       #f)
"/"             (cons '/       #f)
<<EOF>>         (cons '*eoi*   #f)
<<ERROR>>       (error "invalid token")
```

Exemple LALR-SCM


```
;;; File: "parser.scm"

(include "scanner.scm") ;; generated by Silex
(include "Lalr/lalr.scm") ;; defines lalr-parser macro

(define parser
  (lalr-parser

    (INTEGER IDENT + - * /) ;; tokens

    (e (e + t)        : (list '+ $1 $3)
       (e - t)        : (list '- $1 $3)
       (t)            : $1)

    (t (t * f)        : (list '* $1 $3)
       (t / f)        : (list '/ $1 $3)
       (f)            : $1)

    (f (INTEGER)      : $1
       (IDENT)        : $1)))

(lexer-init 'port (current-input-port))

(pretty-print (parser lexer error))
```

Scheme

- Créé au MIT en 1975 par Guy Steele et Gerald Sussman
- Lisp “épuré” pour étudier le langage Actor
  - Syntaxe préfixe
  - Calcul = expression
  - Typage dynamique
  - “Garbage collection”
  - Fonctions d’ordre supérieur
  - Code source = donnée
  - Macros
  - Fermetures
  - Continuations

Syntaxe préfixe parenthésée

- ($\langle$*opération*$\rangle$ $\langle$*arg*$_1\rangle$ $\langle$*arg*$_2\rangle$...)

  L'opération peut-être le nom d'une forme syntaxique spéciale (`lambda`, `define`, etc), ou une expression dont la valeur est une fonction

- Exemple:

  `(* 2 (+ 3 4 5))` = `2 * (3+4+5)`

- En Lisp/Scheme tout ressemble à un appel de fonction (inconvénient et avantage)

`lambda`-expressions

- `(lambda (`$\langle id_1 \rangle$ $\langle id_2 \rangle$`...)` $\langle expr \rangle$`)`

- Dénote une fonction dont les paramètres sont $\langle id_1 \rangle$, $\langle id_2 \rangle$... et le corps est $\langle expr \rangle$

- Exemple :

  ```
  (lambda (x) (* x x))
  ```

  est la fonction de mise au carré

Définitions (1)

- `(define` ⟨*id*⟩ ⟨*expr*⟩`)`
- Crée la variable ⟨*id*⟩ initialisée à la valeur de ⟨*expr*⟩
- Exemple :

```
(define pi (* 4 (atan 1))
(define pi*2 (* pi 2))
(define carre (lambda (x) (* x x)))
(define z (carre pi*2))
```

Définitions (2)

- `(define (`$\langle id_1 \rangle$ $\langle id_2 \rangle$`...)` $\langle expr \rangle$`)` =
  `(define` $\langle id_1 \rangle$ `(lambda (`$\langle id_2 \rangle$`...)` $\langle expr \rangle$`))`

- La forme abrégée est plus compacte mais équivalente à tout point de vue à la definition avec lambda-expression

- Exemple :

```
(define (carre x)
  (* x x))

(define carre
  (lambda (x)
    (* x x)))
```

Affectation

- `(set!` ⟨*id*⟩ ⟨*expr*⟩`)`

- L'affectation permet de changer le contenu d'une variable existante

- Exemple :

```
(define n 10)

(set! n (+ n 1))

n  =>  11

(set! n "allo")

n  =>  "allo"
```

Séquencement

- `(begin` $\langle expr_1 \rangle$ $\langle expr_2 \rangle$`...)`

- Évalue les expressions de gauche à droite et retourne la valeur de la dernière expression

- Utile pour forcer un ordre d'évaluation

- Exemple :

```
(define n 10)

(begin (set! n (+ n 1)) n)  =>  11

(begin
  (display "a")
  (display "b"))  =>  affiche "ab"
```

Évaluation conditionnelle (1)

- `(if` $\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `[`$\langle expr_3 \rangle$`])`
- Évalue soit $\langle expr_2 \rangle$ ou $\langle expr_3 \rangle$ en fonction de la valeur de $\langle expr_1 \rangle$
- Faux = `#f`, Vrai = `#t` ou toute autre valeur
- Exemple :

```
(define (min x y)
  (if (< x y) x y))

(define n (min 5 2))

(if (not (= n 2))
    (display "y a un problème!"))
```

Évaluation conditionnelle (2)

- `(cond` $\langle clause_1 \rangle$ $\langle clause_2 \rangle$`...)`
- $\langle clause \rangle$ ::= `(`$\langle expr_1 \rangle$ $\langle expr_2 \rangle$`)`
- $\langle clause \rangle$ ::= `(else` $\langle expr_2 \rangle$`)`
- Évalue chaque $\langle expr_1 \rangle$ et s'arrête à la première qui est vraie, puis retourne la valeur de $\langle expr_2 \rangle$ de cette clause
- Exemple :

```
(define (f x)
  (cond ((< x 0) "négatif")
        ((> x 0) "positif")
        (else    "zéro")))

(f 4)  =>  "positif"
```

Évaluation conditionnelle (3)

- `(case` $\langle expr_1 \rangle$ $\langle clause_1 \rangle$ $\langle clause_2 \rangle$`...)`
- $\langle clause \rangle$ ::= `((`$\langle lit_1 \rangle$ $\langle lit_2 \rangle$`...)` $\langle expr_2 \rangle$`)`
- $\langle clause \rangle$ ::= `(else` $\langle expr_2 \rangle$`)`
- Évalue $\langle expr_1 \rangle$, c'est la *clé*; trouve la première clause pour laquelle $\langle lit_i \rangle$ est égale à la clé, puis retourne la valeur de $\langle expr_2 \rangle$ de cette clause
- Exemple :

```
(case (* 2 2)
  ((0)       "rien")
  ((1 2 3 4) "peu")
  (else      "beaucoup")) => "peu"
```

Évaluation conditionnelle (4)

- `(and` $\langle expr_1 \rangle$ $\langle expr_2 \rangle$`...)`
- `and` évalue les expressions de gauche à droite et s'arrête au premier résultat `#f` et retourne la valeur de la dernière évaluation
- `(or` $\langle expr_1 \rangle$ $\langle expr_2 \rangle$`...)`
- `or` évalue les expressions de gauche à droite et s'arrête au premier résultat non-`#f` et retourne la valeur de la dernière évaluation
- Exemple :

```
(define n 20)
(and (> n 5) (< n 9))   =>  #f
(or (> n 5) (< n 9))    =>  #t
(or (> n 5) (display "a")) => *rien*
```

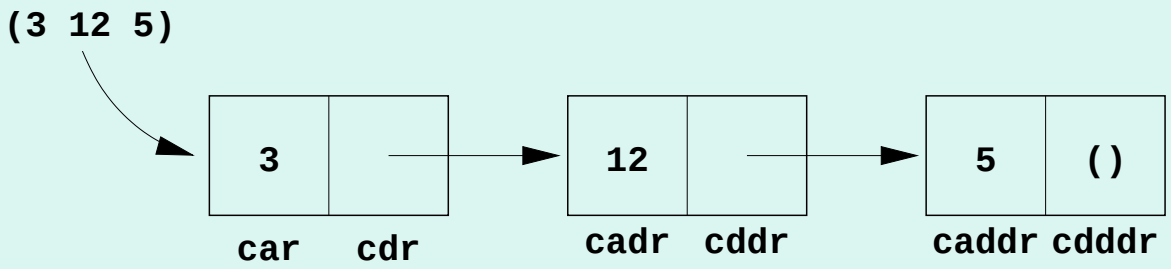
Les listes (1)

- Type structuré prédéfini qui dénote une séquence d'éléments
- Constructeurs = fonctions `list` et `cons`
- Exemple :

```
(list (+ 1 2)
      (* 3 4)
      5)          =>  (3 12 5)

(list)  =>  ()

(cons (+ 1 2)
      (cons (* 3 4)
            (cons 5
                  (list))))
   =>  (3 12 5)
```

Les listes (2)

- Les listes sont constituées de **paires**



- Les fonctions `car`, `cdr`, `cadr`, ... permettent d'accéder aux champs des paires

```
(car (list 3 12 5))   =>  3

(cdr (list 3 12 5))   =>  (12 5)

(cadr (list 3 12 5))  =>  12

(cddr (list 3 12 5))  =>  (5)
```

Les listes (3)

- `cons` est le constructeur de paire

$$\texttt{(car (cons } X \ Y\texttt{))} = X$$
$$\texttt{(cdr (cons } X \ Y\texttt{))} = Y$$

- Le champ `cdr` peut contenir autre chose qu'une liste (*liste impropre*) :

```
(car (cons 1 2))   =>  1

(cdr (cons 1 2))   =>  2

(cons 1 2)         =>  (1 . 2)

(list 1 2)         =>  (1 2)

(cons 1 (list 2))  =>  (1 2)
```

Les listes (4)

- Comment écrire une liste littérale?

```
(car (3 12 5))              =>  *erreur*

(car (quote (3 12 5)))      =>  3

(car '(3 12 5))             =>  3

(cons 3 '())                =>  (3)
```

- `(quote` ⟨*arg*⟩`)` = `'`⟨*arg*⟩

- La forme `quote` traite son argument comme une valeur littérale (elle ne se fait pas évaluer)

Opérations sur les listes (1)


```
(define x (cons 1 2))

(begin (set-car! x 3) x)  =>  (3 . 2)

(begin (set-cdr! x 4) x)  =>  (3 . 4)

(pair? x)                 =>  #t

(null? x)                 =>  #f

(null? '())               =>  #t

(list? x)                 =>  #f

(list? '(3 4))            =>  #t

(begin (set-cdr! x x) (list? x)) => #f
```

Opérations sur les listes (2)


```
(define x (list "a" "b" "c"))

(length x)      => 3

(list-ref x 1)  =>  "b"

(append x x)
   =>  ("a" "b" "c" "a" "b" "c")

(reverse x)
   =>  ("c" "b" "a")
```

Exemples (1)


```
(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))

(define (list-ref x n)
  (if (= n 0)
      (car x)
      (list-ref (cdr x) (- n 1))))

(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Exemples (2)


```
(define (write x)
  (cond ((null? x)
         (display "()"))
        ((pair? x)
         (begin
           (display "(")
           (write (car x))
           (display " . ")
           (write (cdr x))
           (display ")")))
        ((number? x)
         (display (number->string x)))
        ((boolean? x)
         (display (if x "#t" "#f")))
        ((procedure? x)
         (display "#<procedure>"))
        ((string? x)
         ...)
        ...))
```

Fonctions d'ordre supérieur (1)

- Les fonctions sont des données de *première classe*
- Une fonction peut recevoir des fonctions en paramètre et/ou retourner une fonction comme résultat
- Exemple :

```
(map sqrt '(1 4 9))     => (1 2 3)

(map list '(1 2 3))     => ((1) (2) (3))

(apply * (list 2 3 4)) => 24

(define (moyenne nbs)
  (/ (apply + nbs) (length nbs)))

(moyenne '(5 1 3))      => 3
```

Fonctions d’ordre supérieur (2)


```
(define (map f lst)
  (if (pair? lst)
      (cons (f (car lst))
            (map f (cdr lst)))
      '()))

(define (for-each f lst)
  (if (pair? lst)
      (begin
        (f (car lst))
        (for-each f (cdr lst)))))

(for-each display '(1 2 3))

  => affiche 123
```

Paramètres “reste” (1)

- Lorsque la liste de paramètre d’une fonction est une liste impropre, le paramètre dans le dernier `cdr` est le *paramètre reste*
- Le paramètre reste sera lié à la liste des paramètres actuels restants
- Exemple :

```
(define f (lambda (a b c)   (list a b c)))
(define g (lambda (a b . c) (list a b c)))

(f 1 2 3)   => (1 2 3)

(g 1 2)     => (1 2 ())

(g 1 2 3)   => (1 2 (3))

(g 1 2 3 4) => (1 2 (3 4))
```

Paramètres “reste” (2)


```
(define (f a b c)   (list a b c))
(define (g a b . c) (list a b c))


(define (h . x) x)

(h 1 2) => (1 2)   ; h = list


(define (print . x)
  (for-each display x))

(print 1 2 3) => affiche 123
```

Tests d'égalité (1)

- Quand sont deux valeurs égales?
  Ça dépend du contexte!
- Il y a plusieurs types d'égalités :
  - `(=` X Y`)` *égalité numérique*, $X - Y = 0$
  - `(equal?` X Y`)` *égalité structurelle*,
    objets de même "forme" et contenu
  - `(eq?` X Y`)` *identité*,
    mêmes objets (X et Y interchangeables pour toute opération)
  - `(eqv?` X Y`)` *égalité opérationnelle*,
    objets indistinguables par la majorité des opérateurs (est utilisé par `case`)

Tests d'égalité (2)

- Si on suppose que tous les objets sont alloués en mémoire, (`eq?` X Y) est vrai ssi X et Y sont les mêmes pointeurs (test très rapide à effectuer)
- (`eqv?` X Y) se comporte essentiellement comme (`=` X Y) si X et Y sont des nombres, sinon comme (`eq?` X Y)
- (`equal?` X Y) est vrai ssi la structure et le contenu de X et Y sont identiques (en gros, si l'impression de X et Y donne le même résultat)

  S'implante avec un algorithme récursif similaire à `write`

Tests d'égalité (3)


```
(define (equal? x y)
  (if (pair? x)
      (and (pair? y)
           (equal? (car x) (car y))
           (equal? (cdr x) (cdr y)))
      (eqv? x y)))

(define (eqv? x y)
  (if (number? x)
      (and (number? y)
           (eq? (exact? x) (exact? y))
           (= x y))
      (eq? x y)))
```

`equal?` risque de boucler à l'infini si x et y contiennent des cycles (possible à cause de `set-car!` et `set-cdr!`)

Liaison (1)

- Les formes `let`, `let*` et `letrec` créent des variables dont la portée est limité (*variable locale*)
- `(let ((`$\langle id_1 \rangle$ $\langle expr_1 \rangle$`)...)` $\langle expr \rangle$`)`
  liaison parallèle
- `(let* ((`$\langle id_1 \rangle$ $\langle expr_1 \rangle$`)...)` $\langle expr \rangle$`)`
  liaison séquentielle
- `(letrec ((`$\langle id_1 \rangle$ $\langle expr_1 \rangle$`)...)` $\langle expr \rangle$`)`
  liaison récursive

Liaison (2)

- `let` évalue toutes les $\langle expr_1 \rangle$ avant de créer les variables
- `let*` traite les liaisons séquentiellement (donc une variable est visible dans les liaisons qui suivent)
- `letrec` crée les variables en premier, évalue toutes les $\langle expr_1 \rangle$ et affecte les valeurs aux variables

Liaison (3)


```
(define x 4)

(let ((x (+ x 1))
      (y (* x x)))
  (list x y))  =>  (5 16)

(let* ((x (+ x 1))
       (y (* x x)))
  (list x y))  =>  (5 25)

(letrec ((f
          (lambda (n)
            (if (= n 0) #t (g (- n 1)))))
         (g
          (lambda (n)
            (if (= n 0) #f (f (- n 1))))))
  (f 3))  =>  #f
```

Liaison (4)

- La forme `let` est équivalente à l'appel d'une lambda-expression
- La forme `let*` est équivalente à une imbrication de `let`
- Exemple :

```
(let ((x A) (y B)) C)

   = ((lambda (x y) C)
      A
      B)

(let* ((x A) (y B)) C)

   = (let ((x A))
       (let ((y B))
         C))
```

Liaison (5)

- Une variante syntaxique du `let` permet d'exprimer des fonctions récursives et boucles
- `(let` $\langle id_0 \rangle$ `((`$\langle id_1 \rangle$ $\langle expr_1 \rangle$`)...)` $\langle expr \rangle$`)`
- $\langle id_0 \rangle$ est le nom de la lambda-expression utilisée pour faire la liaison ("`let` nommé")
- Exemple :

```
(let fact ((n 10))
  (if (= n 0) 1 (* n (fact (- n 1)))))
=
((letrec ((fact
           (lambda (n)
             (if (= n 0) 1 (* n (fact (- n
   fact)
 10)
```

Liaison (6)

- Dans le corps d’une forme de liaison, ou fonction, ou lambda-expression, on peut utiliser un `define` “interne” pour définir des variables locales
- Cela est équivalent à une liaison avec `letrec`

Liaison (7)


```
(define (even? x)
  (letrec ((f
            (lambda (n)
              (if (= n 0) #t (g (- n 1)))))
           (g
            (lambda (n)
              (if (= n 0) #f (f (- n 1))))))
    (f x)))


(define (even? x)

  (define (f n) (if (= n 0) #t (g (- n 1))))

  (define (g n) (if (= n 0) #f (f (- n 1))))

  (f x))
```

Types de données (1)

- Standard :
  - Nombres
  - Booléens (`#f`, `#t`)
  - Caractères
  - Chaînes de caractères
  - Symboles
  - Listes (paires/liste vide)
  - Vecteurs
  - Procédures
  - Ports (fichiers, entrée/sortie standard)

Types de données (2)

- Extensions non-standard propres à Gambit :
  - Mots clés (keywords)
  - Vecteurs homogènes de nombres
  - Tables
  - Enregistrements (records)
  - Paramètres dynamiques
  - Exceptions
  - Threads, mutex, variables de condition, ...
  - Ports (sockets, processus, ...)
  - ...

Représentation externe (1)

- La majorité des objets Scheme ont une représentation externe textuelle
- Cette représentation textuelle est utilisée par les fonctions `read` et `write` (et `pretty-print`)
- `read` se nomme parfois le **lecteur** (“reader”)
- `write` et `pretty-print` se nomment parfois l’**imprimeur** (“printer”)
- Le chargement de programmes et la boucle “read-eval-print” (REPL) utilisent le lecteur pour lire le code, et l’affichage du résultat se fait avec l’imprimeur

Représentation externe (2)


```
Gambit v4.2.8

> (write (cons 1 2))
(1 . 2)> (read)
(0001 . (   8/4 .3e1))
(1 2 3.)
> (with-output-to-string
    "val="
    (lambda () (write (cons 1 2))))
"val=(1 . 2)"
> (with-input-from-string
    "(0001 . (   8/4 .3e1))"
    (lambda () (read)))
(1 2 3.)
> (define x (list 1 2 3 4 5 6 7 8 9))
> (pretty-print (list x x))
((1 2 3 4 5 6 7 8 9)
 (1 2 3 4 5 6 7 8 9))
```

Nombres (1)

- Classifiés sur deux aspects :
  - position dans la “tour numérique” :
    entiers ⊂ rationnels ⊂ réels ⊂ complexe
  - exactitude = résultat mathématiquement correct
- Dans la pratique tout nombre point-flottant est inexact
- Dans la pratique il n’y a pas de distinction entre réel et rationnel
- Entiers : `-19  #b100  5.00  1e3  #e1e3`
- Rationnels/réels : `1/4  0.25  -1.25e-50`
- Complexe : `+i  1-3i  1/2+3.45i`

Nombres (2)

- Tests de type :
  `number?`, `integer?`, `rational?`, `real?`, `complex?`, `exact?`, `inexact?`
- Opérations fermées sur les nombres exacts :
  `+`, `-`, `*`, `/` (sauf division par 0), `floor`, `round`, `min`, `max`
- Opérations sur les entiers :
  `quotient`, `modulo`, `remainder`, `even?`, `odd?`
- Conversions :
  `(inexact->exact` X`)`
  `(exact->inexact` X`)`
  `(number->string` X [$base$]`)`
  `(string->number` X [$base$]`)`

Nombres (3)


```
Gambit v4.2.8

> (expt 2 129)
680564733841876926926749214863536422912
> (expt 5/2 4)
625/16
> (sqrt 2)
1.4142135623730951
> (* 1000000 1.000001)
1000000.9999999999
> (inexact->exact .1)
3602879701896397/36028797018963968
> (floor 1.000001)
1.
> (floor (inexact->exact 1.000001))
1
> (/ 3 0.)
+inf.
```

Caractères

- Représentation externe :

```
#\x  #\(  #\u1a44  #\space
```

- Opérations :

```
> (integer->char 65)
#\A
> (char->integer #\backspace)
8
> (char>? #\a #\x)
#f
> (char-ci=? #\a #\A)
#t
> (char-alphabetic? #\a)
#t
> (char-upcase #\a)
#\A
```

Chaînes de caractères (1)

- Chaîne = séquence de caractères
- Représentation externe :
  `"allo\n"   "il dit \"allo!\""`
- Opérations :

```
> (make-string 10 #\!)
"!!!!!!!!!!"
> (define x (string #\a #\b #\c #\d))
> x
"abcd"
> (eq? x "abcd")
#f
> (equal? x "abcd")
#t
> (string=? x "abcd")
#t
```

Chaînes de caractères (2)


```
> (string-length x)
4
> (string-ref x 2)
#\c
> (string-set! x 2 #\_)
> x
"ab_d"
> (substring x 1 3)
"b_"
> (string-append "abc" "123")
"abc123"
> (string->list "abcd")
(#\a #\b #\c #\d)
> (list->string '(#\a #\b #\c #\d))
"abcd"
> (string<? "allo" "ane")
#t
> (string-copy "marc")
"marc"
```

Symboles

- Symbole = chaîne immutable et “unique”
- Représentation externe : `n + a.b<->c$d!`
- Un symbole littéral doit être “quoté” pour ne pas le confondre avec une variable
- Opérations :

```
> (symbol->string 'allo)
"allo"
> (define x (string->symbol "allo"))
> x
allo
> (eq? x 'allo)
#t
> (equal? x 'allo)
#t
```

Keywords

- Keyword = chaîne immutable et “unique”
- Représentation externe : `n:` `base:`
- L’évaluation d’un keyword donne ce keyword (il est inutile de le “quoter”)
- Opérations :

```
> (keyword->string allo:)
"allo"
> (define x (string->keyword "allo"))
> x
allo:
> (eq? x allo:)
#t
```

- Les keywords sont principalement utilisés pour spécifier des paramètres optionnels nommés

Paramètres optionnels (1)

- Gambit permet les paramètres optionnels positionnels et nommés
- Dans la liste de paramètres il faut distinguer ces paramètres avec `#!optional` et `#!key`
- Si le paramètre actuel n'est pas fourni à l'appel, le paramètre prendra la valeur `#f` à moins qu'une valeur de défaut ne soit spécifiée
- Exemple :

```
> (define (f a #!optional b (c (* a a)))
    (list a b c))
> (f 3)
(3 #f 9)
> (f 3 4)
(3 4 9)
> (f 3 4 5)
(3 4 5)
```

Paramètres optionnels (2)

- Pour les paramètres optionnels nommés, le paramètre actuel doit être préfixé par le keyword qui correspond au nom du paramètre
- Exemple :

```
> (define (conv n #!key (base 10) (width 0))
    (let* ((s (number->string n base))
           (w (- width (string-length s))))
      (string-append
       (make-string (max w 0) #\space)
       s)))
> (conv 9)
"9"
> (conv 9 width: 5)
"    9"
> (conv 9 base: 2)
"1001"
> (conv 9 width: 5 base: 2)
" 1001"
```

Paramètres optionnels (3)

- Exemple : fonctions pour générer du HTML

```
((I) "italique") => "<I>italique</I>"

((TABLE)
 ((TR) ((TD) "a") ((TD) "b"))
 ((TR) ((TD) "c") ((TD) "d")))

=> "<TABLE><TR><TD>a</TD>..."
```

Paramètres optionnels (4)


```
(define (tag name)
  (lambda ()
    (lambda content
      (string-append
       "<" name ">"
       (apply string-append content)
       "</" name ">"))))

(define I     (tag "I"))
(define TABLE (tag "TABLE"))
(define TR    (tag "TR"))
(define TD    (tag "TD"))
```

Paramètres optionnels (5)

- Extension pour spécifier des attributs, p.ex.
  `((TABLE bgcolor: "red" border: "3") ...)`

```
(define (tag name)
  (lambda (#!key
           (border #f)
           (bgcolor #f))
    (lambda content
      (string-append
       "<" name
       (attr "border" border)
       (attr "bgcolor" bgcolor)
       ">"
       (apply string-append content)
       "</" name ">"))))

(define (attr name value)
  (if value
      (string-append " " name "=" value)
      ""))
```

Paramètres optionnels (6)


```
(define urgent (TABLE bgcolor: "red"))
(define row    (TR))
(define col    (TD))

(urgent
 (row (col "a") (col "b"))
 (row (col "c") (col "d")))

=> "<TABLE bgcolor=red><TR><TD>a</TD>..."
```

Vecteurs (1)

- Vecteur = séquence hétérogène d'objets
- Représentation externe :
  `#(5 () #\x)  #()  #(#(1 2) #(3 4))`
- Un vecteur littéral doit être "quoté"
- Les opérations sont similaires aux chaînes :
  - `(make-vector` N [X]`)` : allocation
  - `(vector` X_1`...)` : construction
  - `(vector-ref` V I`)` : indexation
  - `(vector-set!` V I X`)` : mutation
  - `(vector->list` V`)` : conversion à liste
  - `(list->vector` L`)` : conversion à vecteur
  - `subvector`, `vector-append`, `vector-copy`

Vecteurs (2)

- Matrice = vecteurs de vecteurs
- Par exemple `#(#(1 2 3) #(4 5 6))` est une matrice à deux dimensions (2×3)
- `(vector-ref (vector-ref` M I`)` J`)` accède à l'élément en position (I, J)
- Il serait utile de définir des fonctions pour manipuler des matrices à deux dimensions aisément :
  - `(make-mat` N_1 N_2`)` : allocation
  - `(mat-ref` M I_1 I_2`)` : indexation
  - `(mat-set!` M I_1 I_2 X`)` : mutation

Vecteurs (3)

- Première tentative :

```
(define (make-mat n1 n2)
  (make-vector n1 (make-vector n2 0)))

(define (mat-ref m i1 i2)
  (vector-ref (vector-ref m i1) i2))

(define (mat-set! m i1 i2 x)
  (vector-set! (vector-ref m i1) i2 x))

(define m (make-mat 2 3))

m => #(#(0 0 0) #(0 0 0))

(mat-set! m 1 2 999)

m => #(#(0 0 999) #(0 0 999))
```

Vecteurs (4)

- Il faut créer un vecteur pour chaque rangée!

```
(define (intervalle i j)
  (if (< i j)
      (cons i (intervalle (+ i 1) j))
      '()))

(intervalle 0 5) => (0 1 2 3 4)

(define (make-mat n1 n2)
  (list->vector
   (map (lambda (i) (make-vector n2 0))
        (intervalle 0 n1))))

(define m (make-mat 2 3))

m => #(#(0 0 0) #(0 0 0))

(mat-set! m 1 2 999)

m => #(#(0 0 0) #(0 0 999))
```

Vecteurs (5)

- Généralisation aux matrices multidimensionnelles :
  - (`make-mat` N_1 N_2...) : allocation
  - (`mat-ref` M I_1 I_2...) : indexation
  - (`mat-set!` M I_1 I_2...X) : mutation

```
(define (make-mat n1 . rest)
  (if (null? rest)
      (make-vector n1 0)
      (list->vector
       (map (lambda (i) (apply make-mat rest))
            (intervalle 0 n1)))))

(define m (make-mat 2 3 4))

m => #(#(#(0 0 0 0) #(0 0 0 0) #(0 0 0 0))
       #(#(0 0 0 0) #(0 0 0 0) #(0 0 0 0)))
```

Vecteurs (6)


```
(define (mat-ref m n1 . rest)
  (if (null? rest)
      (vector-ref m n1)
      (apply mat-ref
             (cons (vector-ref m n1)
                   rest))))

(define (mat-set! m n1 . rest)
  (if (null? (cdr rest))
      (vector-set! m n1 (car rest))
      (apply mat-set!
             (cons (vector-ref m n1)
                   rest))))

(mat-set! m 1 0 2 9)

m => #(#(#(0 0 0 0) #(0 0 0 0) #(0 0 0 0))
       #(#(0 0 9 0) #(0 0 0 0) #(0 0 0 0)))

(mat-ref m 1 0 2) => 9
```

Vecteurs homogènes (1)

- Gambit possède des vecteurs homogènes de nombres entiers et flottants
- Plus compact que les vecteurs hétérogènes
- Éléments de même taille :
  - `u8vector` / `s8vector` : entier 8 bits non-signé/signé
  - `u16vector` / `s16vector` : entier 16 bits
  - `u32vector` / `s32vector` : entier 32 bits
  - `u64vector` / `s64vector` : entier 64 bits
  - `f32vector` / `f64vector` : flottant 32/64 bits
- Représentation externe :

```
#u8(1 2 3)      #f64(1.5 2.3)
```

Vecteurs homogènes (2)

- Les mêmes opérations que les vecteurs hétérogènes sont disponibles (le nom de la fonction indique le type de vecteur homogène)
- Exemple :

```
> (define v (make-u8vector 5 9))
> v
#u8(9 9 9 9 9)
> (u8vector-length v)
5
> (u8vector-append '#u8(5 6) v)
#u8(5 6 9 9 9 9 9)
> (subu8vector '#u8(0 1 2 3) 1 3)
#u8(1 2)
```

Ensembles, dictionnaires, et structures avancées

- Certains algorithmes s'expriment bien avec des structures de données avancées telles que :
  - **Ensembles** : union, intersection, différence, test d'appartenance
  - **Dictionnaires** : ensemble de clés, chaque clé a une valeur associée

- Il y a plusieurs façon d'implanter ces structures en Scheme :
  - **Liste d'éléments**
  - **Liste d'association**
  - **Table** d'adressage dispersé (hash table)

Liste d'éléments (1)

- Un ensemble peut se représenter avec une liste de ses éléments :

```
(define premiers
  (list 2 5 7 3))

(define cours
  (list "ift3060" "ift2030" "ift2240"))
```

- Les fonctions prédéfinies `member`, `memv`, et `memq` testent l'appartenance d'une valeur à un ensemble sous forme de liste

- Le test d'égalité utilisé dépend de la fonction :
  `member` → `equal?`
  `memv` → `eqv?`
  `memq` → `eq?`

Liste d’éléments (2)


```
> (member 8 premiers)
#f
> (member 5 premiers)
(5 7 3)
> (member "ift1010" cours)
#f
> (member "ift2030" cours)
("ift2030" "ift2240")
> (memq "ift2030" cours)
#f
> (define cours2
    (list 'ift3060 'ift2030 'ift2240))
> (memq 'ift2030 cours2)
(ift2030 ift2240)
```

Liste d'éléments (3)


```
(define (union e1 e2)
  (cond ((null? e1)
         e2)
        ((member (car e1) e2)
         (union (cdr e1) e2))
        (else
         (cons (car e1)
               (union (cdr e1) e2)))))

(define (intersect e1 e2)
  (cond ((null? e1)
         '())
        ((member (car e1) e2)
         (cons (car e1)
               (intersect (cdr e1) e2)))
        (else
         (intersect (cdr e1) e2))))
```

Liste d'éléments (4)


```
(define (difference e1 e2)
  (cond ((null? e1)
         '())
        ((member (car e1) e2)
         (difference (cdr e1) e2))
        (else
         (cons (car e1)
               (difference (cdr e1) e2)))))

(define (egal? e1 e2)
  (and (null? (difference e1 e2))
       (null? (difference e2 e1))))
```

Liste d'association (1)

- Liste d'association = liste de paires, chaque paire est une association, le `car` est la clé, le `cdr` est la valeur associée

```
(define horaire
  (list (list 'ift3060 'lun 'mer)
        (list 'ift2030 'mar)
        (list 'ift2240 'mer 'jeu)))
```

- Les fonctions prédéfinies `assoc`, `assv`, et `assq` recherchent la paire qui contient la clé demandée (échec → `#f`)

- Le test d'égalité utilisé dépend de la fonction :
  `assoc` → `equal?`
  `assv` → `eqv?`
  `assq` → `eq?`

Liste d’association (2)

- Exemple :

```
> (assoc ’ift1010 horaire)
#f
> (assoc ’ift3060 horaire)
(ift3060 lun mer)
> (assq ’ift3060 horaire)
(ift3060 lun mer)
> (define x
    (cons (list ’ift1010 ’jeu)
          horaire))
> (assq ’ift1010 x)
(ift1010 jeu)
```

Liste d’association (3)

- On retrouve souvent ces idiomes :

```
; addition d’associations :

(let ((new-alist
        (cons (cons cle val) alist)))
  ... utiliser new-alist)

; lookup d’une clé :

(let ((x (assoc cle alist)))
  (if x
      (let ((val (cdr x)))
        ... utiliser val)
      ...))
```

Tables (1)

- Table = structure mutable contenant des associations clé/valeur
- Les opérations sont similaires aux vecteurs :
  - `(make-table)` : allocation
  - `(table-length` T`)` : longueur
  - `(table-ref` T C [V]`)` : indexation
  - `(table-set!` T C [V]`)` : mutation
  - `(table->list` T`)` : conversion à liste d'association
  - `(list->table` L`)` : conversion à table
- `(equal?` $table_1$ $table_2$`)` vrai ssi les deux tables contiennent les mêmes clés, et chaque clé a la même valeur dans les deux tables

Tables (2)


```
> (define t (make-table))
> (table-set! t 2 #t)
> (table-set! t 3 #t)
> (table-set! t 5 #t)
> (table-set! t 7 #t)
> (table-ref t 5 #f)
#t
> (table-ref t 8 #f)
#f
> (table-ref t 8)
*** ERROR -- Unbound table key
> (table-set! t 5)
> (table-ref t 5)
*** ERROR -- Unbound table key
> (table->list t)
((7 . #t) (3 . #t) (2 . #t))
```

Tables (3)

- Utilisation de table comme matrice 2D :

```
> (define t (make-table))
> (table-set! t (list 1 2) 111)
> (table-set! t (list 0 3) 222)
> (table-set! t (list 1 3) 333)
> (table-ref t (list 0 3))
222
```

- Utilisation de table comme dictionnaire :

```
> (define t (make-table))
> (table-set! t 'ift3060 '(lun mer))
> (table-set! t 'ift2030 '(mar))
> (table-set! t 'ift2240 '(mer jeu))
> (table-ref t 'ift3060 #f)
(lun mer)
> (table-ref t 'ift1010 #f)
#f
```

Tables (4)

- L'implantation des tables repose sur la technique d'adressage dispersé
  - Quel test d'égalité pour les clés? (`equal?` par défaut)
  - Quelle fonction de hachage pour les clés? (`equal?-hash` par défaut)
- Le test d'égalité, la fonction de hachage, etc de la table peuvent être spécifiées à la création grâce à des paramètres nommés :

```
> (define t (make-table test: eq?))
> (table-set! t 'pommes 1.25)
> (table-set! t 'oranges 3.50)
> (table-ref t 'pommes)
1.25
```

Tables (5)

- La fonction de hachage par défaut dépend de la fonction de test d'égalité pour les clés :
  `equal?` → `equal?-hash`
  `eqv?` → `eqv?-hash`
  `eq?` → `eq?-hash`
  `string=?` → `string=?-hash`
  `string-ci=?` → `string-ci=?-hash`
  sinon → `(lambda (x) 0)`
- Exemple :

```
> (equal?-hash (list 1 2))
518586543
> (equal?-hash (list 1 2))
518586543
> (eq?-hash (list 1 2))
2
> (eq?-hash (list 1 2))
3
```

Tables (6)

- Propriétés des tables :
  - `test:` $fonction$: comparaison des clés
  - `hash:` $fonction$: hachage des clés
  - `size:` $entier$: taille initiale
  - `min-load:` $nombre$: charge minimale
  - `max-load:` $nombre$: charge maximale
  - `weak-keys:` $Booleen$: clés retenues faiblement?
  - `weak-values:` $Booleen$: valeurs retenues faiblement?

Tables (7)

- Les tables avec `test: eq?` sont utiles pour attacher des propriétés aux objets, par ex. :
  - Numéro de série pour identifier l'objet
  - Sémaphore pour empêcher les accès concurrents à un objet
- Conceptuellement, pour chaque propriété c'est comme si on ajoutait un champ à chaque objet qui possède cette propriété
- Il n'est pas possible de changer la structure d'un objet après sa création, donc il faut utiliser une table externe pour stocker les propriétés
  Clé = objet Valeur = valeur de la propriété

Tables (8)

- Les numéros de série sont utilisés par Gambit pour étiquetter les objets dans la représentation externe :

```
> (define carre (lambda (x) (* x x)))
> (object->serial-number carre)
2
> carre
#<procedure #2 carre>
> (serial-number->object 2)
#<procedure #2 carre>
> #2
#<procedure #2 carre>
> (#2 7)
49
```

- Problème : on veut que le garbage collector puisse récupérer l'objet (et la propriété)

Tables (9)

- Une implantation par liste s'association ne récupère par les objets qui ne sont plus utilisés

```
> (define ns 0)
> (define (++ns) (set! ns (+ ns 1)) ns)
> (define tns '())
> (define (obj->ns o)
    (let ((x (assq o tns)))
      (if x
          (cdr x)
          (let ((n (++ns)))
            (set! tns (cons (cons o n) tns))
            n))))
> (define carre (lambda (x) (* x x)))
> (obj->ns carre)
1
> (obj->ns carre)
1
> (set! carre (lambda (x) (expt x 2)))
> (obj->ns carre)
2
> tns
((#<procedure #2 carre> . 2)
 (#<procedure #3> . 1))
```

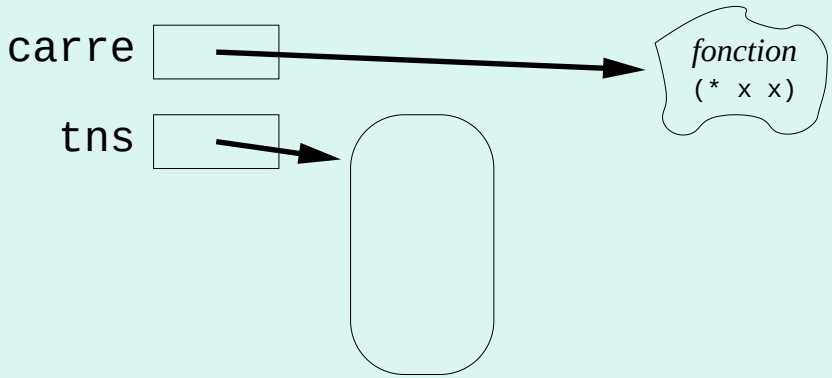
Tables (10)

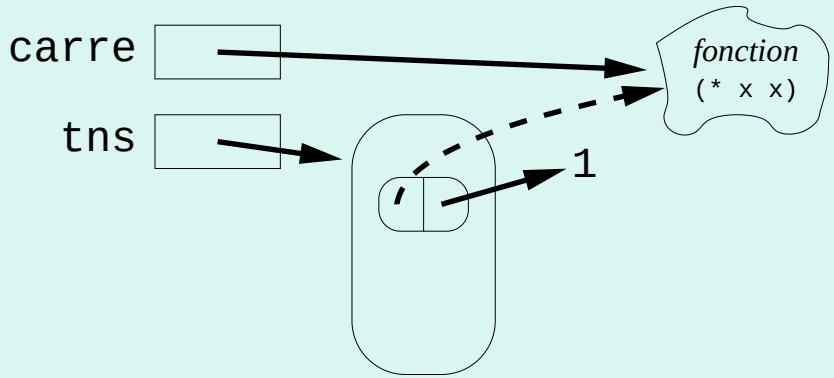
- Avec une table qui retient les clés faiblement, le GC récupèrera la mémoire s'il n'y a pas de référence (forte) vers l'objet

```
> (define tns
    (make-table test: eq? weak-keys: #t))
> (define (obj->ns o)
    (or (table-ref tns o #f)
        (let ((n (++ns)))
          (table-set! tns o n)
          n))))
> (define carre (lambda (x) (* x x)))
> (obj->ns carre)
1
> (obj->ns carre)
1
> (set! carre (lambda (x) (expt x 2)))
> (obj->ns carre)
2
> (##gc)
> (table->list tns)
((#<procedure #2 carre> . 2))
```

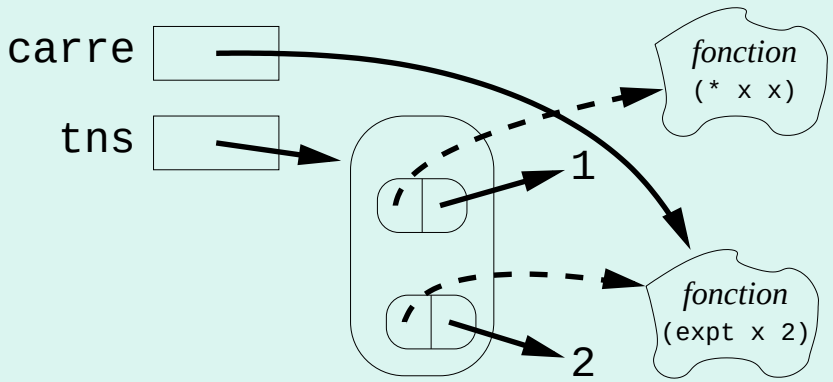
Tables (11)

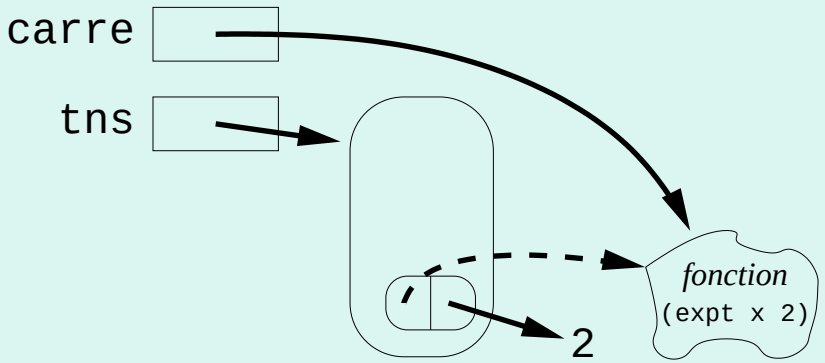
- Avant/après (`obj->ns carre`)





- Avant/après (`##gc`)





Enregistrements (1)

- Gambit supporte les types enregistrement, i.e. structures avec champs nommés
- La forme spéciale `define-type` possède plusieurs options
- La syntaxe la plus simple spécifie simplement le nom du type et le nom de chaque champ :

Gambit : `(define-type point2d x y)`

C : `typedef struct { int x; int y; } point2d;`

Enregistrements (2)

- Une définition de type est équivalente à un ensemble de définitions de fonctions :

```
(define (make-point2d x y)...   constructeur
(define (point2d? obj)...       prédicat
(define (point2d-x p)...        accès champ x
(define (point2d-x-set! p x)... mutation x
(define (point2d-y p)...        accès champ y
(define (point2d-y-set! p y)... mutation y
```

- Exemple :

```
> (define-type point2d x y)
> (define p (make-point2d 11 22))
> p
#<point2d #2 x: 11 y: 22>
> (point2d? p)
#t
> (point2d-x p)
11
> (point2d-x-set! p 33)
> p
#<point2d #2 x: 33 y: 22>
```

Enregistrements (3)

- Les enregistrements supportent l'héritage
- Le paramètre `extender:` nom spécifie le nom de la forme de définition de type qui doit être utilisée pour définir un sous-type :

```
> (define-type point2d
    extender: define-type-of-point2d
    x
    y)
> (define-type-of-point2d point3d z)
> (define p3 (make-point3d 11 22 33))
> p3
#<point3d #2 x: 11 y: 22 z: 33>
> (point2d? p3)
#t
> (point3d? p3)
#t
> (point2d-x p3)
11
> (point3d-z p3)
33
```

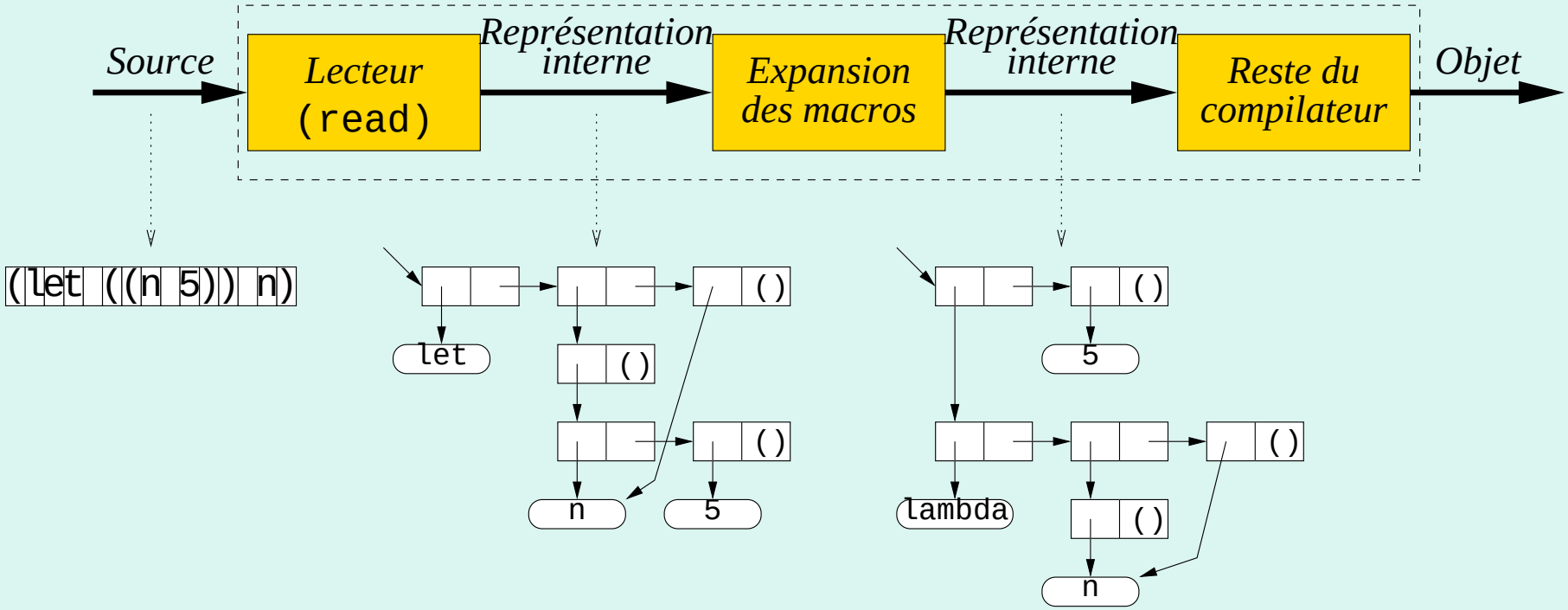
Enregistrements (4)

- Attributs des champs :
  - `read-only:` champ non-mutable
  - `equality-skip:` `equal?` l'ignore
  - `unprintable:` `write` l'ignore
  - `init:` *constante* valeur initiale
- Exemple :

```
> (define-type noeud
    (a unprintable:)
    (b read-only: init: 9)
    (c equality-skip:))
> (define x (make-noeud 1 2))
> (noeud-a-set! x x)
> x
#<noeud #2 b: 9 c: 2>
> (equal? x (make-noeud x 3))
#t
> (noeud-b-set! x 999)
*** ERROR -- Unbound variable: noeud-b-set!
```

Phases d'un compilateur Scheme

- Les phases d'un compilateur/interprète Scheme incluent **l'expansion des macros**



- Le lecteur fait l'analyse lexicale (et une analyse syntaxique de base, i.e. syntaxe des données)
- L'expanseur de macro détecte l'usage d'une macro et fait une transformation qui dépend de sa définition

Macros (1)

- Une macro exprime donc une **transformation source-à-source** sur le programme
- Une définition de macro indique la fonction qui transforme l'appel de macro en une **expression** qui sera évaluée à sa place
- Une définition de macro est donc un mini-compilateur, avec la particularité que le langage source et le langage cible sont identiques
- Les paramètres de la macro correspondent aux arguments **non-évalués** de l'appel de macro

Macros (2)

- Exemple : `(inc n)` → `(set! n (+ n 1))`

```
(define-macro (inc var)
  (list 'set! var (list '+ var 1)))
```

- L'appel de macro `(inc n)` va lier le paramètre `var` au **symbole** `n` (en d'autres termes, n ne se fait pas évaluer)

- L'évaluation du corps de la macro va créer **la liste** `(set! n (+ n 1))` qui est la représentation de l'expression après transformation

- Note : `inc` ne peut se définir comme une fonction car le paramètre n serait alors toujours évalué

Macros (3)

- Exemple d'expansion de macro :

```
(let ((x 1) (y 2))          ⟶   (let ((x 1) (y 2))
  (inc x)                          (set! x (+ x 1))
  (inc y)                          (set! y (+ y 1))
  (* x y))                         (* x y))
```

- Comment visualiser la transformation? Utiliser pp qui affiche le code d'une fonction :

```
> (pp (lambda () (inc z)))
(lambda () (set! z (+ z 1)))
```

- La puissance des macros en Lisp/Scheme vient du fait que le langage entier est disponible pour effectuer la transformation

Macros (4)

- Utile pour déboguer : (`assert` *précondition*)

```
> (define (fact n)
    (assert (>= n 0))
    (if (= n 0) 1 (* n (fact (- n 1)))))
> (fact 10)
3628800
> (fact -5)
*** ERROR IN fact -- failed: (>= n 0)
```

- Règle de transformation :

```
                        (or (>= n 0)
(assert (>= n 0)) ⟶         (error
                             "failed: (>= n 0)"))
```

Macros (5)


```
> (define-macro (assert precond)
    (let ((msg
           (with-output-to-string
             "failed: "
             (lambda () (write precond)))))
      (list 'or
            precond
            (list 'error msg))))
> (pp (lambda () (assert (>= n 0))))
(lambda ()
  (or (>= n 0)
      (error "failed: (>= n 0)")))
```

Macros (6)

- (let ((x E_1) (y E_2)) E_3) $\rightarrow$
  ((lambda (x y) E_3) E_1 E_2)

```
> (define-macro (let bindings body)
    (cons (list 'lambda
                (map car bindings)
                body)
          (map cadr bindings)))
> (let ((a 1) (b 2)) (+ a b))
3
> (pp (lambda ()
        (let ((a 1) (b 2)) (+ a b))))
(lambda ()
  ((lambda (a b) (+ a b))
   1
   2))
```

Macros (7)

- (let ((x E_1) (y E_2)) E_3...) $\rightarrow$
  ((lambda (x y) E_3...) E_1 E_2)

```
> (define-macro (let bindings . body)
    (cons (cons 'lambda
                (cons (map car bindings)
                      body))
          (map cadr bindings)))
> (let ((a 1) (b 2)) (write a) (+ a b))
3
> (pp (lambda ()
        (let ((a 1) (b 2))
          (write a)
          (+ a b))))
(lambda ()
  ((lambda (a b) (write a) (+ a b))
   1
   2))
```

Macros (8)

- (let* ((x E_1) (y E_2)) E_3...) $\rightarrow$
  (let ((x E_1)) (let* ((y E_2)) E_3...))
- (let* () E...) $\rightarrow$ (let () E...)

```
> (define-macro (let* bindings . body)
    (cons 'let
          (if (or (null? bindings)
                  (null? (cdr bindings)))
              (cons bindings body)
              (list (list (car bindings))
                    (cons 'let*
                          (cons (cdr bindings)
                                body))))))
> (pp (lambda ()
        (let* ((a 1) (b (+ a 1)))
          (write a)
          (+ a b))))
(lambda ()
  (let ((a 1))
    (let ((b (+ a 1)))
      (write a)
      (+ a b))))
```

Quasiquote (1)

- La construction de l'expression avec `list` et `cons` devient difficile à lire pour les expressions le moindrement complèxes

- La forme spéciale `(quasiquote` P`)` qu'on peut abréger `‘` P permet de construire une donnée qui est presque constante

- P est un gabarit où chaque partie qui est variable est préfixée d'une virgule :
  - `,`E : remplace cette partie par la valeur de E
  - `,@`E : remplace cette partie par les éléments de la liste obtenue en évaluant E

Quasiquote (2)

- Exemple : `‘(a ,b c) = (list ’a b ’c)`
- Exemple :
  `‘(a b ,@c) = (cons ’a (cons ’b c))`
- Exemple :

```
(define-macro (inc var)
  (list ’set! var (list ’+ var 1)))

(define-macro (inc var)
  ‘(set! ,var (+ ,var 1)))

(define-macro (assert precond)
  ‘(or ,precond
       (error ,(with-output-to-string
                 "failed: "
                 (lambda () (write precond))))))
```

Quasiquote (3)


```
(define-macro (let bindings . body)
  (cons (cons 'lambda
              (cons (map car bindings)
                    body))
        (map cadr bindings)))

(define-macro (let bindings . body)
  `((lambda ,(map car bindings) ,@body)
    ,@(map cadr bindings)))

(define (moy . nbs)
  (/ (apply + nbs) (length nbs)))

(define-macro (moy . nbs)
  `(/ (+ ,@nbs) ,(length nbs)))
```

Macro ou fonction?

- **Éliminer le coût d'appel de fn.** (macro `moy`)
- **Passer des paramètres autrement que par valeur** (macro `inc`)
- **Faire une partie du calcul à la compilation** : initialisation, précalcul
- **Définir un** *langage spécifique au domaine* : créer un langage adapté au problème (IA, jeux, musique, robotique, ...) pour développer les programmes plus rapidement/facilement
- **Attention aux abus** : programme plus difficile à comprendre/maintenir, impossible de passer des macros en paramètre à une fonction (e.g. `(apply and lst)`), code final plus gros

Capture de noms (1)

- Il faut se méfier de la **capture de noms**
- Exemple (utilisation d'un nom libre) :

```
> (define debug #t)
> (define-macro (? expr)
    `(if debug
         (let ((res ,expr))
           (pp (list ',expr '=> res))
           res)
         ,expr))
> (define (fact n)
    (if (? (= n 0))
        1
        (* n (fact (? (- n 1))))))
> (fact 3)
((= n 0) => #f)
((- n 1) => 2)
((= n 0) => #f)
((- n 1) => 1)
((= n 0) => #f)
((- n 1) => 0)
((= n 0) => #t)
6
```

Capture de noms (2)

- L'usage de la macro ? dans `fact` ne cause pas de problème de capture de nom :

```
> (set! debug #f)
> (fact 3)
6
> (pp fact)
(lambda (n)
  (if (if debug
          (let ((res (= n 0)))
            (pp (list '(= n 0) '=> res))
            res)
          (= n 0))
      1
      (* n
         (fact (if debug
                   (let ((res (- n 1)))
                     (pp (list '(- n 1) '=> res))
                     res)
                   (- n 1))))))
```

Capture de noms (3)

- Cependant, la mauvaise variable debug est accédée dans ce cas :

```
> (define (get-message)
    (let ((debug (open-input-file "debug.txt")))
      (let ((msg (read debug)))
        (close-input-port debug)
        (? (car msg)))))
> (pp get-message)
(lambda ()
  (let ((debug (open-input-file "debug.txt")))
    (let ((msg (read debug)))
      (close-input-port debug)
      (if debug
          (let ((res (car msg)))
            (pp (list '(car msg) '=> res))
            res)
          (car msg)))))
```

- Ici une définition de variable autour de l'appel de macro capture un nom libre généré par la macro

Capture de noms (4)

- Le problème de capture de nom peut survenir aussi lorsqu'une macro introduit des noms (variables locales, etc)

```
> (define-macro (begin1 expr1 . exprs)
    `(let ((x ,expr1)) ,@exprs x))
> (define n 0)
> (define (n++) (begin1 n (set! n (+ n 1))))
> (n++)
0
> (n++)
1
> (define (add x) (begin1 n (set! n (+ n x))))
> (add 5)
2
> (add 5)
4
> (add 5)
8
> (pp add)
(lambda (x) (let ((x n)) (set! n (+ n x)) x))
```

Éviter la capture de noms (1)

- Choisir des noms plus explicatifs, e.g. `debug` → `show-result-of-?-forms`
- Forcer l'évaluation dans le bon contexte, par exemple avec un *thunk*

```
> (define-macro (begin1 expr1 . exprs)
    `(let ((x ,expr1) (t (lambda () ,@exprs)))
       (t)
       x))
```

- Créer des symboles non-internés avec la fonction `gensym`

```
> (define-macro (begin1 expr1 . exprs)
    (let ((v (gensym)))
      `(let ((,v ,expr1)) ,@exprs ,v)))
> (define (add x) (begin1 n (set! n (+ n x))))
> (pp add)
(lambda (x)
  (let ((#:g1 n)) (set! n (+ n x)) #:g1))
```

Éviter la capture de noms (2)

- Les symboles non-internés sont distincts de tout autre symbole

```
> (define a (gensym))
> (define b (gensym))
> (define c (gensym))
> (symbol->string a)
"g1"
> (symbol->string b)
"g2"
> (symbol->string c)
"g3"
> (eq? a 'g1)
#f
> (eq? a a)
#t
> (list a b c)
(#:g1 #:g2 #:g3)
```

Capture de nom intentionnelle

- La capture de nom peut être voulue dans certains cas, tel que des macros qui étendent l'environnement d'évaluation avec de nouvelles liaisons
- Exemple :

```
> (define-macro (with expr . body)
    `(let ((it ,expr)) ,@body))
> (with (make-vector 3)
    (vector-set! it 0 'un)
    (vector-set! it 1 'deux)
    (vector-set! it 2 'trois)
    it)
#(un deux trois)
```

Macro while

- Une macro pour écrire des boucles while :

```
> (define-macro (while expr . body)
    (let ((sym (gensym)))
      `(let ,sym ()
         (if ,expr
             (let ()
               ,@body
               (,sym))))))
> (let ((i 0))
    (while (< i 5)
      (pp (list i (* i i)))
      (set! i (+ i 1))))
(0 0)
(1 1)
(2 4)
(3 9)
(4 16)
```

Étude de cas : hachage parfait (1)

- Objectif : structure de donnée pour représenter les **ensembles de chaînes** et une fonction pour tester rapidement si une chaîne est dans l'ensemble (e.g. ensemble des verbes, ensemble des noms de commande, etc)
- Implantation : ensemble = table d'adressage dispersée (hash table) avec **hachage parfait**
- Avec le hachage parfait il n'y a jamais de collision (i.e. deux chaînes dans l'ensemble ne donnent pas la même valeur de hachage)
- Cela permet de vérifier l'appartenance à l'ensemble sans recherche

Hachage parfait (2)


```
(define (hash-string str n)
  (let loop ((h 0) (i 0))
    (if (< i (string-length str))
        (loop (modulo
               (+ (* h 65536)
                  (char->integer
                   (string-ref str i)))
               n)
              (+ i 1))
        h)))

(define (make-htable n)
  (make-vector n #f))

(define (htable-member ht key)
  (let* ((n (vector-length ht))
         (i (hash-string key n))
         (x (vector-ref ht i)))
    (and x
         (string=? x key)
         i)))
```

Hachage parfait (3)


```
(define (htable-add! ht key)
  (let* ((n (vector-length ht))
         (i (hash-string key n))
         (x (vector-ref ht i)))
    (if x
        #f
        (begin
          (vector-set! ht i key)
          #t))))

(define (list->htable elems)
  (let loop1 ((n 1))
    (let ((ht (make-htable n)))
      (let loop2 ((lst elems))
        (cond ((null? lst)
               ht)
              ((htable-add! ht (car lst))
               (loop2 (cdr lst)))
              (else
               (loop1 (+ n 1)))))))))
```

Hachage parfait (4)


```
> (define ens
    (list->htable '("un" "deux" "trois")))
> ens
#("deux" #f "un" "trois")
> (htable-member ens "un")
2
> (htable-member ens "trois")
3
> (htable-member ens "quatre")
#f
> (list->htable '("one" "two" "three"))
#(#f "two" "one" "three" #f #f #f)
```

Hachage parfait (5)

- Si le programme contient des ensembles de chaînes qui sont constants (connus à la compilation du programme), il est avantageux de définir une macro pour créer ces ensembles

```
> (define-macro (ensemble . chaines)
    ''(list->htable chaines))
> (define ens
    (ensemble "un" "deux" "trois"))
> (htable-member ens "un")
2
```

- Le travail de construction de chaque ensemble constant se fait maintenant **à la compilation du programme** (plutôt qu'à chaque exécution du programme)

Étude de cas : pattern matching (1)

- Objectif : avoir une forme spéciale `match` similaire à `case` mais qui teste avec lequel de plusieurs gabarits (patterns) concorde le sujet
- `(match` $\langle expr_1 \rangle$ $\langle clause_1 \rangle$ $\langle clause_2 \rangle$`...)`
- $\langle clause \rangle$ ::= ($\langle gabarit \rangle$ $\langle expr_2 \rangle$)
- Exemple :

```
(match (list 'foo 4)
  ((bar 9)
   (pp "cas 1"))
  ((foo 4)
   (pp "cas 2"))) => imprime "cas 2"
```

Pattern matching (2)

- Quelle devrait être l'expansion de la macro?

```
(match (list 'foo 4)
  ((bar 9)
   (pp "cas 1"))
  ((foo 4)
   (pp "cas 2")))
```

→

```
(let ((s (list 'foo 4)))

  (define (f1)
    (if (equal? s '(bar 9))
        (pp "cas 1")
        (f2)))

  (define (f2)
    (if (equal? s '(foo 4))
        (pp "cas 2")
        (f3)))

  (define (f3)
    (error "match failed"))

  (f1))
```

Macro `match` de base (1)


```
(define-macro (match sujet . clauses)
  (let* ((var
          (gensym))
         (fns
          (map (lambda (x) (gensym))
               clauses))
         (err
          (gensym)))
    `(let ((,var ,sujet))
       ,@(map (lambda (fn1 fn2 clause)
                `(define (,fn1)
                   (if (equal? ,var
                               ',(car clause))
                       ,(cadr clause)
                       (,fn2))))
              fns
              (append (cdr fns) (list err))
              clauses)
       (define (,err) (error "match failed"))
       (,(car fns)))))
```

Macro `match` de base (2)


```
> (pp (lambda () (match E0 (a E1) (b E2))))
(lambda ()
  (let ((#:g0 E0))
    (letrec ((#:g1
              (lambda ()
                (if (equal? #:g0 'a)
                    E1
                    (#:g2))))
             (#:g2
              (lambda ()
                (if (equal? #:g0 'b)
                    E2
                    (#:g3))))
             (#:g3
              (lambda ()
                (error "match failed"))))
      (#:g1))))
```

Intégrer `equal?` (1)

- Pour accélérer le test `(equal? s '(foo 4))` où un des paramètres est constant on peut intégrer les tests que fera `equal?`

```
(if (equal? s '(foo 4)) X Y) =

(if (pair? s)
    (let ((a (car s)))
      (if (eq? a 'foo)
          (let ((b (cdr s)))
            (if (pair? b)
                (let ((c (car b)))
                  (if (eqv? c 4)
                      (let ((d (cdr b)))
                        (if (null? d)
                            X
                            Y))
                      Y))
                Y))
          Y))
    Y)
```

Intégrer `equal?` (2)

- Pour genérer les tests de

  `(if (equal?` var `'`gab`)` oui non`)`

  fonction

  `(if-equal?` `'`var `'`gab `'`oui `'`non`)`

- En d'autres termes :

  si `(if-equal?` `'`var `'`gab `'`oui `'`non`)` $\rightarrow E$

  alors `(if (equal?` var `'`gab`)` oui non`)` $= E$

Intégrer equal? (3)


```
(define (if-equal? var gab oui non)
  (cond ((null? gab)
         `(if (null? ,var) ,oui ,non))
        ((symbol? gab)
         `(if (eq? ,var ',gab) ,oui ,non))
        ((number? gab)
         `(if (eqv? ,var ,gab) ,oui ,non))
        ((pair? gab)
         (let ((carvar (gensym))
               (cdrvar (gensym)))
           `(if (pair? ,var)
                (let ((,carvar (car ,var)))
                  ,(if-equal?
                    carvar
                    (car gab)
                    `(let ((,cdrvar (cdr ,var)))
                       ,(if-equal?
                         cdrvar
                         (cdr gab)
                         oui
                         non))
                    non))
                ,non)))
        (else
         (error "unknown pattern"))))
```

Intégrer `equal?` (4)


```
> (if-equal? 'a '(foo 4) 'X 'Y)
(if (pair? a)
    (let ((#:g0 (car a)))
      (if (eq? #:g0 'foo)
          (let ((#:g1 (cdr a)))
            (if (pair? #:g1)
                (let ((#:g2 (car #:g1)))
                  (if (eqv? #:g2 4)
                      (let ((#:g3 (cdr #:g1)))
                        (if (null? #:g3)
                            X
                            Y))
                      Y))
                Y))
          Y))
    Y)
```

Intégrer equal? (5)


```
(define-macro (match sujet . clauses)

  (define (if-equal? var gab oui non) ...)

  (let* ((var
          (gensym))
         (fns
          (map (lambda (x) (gensym))
               clauses))
         (err
          (gensym)))
    `(let ((,var ,sujet))
       ,@(map (lambda (fn1 fn2 clause)
                `(define (,fn1)
                   ,(if-equal? var
                               (car clause)
                               (cadr clause)
                               `(,fn2))))
              fns
              (append (cdr fns) (list err))
              clauses)
       (define (,err) (error "match failed"))
       (,(car fns)))))
```

Intégrer equal? (6)


```
> (pp (lambda ()
        (match (list 'foo 4)
          ((foo 9) (pp "cas 1"))
          ((foo 4) (pp "cas 2")))))
(lambda ()
  (let ((#:g0 (list 'foo 4)))
    (letrec ((#:g1 (lambda ()
                     (if (pair? #:g0)
                         (let ((#:g4 (car #:g0)))
                           (if (eq? #:g4 'foo)
                               (let ((#:g5 (cdr #:g0)))
                                 (if (pair? #:g5)
                                     (let ((#:g6 (car #:g5)))
                                       (if (eqv? #:g6 9)
                                           (let ((#:g7 (cdr #:g5)))
                                             (if (null? #:g7)
                                                 (pp "cas 1")
                                                 (#:g2)))
                                           (#:g2)))
                                     (#:g2)))
                               (#:g2)))
                         (#:g2))))
             (#:g2 (lambda ()
                     (if (pair? #:g0)
                         (let ((#:g8 (car #:g0)))
                           (if (eq? #:g8 'foo)
                               (let ((#:g9 (cdr #:g0)))
                                 (if (pair? #:g9)
                                     (let ((#:g10 (car #:g9)))
                                       (if (eqv? #:g10 4)
                                           (let ((#:g11 (cdr #:g9)))
                                             (if (null? #:g11)
                                                 (pp "cas 2")
                                                 (#:g3)))
                                           (#:g3)))
                                     (#:g3)))
                               (#:g3)))
                         (#:g3)))))
      (#:g1))))
```

Deuxième approche (1)

- Objectif : remplacer cascade de `if` par `and`

```
> (if-equal? 'a '(foo 4) 'X 'Y)
(if (and (pair? a)
         (eq? (car a) 'foo)
         (pair? (cdr a))
         (eqv? (car (cdr a)) 4)
         (null? (cdr (cdr a))))
    X
    Y)
```

Deuxième approche (2)


```
(define (conditions exp gab)
  (cond ((null? gab)
         (list '(null? ,exp)))
        ((symbol? gab)
         (list '(eq? ,exp ',gab)))
        ((number? gab)
         (list '(eqv? ,exp ,gab)))
        ((pair? gab)
         (append (list '(pair? ,exp))
                 (conditions '(car ,exp) (car gab))
                 (conditions '(cdr ,exp) (cdr gab))))
        (else
         (error "unknown pattern"))))

(conditions 'a '(foo 4)) => ((pair? a)
                             (eq? (car a) 'foo)
                             (pair? (cdr a))
                             (eqv? (car (cdr a)) '4)
                             (null? (cdr (cdr a)))))
```

Deuxième approche (3)

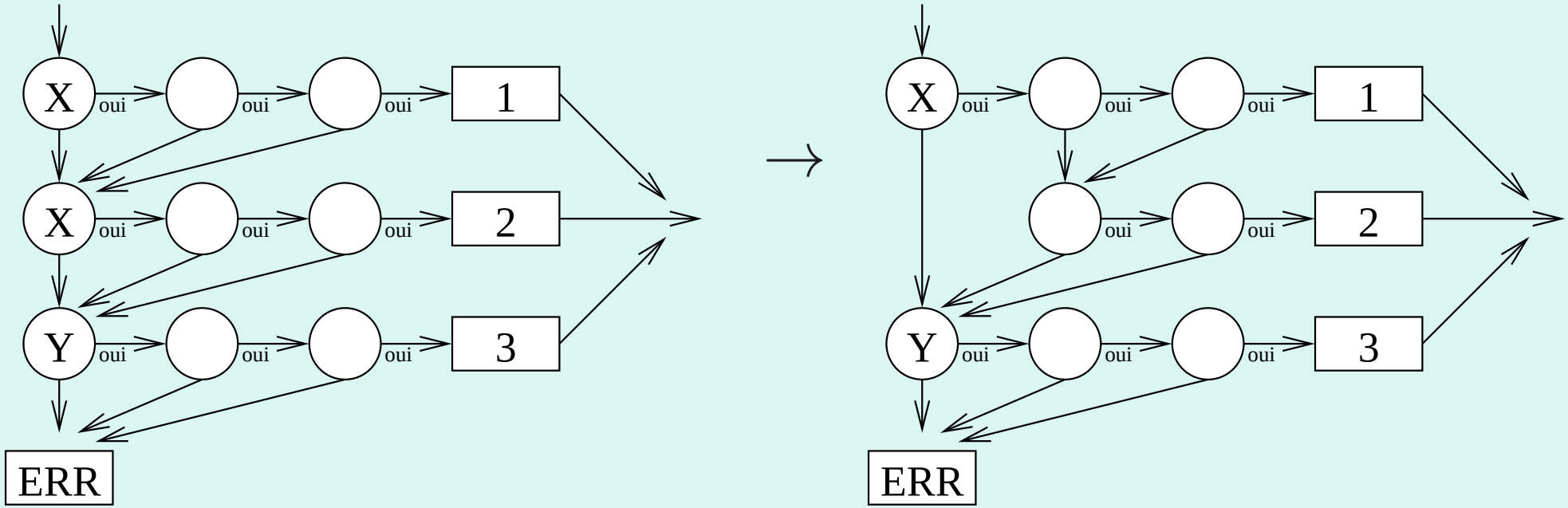

```
(define-macro (match sujet . clauses)

  (define (conditions exp gab) ...)

  (define (if-equal? var gab oui non)
    `(if (and ,@(conditions var gab))
          ,oui
          ,non))

  (define (gen var clauses)
    (if (pair? clauses)
        (let ((clause (car clauses)))
          (if-equal? var
                     (car clause)
                     (cadr clause)
                     (gen var (cdr clauses))))
        `(error "match failed")))

  (let ((var (gensym)))
    `(let ((,var ,sujet))
       ,(gen var clauses))))
```

Deuxième approche (4)


```
> (pp (lambda ()
        (match (list 'foo 4)
          ((foo 9) (pp "cas 1"))
          ((foo 4) (pp "cas 2")))))
(lambda ()
  (let ((#:g0 (list 'foo 4)))
    (cond ((and (pair? #:g0)
                (eq? (car #:g0) 'foo)
                (pair? (cdr #:g0))
                (eqv? (car (cdr #:g0)) 9)
                (null? (cdr (cdr #:g0))))
           (pp "cas 1"))
          ((and (pair? #:g0)
                (eq? (car #:g0) 'foo)
                (pair? (cdr #:g0))
                (eqv? (car (cdr #:g0)) 4)
                (null? (cdr (cdr #:g0))))
           (pp "cas 2"))
          (else
           (error "match failed")))))
```

Deuxième approche (5)

- Objectif : factoriser les tests communs aux cas
- Transformation :



Deuxième approche (6)


```
(define (split f lst)
  (if (null? lst)
      '(() . ())
      (let ((x (car lst)))
        (if (f x)
            (let ((s (split f (cdr lst))))
              (cons (cons x (car s))
                    (cdr s)))
            (cons '() lst)))))

(split odd? '(3 1 4 1 5)) => ((3 1) . (4 1 5))
```

Deuxième approche (7)


```
(define-macro (match sujet . clauses)

  (define (split f lst) ...)

  (define (gen ces err) ...)

  (let ((var (gensym)))
    `(let ((,var ,sujet))
       ,(gen (map (lambda (clause)
                    (cons (conditions
                           var
                           (car clause))
                          (cadr clause)))
                  clauses)
             `(error "match failed"))))))
```

Deuxième approche (8)


```
(define (gen ces err)
  (if (null? ces)
      err
      (let* ((ce (car ces))
             (cnds (car ce))
             (expr (cdr ce)))
        (if (null? cnds)
            expr
            (let* ((cnd1 (car cnds))
                   (s    (split (lambda (ce)
                                  (let ((c (car ce)))
                                    (and (pair? c)
                                         (equal? (car c)
                                                 cnd1))))
                                ces))
                   (ces1 (car s))
                   (ces2 (cdr s))
                   (err2 (gen ces2 err)))
              `(if ,cnd1
                   ,(gen (map (lambda (ce)
                                (cons (cdr (car ce))
                                      (cdr ce)))
                              ces1)
                         err2)
                   ,err2)))))))
```

Deuxième approche (9)


```
> (pp (lambda ()
        (match (list 'foo 4)
          ((foo 9) (pp "cas 1"))
          ((foo 4) (pp "cas 2")))))
(lambda ()
  (let ((#:g0 (list 'foo 4)))
    (if (pair? #:g0)
        (if (eq? (car #:g0) 'foo)
            (if (pair? (cdr #:g0))
                (if (eqv? (car (cdr #:g0)) 9)
                    (if (null? (cdr (cdr #:g0)))
                        (pp "cas 1")
                        (if (eqv? (car (cdr #:g0)) 4)
                            (if (null? (cdr (cdr #:g0)))
                                (pp "cas 2")
                                (error "match failed"))
                            (error "match failed")))
                    (if (eqv? (car (cdr #:g0)) 4)
                        (if (null? (cdr (cdr #:g0)))
                            (pp "cas 2")
                            (error "match failed"))
                        (error "match failed")))
                (error "match failed"))
            (error "match failed"))
        (error "match failed"))))
```

Extension: variables (1)

- Notation inspirée de la forme `quasiquote` : `,`var indique une partie variable du gabarit (i.e. qui concorde avec n'importe quoi dans le sujet) :

  `(foo ,x 4)`

  concorde avec une liste de 3 éléments :
  - 1er = symbole `foo`
  - 2em = n'importe quoi
  - 3em = entier `4`

- `x` prendra la valeur du 2em élément

- Note : `,`X = `(unquote` X`)`

Extension: variables (2)

- Modification de `if-equal?` :

```
(define (if-equal? var gab oui non)
  (cond ((and (pair? gab)
              (eq? (car gab) 'unquote)
              (pair? (cdr gab))
              (null? (cddr gab)))
         `(let ((,(cadr gab) ,var))
            ,oui))
        ((null? gab)
         `(if (null? ,var) ,oui ,non))

        ...etc))
```

- Le `let` qui est généré crée une liaison de la variable qui est visible dans l'expression de cette clause

Extension: variables (3)


```
> (define f (lambda (x)
              (match x
                ((add ,v1 ,v2)
                 (+ v1 v2)))))
> (f '(add 3 4))
7
> (pp f)
(lambda (x)
  (let ((#:g0 x))
    (letrec ((#:g1
              (lambda ()
                (if (pair? #:g0)
                    (let ((#:g3 (car #:g0)))
                      (if (eq? #:g3 'add)
                          (let ((#:g4 (cdr #:g0)))
                            (if (pair? #:g4)
                                (let ((#:g5 (car #:g4)))
                                  (let ((v1 #:g5))
                                    (let ((#:g6 (cdr #:g4)))
                                      (if (pair? #:g6)
                                          (let ((#:g7 (car #:g6)))
                                            (let ((v2 #:g7))
                                              (let ((#:g8 (cdr #:g6)))
                                                (if (null? #:g8)
                                                    (+ v1 v2)
                                                    (#:g2)))))
                                          (#:g2)))))
                                (#:g2)))
                          (#:g2)))
                    (#:g2)))))
      (#:g2
       (lambda () (error "match failed"))))
     (#:g1))))
```

Extension: variables (4)


```
> (define f (lambda (x)
              (match x
                ((add . ,lst)
                 (apply + lst)))))
> (f '(add 1 2 3))
6
> (pp f)
(lambda (x)
  (let ((#:g0 x))
    (letrec ((#:g1
              (lambda ()
                (if (pair? #:g0)
                    (let ((#:g3 (car #:g0)))
                      (if (eq? #:g3 'add)
                          (let ((#:g4 (cdr #:g0)))
                            (let ((lst #:g4))
                              (apply + lst)))
                          (#:g2)))
                    (#:g2))))
             (#:g2
              (lambda () (error "match failed"))))
      (#:g1))))
```

Extension: variables (5)


```
> (define (calc expr)
    (match expr
      ((add . ,lst) (apply + (map calc lst)))
      ((sub . ,lst) (apply - (map calc lst)))
      ((mul . ,lst) (apply * (map calc lst)))
      ((div . ,lst) (apply / (map calc lst)))
      ((sqrt ,x)    (sqrt (calc x)))
      ((sin ,x)     (sin (calc x)))
      ((cos ,x)     (cos (calc x)))
      ((tan ,x)     (tan (calc x)))
      (pi           3.1415926)
      (,x           x)))
> (calc '(mul 2 (sqrt pi)))
3.5449076715762287
```

Extension: gardes (1)

- Garde = **condition supplémentaire** à vérifier
- ⟨*clause*⟩ ::= (⟨*gabarit*⟩ [when ⟨*garde*⟩] ⟨$expr_2$⟩)
- Exemple :

```
> (define (f x)
    (match x
      ((foo ,y) when (< y 0)
       (list 'neg y))
      ((foo ,y) when (> y 0)
       (list 'pos y))))
> (f '(foo 8))
(pos 8)
> (f '(foo -8))
(neg -8)
```

Extension: gardes (2)


```
(define-macro (match sujet . clauses)
  (let* ((var
          (gensym))
         (fns
          (map (lambda (x) (gensym))
               clauses))
         (err
          (gensym)))
    `(let ((,var ,sujet))
       ,@(map (lambda (fn1 fn2 clause)
                `(define (,fn1)
                   ,(if-equal? var
                               (car clause)
                               (if (and (eq? (cadr clause) 'when)
                                        (pair? (cddr clause)))
                                   `(if ,(caddr clause)
                                        ,(cadddr clause)
                                        (,fn2))
                                   (cadr clause))
                               `(,fn2))))
              fns
              (append (cdr fns) (list err))
              clauses)
       (define (,err) (error "match failed"))
       (,(car fns))))))
```

Exemple (1)

- Implantation de la syntaxe complète de la forme `cond`
- `(cond` $\langle clause_1 \rangle$ $\langle clause_2 \rangle$ `...)`
- $\langle clause \rangle$ ::= `(`$\langle expr_1 \rangle$`)`
- $\langle clause \rangle$ ::= `(`$\langle expr_1 \rangle$ `=>` $\langle expr_2 \rangle$`)`
- $\langle clause \rangle$ ::= `(`$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ $\langle expr_3 \rangle$ `...)`
- $\langle clause \rangle$ ::= `(else` $\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `...)`
- Les clauses `else` peuvent seulement apparaitre comme dernière clause d'une forme `cond`

Exemple (2)

- Exemple :

```
(define (f x)
  (cond ((< x 10))
        ((> x 20) "big")
        ((odd? x) (display x) "odd")
        ((assoc x lst) => cdr)
        (else (display "wow") 999)))

(define lst '((12 . douze) (16 . seize)))

(f 4)  =>  #t
(f 22) =>  "big"
(f 11) =>  "odd"  (et affiche 11)
(f 16) =>  seize
(f 14) =>  999    (et affiche wow)
```

Exemple (3)


```
(define-macro (cond . clauses)
  (match (cons 'cond clauses)
    ((cond)
     '#f)
    ((cond (else ,e1 . ,es))
     '(begin ,e1 ,@es))
    ((cond (else . ,es) . ,rest)
     (error "improper else clause"))
    ((cond (,test) . ,rest)
     '(or ,test (cond ,@rest)))
    ((cond (,test => ,fn) . ,rest)
     (let ((v (gensym)))
       '(let ((,v ,test))
          (if ,v
              (,fn ,v)
              (cond ,@rest)))))
    ((cond (,test => . ,es) . ,rest)
     (error "improper => clause"))
    ((cond (,test ,e1 . ,es) . ,rest)
     '(if ,test
          (begin ,e1 ,@es)
          (cond ,@rest)))))
```

Environnement lexical (1)

- Un **environnement** associe des valeurs à des noms
- Étant donné un nom et un environnement on peut retrouver la valeur associée (**“lookup”**)
- Nous allons étudier les environnements dans le contexte de l’interprétation et compilation de ce sous-ensemble de Scheme:

$$\begin{array}{lll} \langle \textit{expr} \rangle & ::= & C \\ & | & V \\ & | & \texttt{(lambda (}V\texttt{...)} \quad \langle \textit{expr} \rangle \texttt{)} \\ & | & \texttt{(}\langle \textit{expr} \rangle \ \langle \textit{expr} \rangle \texttt{...)} \end{array}$$

`cons`, `+`, etc sont des variables prédéfinies

Environnement lexical (2)

- Dans un premier temps, simplifions encore plus le sous-ensemble pour retirer les lambda-expressions:

$$\begin{array}{lll} \langle expr \rangle & ::= & C \\ & | & V \\ & | & (\langle expr \rangle \ \langle expr \rangle \ldots) \end{array}$$

Environnement lexical (3)

- En Scheme, la **liste d'association** est une représentation simple pour les environnements:

```
(define genv                    ;; environnement global
  (list (cons 'cons cons)
        (cons 'car car)
        (cons 'cdr cdr)
        (cons '+ +)))

(define (env-lookup env var)
  (let ((x (assq var env)))
    (if x (cdr x) (error "unbound" var))))



(env-lookup genv 'cons) => #<procedure cons>

(env-lookup genv 'x)    => erreur: unbound x
```

Environnement lexical (4)


```
(define (ev expr)  ;; interprète sans lambda
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     (env-lookup genv var))

    ((,fun . ,exprs)
     (apply (ev fun)
            (map (lambda (x) (ev x))
                 exprs)))))

(define (constant? x)
  (or (number? x) (string? x) (boolean? x)))

(define (variable? x)
  (symbol? x))

(define (eval expr)
  (ev expr))


(eval '(cons 1 2)) => (1 . 2)
```

Environnement lexical (5)

- Pour interpréter les lambda-expressions, il faut une fonction pour **étendre** un environnement avec des nouvelles associations de noms et valeurs:

```
(define (env-extend env vars vals)
  (append (map cons vars vals) env))

(env-lookup genv 'cons) => #<procedure cons>

(env-lookup genv 'j) => erreur: unbound j

(define e (env-extend genv '(j cons) '(1 2)))

(env-lookup e 'j) => 1

(env-lookup e 'cons) => 2
```

Environnement lexical (6)


```
(define (ev expr env)  ;; interprète avec env
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     (env-lookup env var))

    ((lambda ,params ,body)
     (lambda args (ev body (env-extend env params args))))

    ((,fun . ,exprs)
     (apply (ev fun env)
            (map (lambda (x) (ev x env))
                 exprs)))))

(define (eval expr)
  (ev expr genv))


(eval '((lambda (x y) (cons y x)) 1 2)) => (2 . 1)
```

Environnement lexical (7)

- Alternative: maintenir séparément un **environnement de compilation** et un **environnement d'exécution**

```
(define gcte (list 'cons 'car 'cdr '+))
(define grte (list  cons  car  cdr  +))

(define (cte-extend cte vars) (append vars cte))
(define (rte-extend rte vals) (append vals rte))

(define (cte-lookup cte var)
  (let ((x (memq var cte)))
    (if x
        (- (length cte) (length x))
        (error "unbound" var))))

(define (rte-lookup rte pos)
  (list-ref rte pos))


(define pos (cte-lookup gcte '+))
(rte-lookup grte pos) => #<procedure +>
```

Environnement lexical (8)


```
(define (ev expr cte rte)  ;; interprète avec cte/rte
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     (let ((pos (cte-lookup cte var)))
       (rte-lookup rte pos)))

    ((lambda ,params ,body)
     (lambda args (ev body (cte-extend cte params)
                           (rte-extend rte args))))

    ((,fun . ,exprs)
     (apply (ev fun cte rte)
            (map (lambda (x) (ev x cte rte))
                 exprs)))))

(define (eval expr)
  (ev expr gcte grte))
```

Environnement lexical (9)


```
(define (ev expr cte)  ;; interprète currifié
  (lambda (rte)
    (match expr

      (,const when (constant? const)
       const)

      (,var when (variable? var)
       (let ((pos (cte-lookup cte var)))
         (rte-lookup rte pos)))

      ((lambda ,params ,body)
       (lambda args ((ev body (cte-extend cte params))
                     (rte-extend rte args))))

      ((,fun . ,exprs)
       (apply ((ev fun cte) rte)
              (map (lambda (x) ((ev x cte) rte))
                   exprs))))))

(define (eval expr)
  ((ev expr gcte) grte))
```

Environnement lexical (10)


```
(define (ev expr cte)  ;; interprète rapide
  (match expr

    (,const when (constant? const)
     (lambda (rte) const))

    (,var when (variable? var)
     (let ((pos (cte-lookup cte var)))
       (lambda (rte) (rte-lookup rte pos))))

    ((lambda ,params ,body)
     (let ((b (ev body (cte-extend cte params))))
       (lambda (rte)
         (lambda args (b (rte-extend rte args))))))

    ((,fun . ,exprs)
     (let ((f (ev fun cte))
           (es (map (lambda (x) (ev x cte)) exprs)))
       (lambda (rte)
         (apply (f rte)
                (map (lambda (e) (e rte)) es)))))))

(define (eval expr)
  ((ev expr gcte) grte))
```

Environnement lexical (11)


```
(define const1 (lambda (rte) 1))
(define const2 (lambda (rte) 2))
(define pos0 (lambda (rte) (car rte)))
(define pos1 (lambda (rte) (cadr rte)))

(define (ev expr cte)  ;; interprète rapide optimisé
  (match expr

    (,const when (constant? const)
     (case const
       ((1) const1)
       ((2) const2)
       (else
        (lambda (rte) const))))

    (,var when (variable? var)
     (let ((pos (cte-lookup cte var)))
       (case pos
         ((0) pos0)
         ((1) pos1)
         (else
          (lambda (rte) (rte-lookup rte pos)))))))

    ...))
```

Environnement lexical (12)


```
;; Environnements chaines:

(define gcte (list (list  'cons 'car 'cdr '+)))
(define grte (list (vector cons  car  cdr  +)))

(define (cte-extend cte vars) (cons vars cte))
(define (rte-extend rte vals) (cons (list->vector vals) rte

(define (cte-lookup cte var)
  (let loop ((cte cte) (up 0))
    (if (null? cte)
        (error "unbound" var)
        (let ((x (memq var (car cte))))
          (if x
              (cons up (- (length (car cte)) (length x)))
              (loop (cdr cte) (+ up 1)))))))

(define (rte-lookup rte pos)
  (vector-ref (list-ref rte (car pos)) (cdr pos)))
```

Conversion de fermeture (1)

- Caractéristiques pertinentes de Scheme :
  - Portée lexicale
  - Étendue indéfinie des fonctions (une instance de variable peut survivre l'activation de fonction qui l'a créée)
- Solutions :
  - **Fermetures** : mémorisent l'environnement du contexte de création de la fonction
  - **Garbage collector** : pour récupérer les environnements qui ne sont plus utiles au calcul

Conversion de fermeture (2)

- Exemple de fermetures :

```
(define (keep f lst)
  (cond ((null? lst)
         '())
        ((f (car lst))
         (cons (car lst) (keep f (cdr lst))))
        (else
         (keep f (cdr lst)))))

(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y) (f x y)))))

(define swap2
  (lambda (g)
    (lambda (x y) (g y x))))

(define gt (curry2 (swap2 >)))

(keep (gt 3) '(3 1 4 1 5)) => (4 5)
```

Conversion de fermeture (3)

- Dans une expression E, une référence à une variable V est :
  - **liée** (“bound”) si V est déclarée dans E
  - **libre** (“free”) si V n’est pas déclarée dans E

- Exemple : dans `(lambda (y) (+ x y))`
  - x et + sont libres
  - y est liée

- Les variables libres de E sont donc les variables dans l’environnement d’évaluation de E qui “paramétrisent” l’expression E

Conversion de fermeture (4)

- Analyse des variables libres pour le noyaux de Scheme :

$$FV(\ c\ ) = \{\}$$

$$FV(\ v\ ) = \{v\}$$

$$FV(\ \texttt{(lambda (}p_1 ...\texttt{)}\ E\texttt{)}\ ) = FV(E) \setminus \{p_1, ...\}$$

$$FV(\ \texttt{(}E_0\ E_1 ...\texttt{)}\ ) = FV(E_0) \cup FV(E_1) \cup ...$$

- Exemple :

$$FV(\ \texttt{(lambda (y) (+ x y))}\ ) =$$

$$FV(\ \texttt{(+ x y)}\ ) \setminus \{\texttt{y}\} =$$

$$FV(\ \texttt{+}\ ) \cup FV(\ \texttt{x}\ ) \cup FV(\ \texttt{y}\ ) \setminus \{\texttt{y}\} =$$

$$\{\texttt{+}, \texttt{x}, \texttt{y}\} \setminus \{\texttt{y}\} = \{\texttt{+}, \texttt{x}\}$$

Conversion de fermeture (5)


```
(define (fv expr)
  (match expr

    (,const when (constant? const)
     '())

    (,var when (variable? var)
     (list var))

    ((set! ,var ,E1)
     (union (list var) (fv E1)))

    ((if ,E1 ,E2)
     (union (fv E1) (fv E2)))
    ((if ,E1 ,E2 ,E3)
     (union (fv E1) (union (fv E2) (fv E3))))

    ((lambda ,params ,E)
     (difference (fv E) params))

    ...))
```

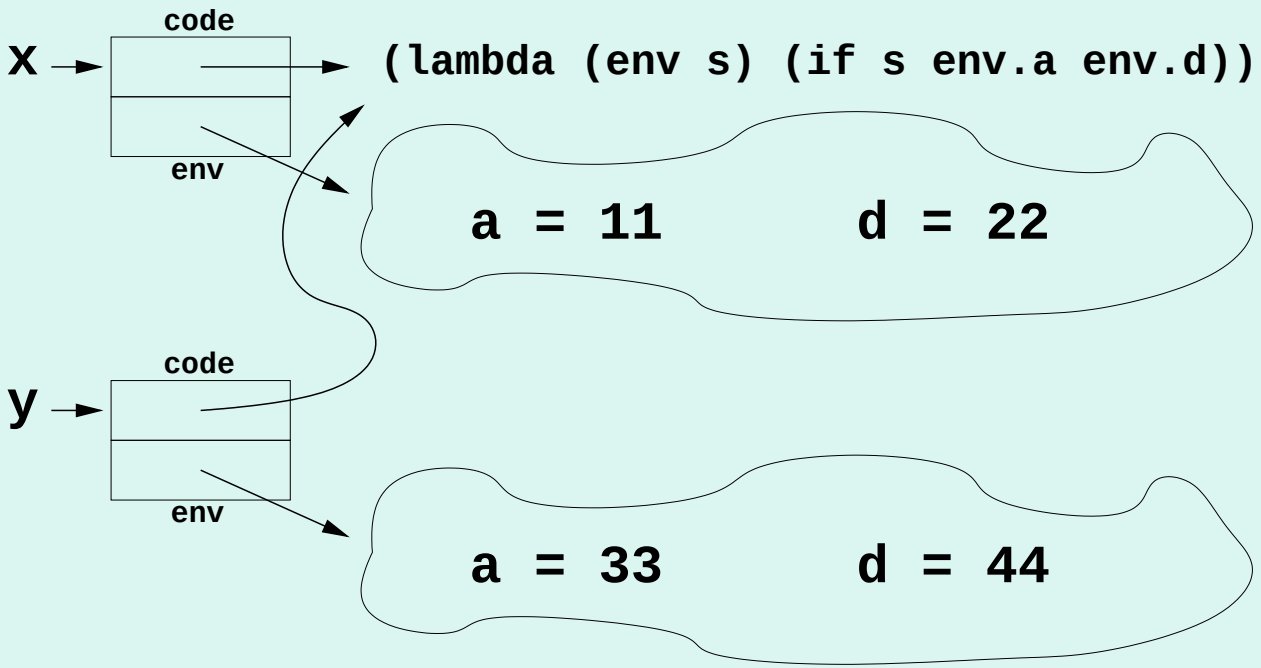
Conversion de fermeture (6)

- Soit une lambda-expression E dont l'évaluation donne une fermeture F
- $FV(E)$ sont les variables de l'environnement d'évaluation de E qui sont (possiblement) utilisées pour éxécuter le corps de F
- Exemple :

```
(define (kons a d) (lambda (s) (if s a d)))
(define (kar p) (p #t))
(define (kdr p) (p #f))

; FV( (lambda (s) (if s a d)) ) = {a,d}

(define x (kons 11 22))
(define y (kons 33 44))

(kar x) => 11
(kar y) => 33
(kdr y) => 44
```

Conversion de fermeture (7)

- Représentation typique des fermetures : objet qui contient un **pointeur vers le code exécutable** de la fermeture et l'**environnement** qui contient au moins les variables libres




```
(p #t) -> (p.code p.env #t)
```

Conversion de fermeture (8)

- Représentation typique des environnements :
  - **Chaîne lexicale** : environnement = liste des blocs d'activation englobants
  - **"Display"** : environnement = vecteur des blocs d'activation englobants
  - **Fermeture plate** : stocker les variables libres directement dans la fermeture

Conversion de fermeture (9)

- Pour expliquer l'implantation tout en restant en Scheme, on va supposer une transformation de code qui **crée explicitement les blocs d'activation** des fonctions

1. `(f X Y Z)` $\rightarrow$ `(f (vector X Y Z))`
2. `(lambda (`$...p_i...$`) `$...p_i...$`)` $\rightarrow$
   `(lambda (`a`) ...(vector-ref `a $i-1$`)...)`

- Le compilateur doit se souvenir dans quel bloc d'activation (a) une variable p_i est déclarée en plus de sa position dans le bloc (i)

Conversion de fermeture (10)


```
(lambda (a0 a1 a2)
  ...
    (lambda (b0 b1)
      ...
        (lambda (c0 c1 c2)
          ...
            (lambda (x) (+ x c0 b1 a2))
          ...)
      ...)
  ...)
```

devient

```
(lambda (a)
  ...
    (lambda (b)
      ...
        (lambda (c)
          ...
            (lambda (d) (+ (vector-ref d 0)
                           (vector-ref c 0)
                           (vector-ref b 1)
                           (vector-ref a 2)))
          ...)
      ...)
  ...)
```

Conversion de fermeture (11)

- Cette façon de faire nous rapproche du langage machine
- En effet, puisque toutes les fonctions ont toujours un seul paramètre (le bloc d'activation courant) on peut utiliser **un registre machine** pour implanter le protocole d'appel de fonction

Conversion de fermeture (12)

- L'approche par chaîne lexicale représente l'environnement comme une **liste de blocs d'activation**
- Chaque fermeture mémorise l'environnement du **contexte d'évaluation** de la lambda-expression

Conversion de fermeture (13)

- Le code de la fermeture reçoit l'environnement en paramètre

```
(define (make-closure code env) (vector code env))
(define (closure-code clo) (vector-ref clo 0))
(define (closure-env clo) (vector-ref clo 1))

(lambda (a) ...) -> (make-closure
                     (lambda (env a) ...)
                     ENV)

(f (vector X Y)) -> ((closure-code f)
                     (closure-env f)  ;; -> env
                     (vector X Y))    ;; -> a
```

Conversion de fermeture (14)

- À noter que maintenant les fonctions ont toujours 2 paramètres

- On peut donc maintenant utiliser 2 registres machine pour implanter le protocole d'appel de fonction (un pour l'**environnement** et l'autre pour le **bloc d'activation courant**)

Conversion de fermeture (15)


```
(make-closure
 (lambda (env a)
   ...
    (make-closure
     (lambda (env b)
       ...
        (make-closure
         (lambda (env c)
           ...
           (make-closure
            (lambda (env d)
              (+ (vector-ref d 0)
                 (vector-ref (car env) 0)
                 (vector-ref (cadr env) 1)
                 (vector-ref (caddr env) 2)))
            (cons c env))
           ...)
         (cons b env))
       ...)
     (cons a env))
  ...)
 env-global)
```

Conversion de fermeture (16)

- Le temps d'accès à une variable dépend de sa profondeur dans la chaîne (pire cas = variables les plus globales)
- L'approche par "display" représente l'environnement comme un **vecteur des blocs d'activation**
- Le temps d'accès à toute variable est constant
- Cependant le temps de création des environnements n'est plus constant (pire cas = lambda-expression très imbriquée)

Conversion de fermeture (17)


```
(make-closure
 (lambda (env a)
   ...
    (make-closure
     (lambda (env b)
       ...
        (make-closure
         (lambda (env c)
           ...
           (make-closure
            (lambda (env d)
              (+ (vector-ref d 0)
                 (vector-ref (vector-ref env 0) 0)
                 (vector-ref (vector-ref env 1) 1)
                 (vector-ref (vector-ref env 2) 2)))
            (vector c
                    (vector-ref env 0)
                    (vector-ref env 1)
                    (vector-ref env 2)))
           ...)
         (vector b (vector-ref env 0) (vector-ref env 1)))
       ...)
     (vector a (vector-ref env 0)))
   ...)
 (vector env-global))
```

Conversion de fermeture (18)

- Exemple de la fonction `curry2` :

```
(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y) (f x y)))))

devient...
```

Conversion de fermeture (19)


```
(define curry2
  (make-closure
   (lambda (env a)
     (make-closure
      (lambda (env b)
        (make-closure
         (lambda (env c)
           (let ((f (vector-ref (vector-ref env 1) 0)))
             ((closure-code f)
              (closure-env f)
              (vector (vector-ref (vector-ref env 0) 0)
                      (vector-ref c 0)))))
         (vector b (vector-ref env 0) (vector-ref env 1))))
      (vector a (vector-ref env 0))))
   (vector env-global)))
```

Conversion de fermeture (20)

- Les environnements **chainés** et avec **“display”** sont similaires : l’environnement contient l’ensemble des blocs d’activation des lambda-expressions englobantes

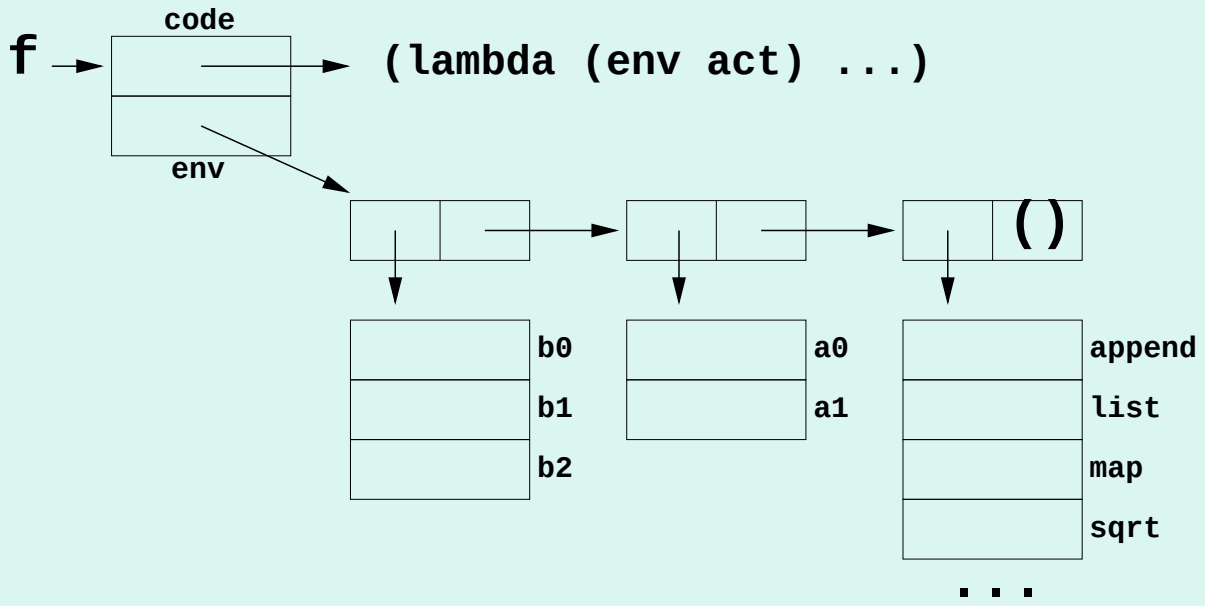
- Soit le code source suivant :

```
(lambda (a0 a1)
  ...
    (lambda (b0 b1 b2)
      ...
        (let ((f (lambda (c0)
                   (list c0 b1 b2))))
          ...)
      ...)
  ...)
```

- Quelle est la valeur de `f` avec chaque représentation?

Conversion de fermeture (21)

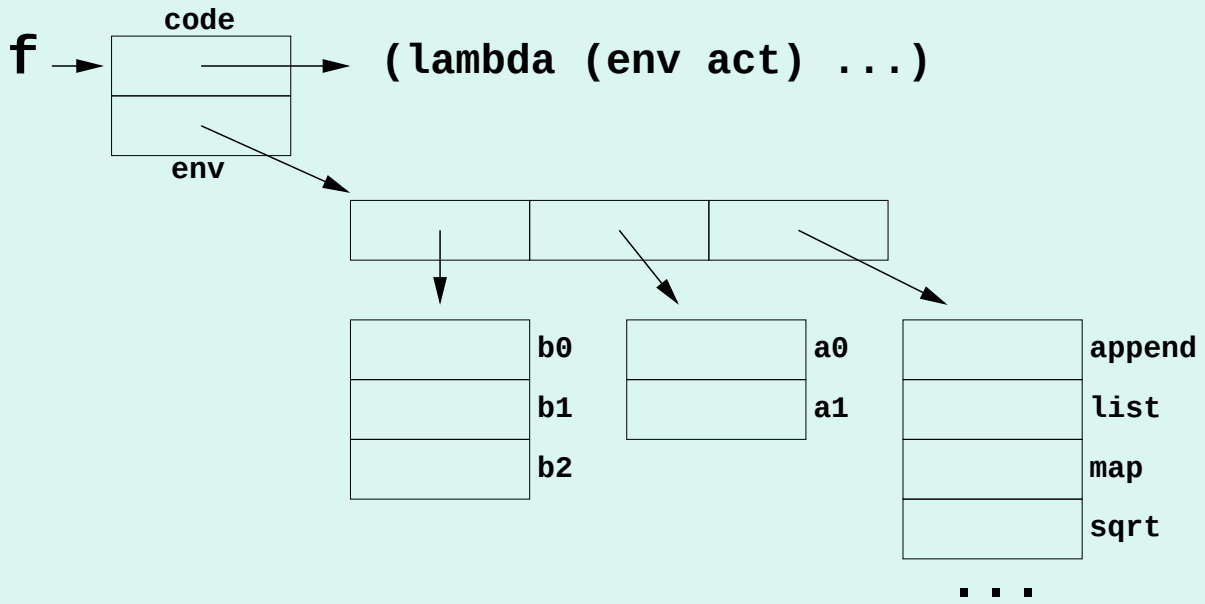
- Fermeture avec environnement chaîné :




```
(lambda ...) -> (make-closure
                  (lambda (env act) ...)
                  (cons act env))

(F X Y Z)    -> (let ((clo F))
                  ((closure-code clo)
                   (closure-env clo)
                   (vector X Y Z)))

(define env '())
(define act (vector append list ...))
```

Conversion de fermeture (22)

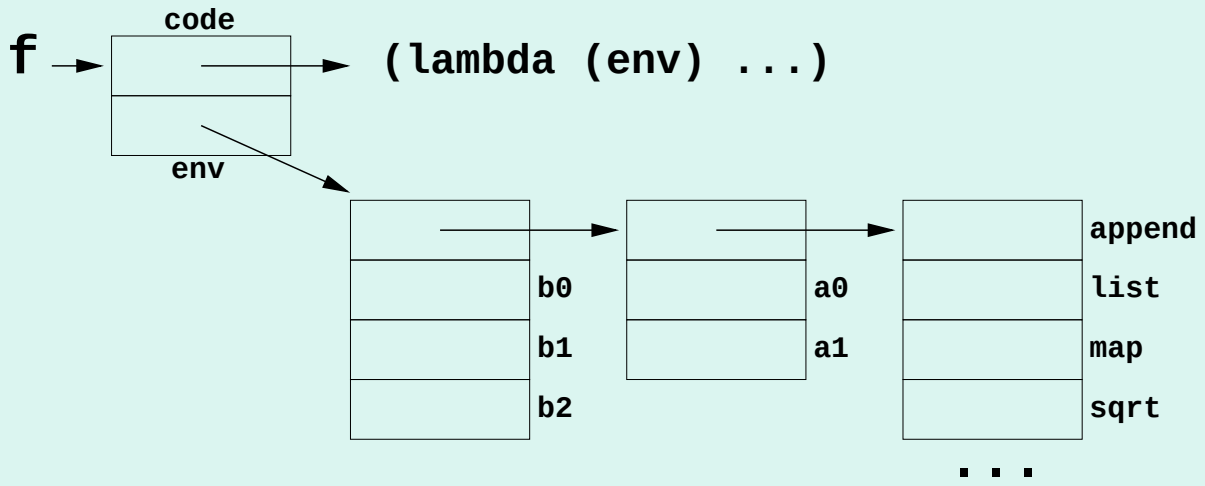
- Fermeture avec environnement “display” :




```
(lambda ...) -> (make-closure
                  (lambda (env act) ...)
                  (vector act
                          (vector-ref env 0)
                          (vector-ref env 1)
                          ...))

(define env '#())
(define act (vector append list ...))
```

Conversion de fermeture (23)

- Peut-on améliorer la représentation chaînée?
- Oui : faire le chaînage **à même les blocs d'activation**



- L'**appelant** construit l'environnement de la fonction appelée en allouant un bloc d'activation ("frame") qui contient les **paramètres** et l'**environnement de la fermeture**

Conversion de fermeture (24)

- L'appel `(F X Y Z)` se traduit par

```
(let ((clo F))
  ((closure-code clo)
   (vector (closure-env clo) X Y Z)))
```

- La lambda-expression `(lambda ...)` se traduit par

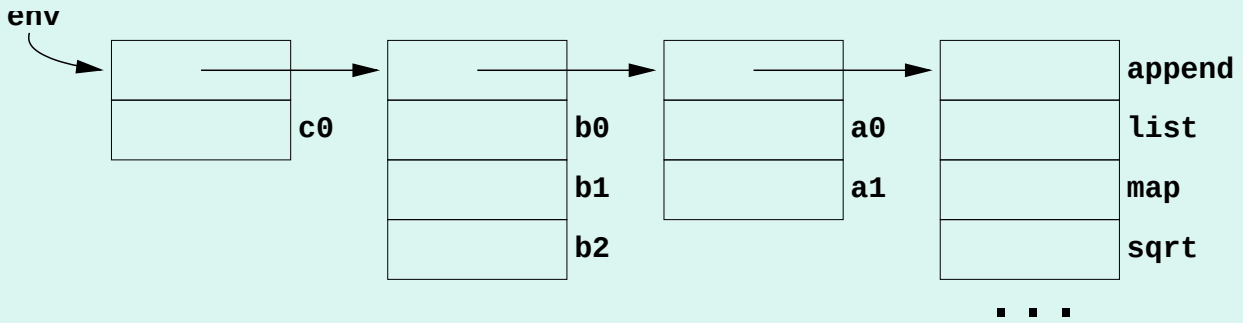
```
(make-closure (lambda (env) ...)
              env)
```

- Globalement on a

```
(define env (vector append list ...))
```

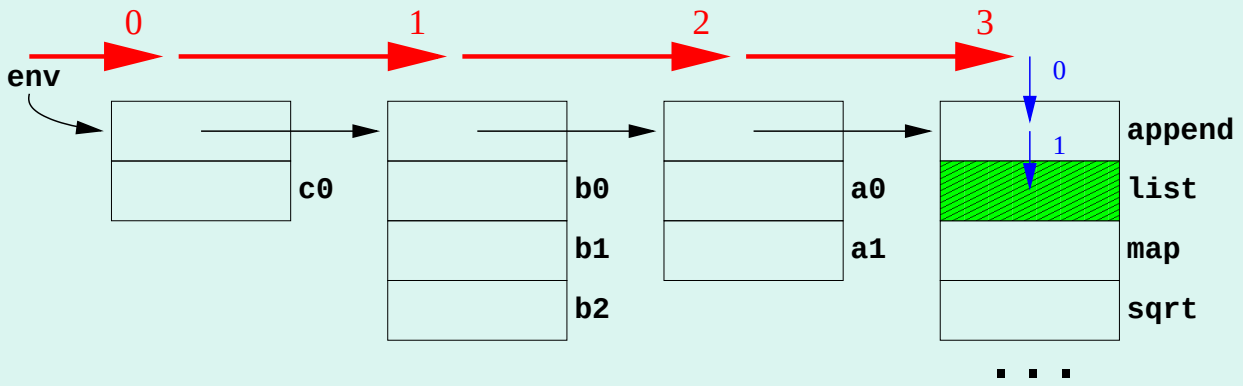
Conversion de fermeture (25)

- Dans le corps de la fonction appelée env réfère au début de la chaîne d'environnement, donc le bloc d'activation de la fonction
- Par exemple, dans le corps de la fonction "code" de `f` :



Conversion de fermeture (26)

- Chaque variable dans l'environnement a une coordonnée *up* et *over*
- *up* indique le nombre de liens de chaînage qu'il faut traverser pour se rendre au bloc d'activation qui contient la variable
- *over* c'est l'index de la variable dans le bloc d'activation
- Par exemple, pour `list` : *up*=**3**, *over*=**1**



Conversion de fermeture (27)

- Lors de la traduction il faut maintenir un **environnement de compilation** qui décrit la forme qu'aura l'environnement à l'exécution

- Cet environnement de compilation permet d'obtenir la coordonnée *up/over* pour toute variable lexicale

- Nous utiliserons des listes de listes d'identificateurs (et `#f` = lien) :

```
((#f c0)
 (#f b0 b1 b2)
 (#f a0 a1)
 (append list map ...))
```

Conversion de fermeture (28)

- La **conversion de fermeture** (“closure conversion”) c’est la procédure de traduction d’une expression E en une expression E' équivalente où
  - E contient (possiblement) des lambda-expressions avec des variables libres
  - E' contient seulement des lambda-expressions sans variables libres

- Donc, dans E' les lambda-expressions peuvent s’implanter simplement comme un **pointeur vers du code machine** (ou un **index dans une table de fonctions**)

Conversion de fermeture (29)


```
(define gcte
  '((append list map ...)))

(define (cte-lookup cte var)
  (let loop ((cte cte) (up 0))
    (if (null? cte)
        (error "unbound" var)
        (let ((x (memq var (car cte))))
          (if x
              (cons up
                    (- (length (car cte))
                       (length x)))
              (loop (cdr cte)
                    (+ up 1)))))))

(define (closure-conv expr)
  (closurec expr gcte))
```

Conversion de fermeture (30)


```
(define (closurec expr cte)

  (define (cc expr)
    (closurec expr cte))

  (define (get-frame up)
    (if (= up 0)
        'env
        '(vector-ref ,(get-frame (- up 1)) 0)))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (let* ((p (cte-lookup cte var))
            (up (car p))
            (over (cdr p)))
       '(vector-ref ,(get-frame up)
                    ,over)))
```

Conversion de fermeture (31)


```
((set! ,var ,E1)
 (let* ((p (cte-lookup cte var))
        (up (car p))
        (over (cdr p)))
   `(vector-set! ,(get-frame up)
                 ,over
                 ,(cc E1))))

((lambda ,params ,E)
 (let ((new-cte
        (cons (cons #f params)
              cte)))
   `(make-closure
     (lambda (env)
       ,(closurec E new-cte))
     env)))
```

Conversion de fermeture (32)


```
((if ,E1 ,E2)
 `(if ,(cc E1) ,(cc E2)))
((if ,E1 ,E2 ,E3)
 `(if ,(cc E1) ,(cc E2) ,(cc E3)))

((,E0 . ,Es)
 (if (primitive? E0)
     `(,E0
       ,@(map cc Es))
     `(let ((clo ,(cc E0)))
        ((closure-code clo)
         (vector (closure-env clo)
                 ,@(map cc Es))))))

(,_
 (error "unknown expression" expr))))
```

Conversion de fermeture (33)

- Exemple de traduction :

```
> (closure-conv
    '(lambda (x) (lambda (y) (list x y))))

(make-closure
 (lambda (env)
   (make-closure
    (lambda (env)
      (let ((clo
             (vector-ref
               (vector-ref (vector-ref env 0)
                           0)
               1)))
        ((closure-code clo)
         (vector (closure-env clo)
                 (vector-ref (vector-ref env 0)
                             1)
                 (vector-ref env 1)))))
    env))
 env)
```

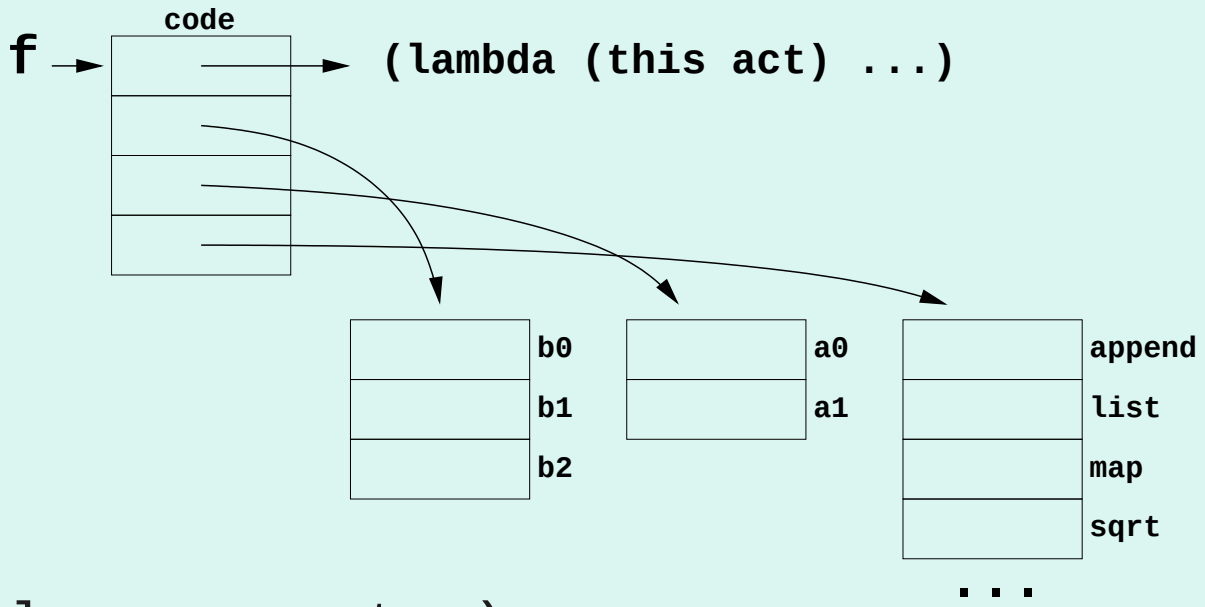
Conversion de fermeture (34)

- Avec variables renommées pour rendre plus lisible :

```
> (closure-conv
    '(lambda (x) (lambda (y) (list x y))))

(make-closure
 (lambda (env1)
   (make-closure
    (lambda (env2)
      (let ((clo
             (vector-ref
               (vector-ref (vector-ref env2 0)
                           0)
               1)))
        ((closure-code clo)
         (vector (closure-env clo)
                 (vector-ref (vector-ref env2 0)
                             1)
                 (vector-ref env2 1)))))
    env1))
 global-env)
```

Conversion de fermeture (35)

- Peut-on améliorer la représentation avec “display”?
- Oui : **intégrer le display à la fermeture**




```
(define make-closure vector)

(define (closure-code clo) (vector-ref clo 0))

(define (closure-ref clo i) (vector-ref clo (+ i 1)))
```

Conversion de fermeture (36)

- Traduction des lambda-expressions et appel de fermeture :

```
(lambda ...) -> (make-closure
                  (lambda (this act) ...)
                  act
                  (closure-ref this 0)
                  (closure-ref this 1)
                  ...)

(F X Y Z)    -> (let ((clo F))
                  ((closure-code clo)
                   clo
                   (vector X Y Z)))
```

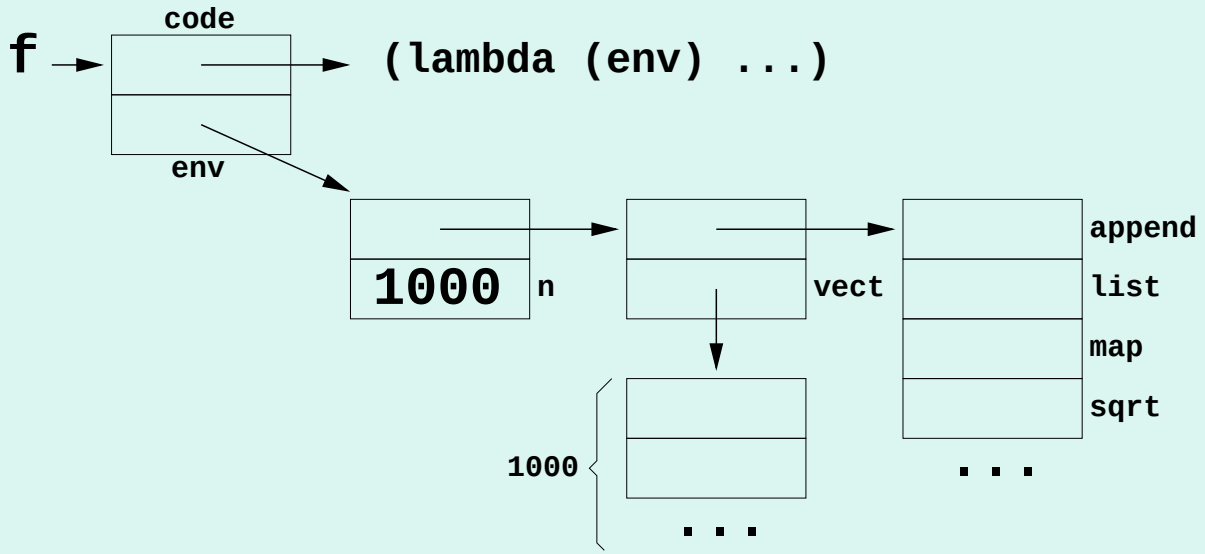
Conversion de fermeture (37)

- Toutes ces représentations d'environnement ont un **problème de rétention mémoire**
- Il se peut qu'une fermeture retienne des variables dans son environnement qui sont **mortes** (i.e. qui ne sont plus utiles pour le reste du calcul)
- Exemple :

```
(define (test vect)
  (let ((n (vector-length vect)))
    (lambda (x)
      (list x n))))

(define f (test (make-vector 1000)))

(f 17) => (17 1000)
(f 50) => (50 1000)
```

Conversion de fermeture (38)

- La fermeture créée sera comme suit

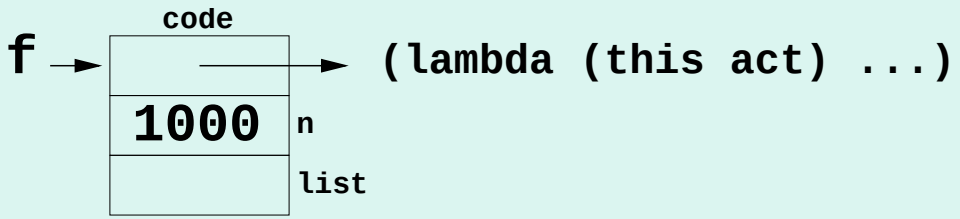


- Mais la variable `vect` est inutile pour l'exécution du corps de la fonction `f`
  - Le GC ne peut pas se débarasser du vecteur tant que la fermeture est accessible
  - L'environnement est pollué par des variables inutiles (perte d'espace, accès lent)

Conversion de fermeture (39)

- La **représentation plate** des fermetures offre une solution à ce problème

- Idée : stocker directement dans la fermeture seulement les variables que la fonction utilise, i.e. les **variables libres** de la fonction



- `this` permet d'accéder aux variables libres de la fonction et `act` permet d'accéder aux paramètres de la fonction

Conversion de fermeture (40)

- Cette représentation peut prendre moins ou plus d'espace que les autres représentations (ça dépend du contexte)
- De plus, telle quelle, cette représentation ne permet pas l'affectation aux variables libres
- La cause c'est qu'une variable peut être dupliquée dans plus d'une fermeture
- La représentation plate prend une **copie de chaque variable libre**

Conversion de fermeture (41)

- Exemple :

```
(define test
  (lambda (n)
    (cons (lambda (x)
            (begin
              (set! n (+ n x))
              n))
          (lambda (x)
            (begin
              (set! n (- n x))
              n)))))

(define p (test 0))
(define inc (car p))
(define dec (cdr p))

(inc 3) => 3
(inc 3) => 6
(dec 1) => 5
```

Conversion de fermeture (42)

- Conversion de fermeture avec environnements chaînés :

```
(define test
  (make-closure
   (lambda (env1)
     (cons (make-closure
            (lambda (env2)
              (begin
                (vector-set!
                 (vector-ref env2 0)
                 1
                 (+ (vector-ref (vector-ref env2 0) 1)
                    (vector-ref env2 1)))
                (vector-ref (vector-ref env2 0) 1)))
            env1)
           (make-closure
            (lambda (env3)
              (begin
                (vector-set!
                 (vector-ref env3 0)
                 1
                 (- (vector-ref (vector-ref env3 0) 1)
                    (vector-ref env3 1)))
                (vector-ref (vector-ref env3 0) 1)))
            env1)))
   global-env))
```

Conversion de fermeture (43)

- Pour traiter les affectations correctement il faut
  - Identifier les variables qui sont **mutables** (qui sont la cible d'un `set!`)
  - Créer des cellules pour contenir leur valeur
  - Les fermetures partageront ces cellules

- Note : heureusement, en Scheme l'affectation est peu souvent employée

Conversion de fermeture (44)

- Étapes de traduction :
  - **Alpha-conversion** qui donne un nom unique à toutes les variables (pour simplifier le reste de la traduction)
  - **Analyse de mutation** (“mutation analysis”)
  - **Conversion des affectations** (“assignment conversion”)
  - **Conversion des fermetures** (“closure conversion”)

Conversion de fermeture (45)


```
(define (alpha-conv expr)
  (alphac expr '()))

(define (alphac expr env)

  (define (rename v)
    (cond ((assq v env) => cdr)
          (else v)))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (rename var))

    ((set! ,var ,E1)
     `(set! ,(rename var)
            ,(alphac E1 env)))
```

Conversion de fermeture (46)


```
((lambda ,params ,E)
 (let* ((fresh-params
         (map (lambda (p) (cons p (gensym)))
              params))
        (new-env
         (append fresh-params env)))
   `(lambda ,(map cdr fresh-params)
      ,(alphac E new-env))))

((if ,E1 ,E2)
 `(if ,(alphac E1 env)
      ,(alphac E2 env)))
((if ,E1 ,E2 ,E3)
 `(if ,(alphac E1 env)
      ,(alphac E2 env)
      ,(alphac E3 env)))

((,E0 . ,Es)
 `(,(if (primitive? E0) E0 (alphac E0 env))
   ,@(map (lambda (e) (alphac e env))
          Es)))

(,_
 (error "unknown expression" expr))))
```

Conversion de fermeture (47)


```
(define (mv expr)
  (match expr
    (,const when (constant? const)
     '())

    (,var when (variable? var)
     '())

    ((set! ,var ,E1)
     (union (list var) (mv E1)))

    ((lambda ,params ,E)
     (mv E))

    ((if ,E1 ,E2)
     (union (mv E1) (mv E2)))
    ((if ,E1 ,E2 ,E3)
     (union (mv E1) (union (mv E2) (mv E3))))

    ((,E0 . ,Es)
     (union (if (primitive? E0) '() (mv E0))
            (apply union (map mv Es))))

    (,_
     (error "unknown expression" expr))))
```

Conversion de fermeture (48)


```
(define (assign-conv expr)
  (assignc expr
           (union (mv expr) (fv expr))))

(define (assignc expr mut-vars)

  (define (mutable? v) (memq v mut-vars))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (if (mutable? var)
         `(car ,var)
         var))

    ((set! ,var ,E1)
     `(set-car! ,var
                ,(assignc E1 mut-vars)))
```

Conversion de fermeture (49)


```
((lambda ,params ,E)
 (let* ((mut-params
         (map (lambda (p) (cons p (gensym)))
              (keep mutable? params)))
        (params2
         (map (lambda (p)
                (if (mutable? p)
                    (cdr (assq p mut-params))
                    p))
              params)))
   `(lambda ,params2
      ,(if (null? mut-params)
           (assignc E mut-vars)
           `((lambda ,(map car mut-params)
               ,(assignc E mut-vars))
             ,@(map (lambda (x) `(list ,(cdr x)))
                    mut-params))))))
```

Conversion de fermeture (50)


```
((if ,E1 ,E2)
 `(if ,(assignc E1 mut-vars)
      ,(assignc E2 mut-vars)))
((if ,E1 ,E2 ,E3)
 `(if ,(assignc E1 mut-vars)
      ,(assignc E2 mut-vars)
      ,(assignc E3 mut-vars)))

((,E0 . ,Es)
 `(,(if (primitive? E0) E0 (assignc E0 mut-vars))
   ,@(map (lambda (e) (assignc e mut-vars))
          Es)))

(,_
 (error "unknown expression" expr))))
```

Conversion de fermeture (51)


```
(define (closure-conv expr)
  (closurec expr '()))

(define (closurec expr cte)

  (define (pos id)
    (let ((x (memq id cte)))
      (and x
           (- (length cte)
              (length x)))))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (let ((p (pos var)))
       (if p
           `(closure-ref $this ,p)
           var)))
```

Conversion de fermeture (51)


```
((lambda ,params ,E)
 (let ((new-cte (fv expr)))
   `(make-closure
     (lambda ($this ,@params)
       ,(closurec E new-cte))
     ,@(map (lambda (v) (closurec v cte))
            new-cte))))

((if ,E1 ,E2)
 `(if ,(closurec E1 cte) ,(closurec E2 cte)))
((if ,E1 ,E2 ,E3)
 `(if ,(closurec E1 cte) ,(closurec E2 cte) ,(closurec

((,E0 . ,Es)
 (if (primitive? E0)
     `(,E0
       ,@(map (lambda (e) (closurec e cte))
              Es))
     `(let (($clo ,(closurec E0 cte)))
        ((closure-code $clo)
         $clo
         ,@(map (lambda (e) (closurec e cte))
                Es)))))

(,_
```

Conversion de fermeture (52)

- Traduction de l'exemple `inc/dec` :

```
(define test
  (make-closure
   (lambda ($this g99)
     (let (($clo (make-closure
                  (lambda ($this n)
                    (cons (make-closure
                           (lambda ($this x)
                             (begin
                               (set-car!
                                (closure-ref $this 0)
                                (+ (car (closure-ref $this 0)) x))
                               (car (closure-ref $this 0))))
                           n)
                          (make-closure
                           (lambda ($this x)
                             (begin
                               (set-car!
                                (closure-ref $this 0)
                                (- (car (closure-ref $this 0)) x))
                               (car (closure-ref $this 0))))
                           n))))))
       ((closure-code $clo) $clo (list g99)))))

(define p (let (($clo test)) ((closure-code $clo) $clo 0)))
(define inc (car p))
(define dec (cdr p))

(let (($clo inc)) ((closure-code $clo) $clo 3)) => 3
(let (($clo inc)) ((closure-code $clo) $clo 3)) => 6
(let (($clo dec)) ((closure-code $clo) $clo 1)) => 5
```

Conversion de fermeture (53)

- Application à l'interprète rapide (dans le but de sérialiser les fermetures)

```
(define (ev expr cte)

  (cond ((constant? expr)
         (let ((val (constant-val expr)))
           (lambda (rte) val)))

        ((variable? expr)
         (let ((pos (cte-lookup cte expr)))
           (lambda (rte) (rte-lookup rte pos))))

        ((lambda? expr)
         (let ((b (ev (lambda-body expr)
                      (cte-extend cte
                                  (lambda-params expr)))))
           (lambda (rte)
             (lambda args
               (b (rte-extend rte args))))))

        ...))
```

Conversion de fermeture (54)


```
(define (ev expr cte)

  (cond ((constant? expr)
         (let ((val (constant-val expr)))
           (make-closure
            (lambda ($this rte)
              (closure-ref $this 0)) ;; val
            val)))

        ((variable? expr)
         (let ((pos (cte-lookup cte expr)))
           (make-closure
            (lambda ($this rte)
              (rte-lookup rte
                          (closure-ref $this 0))) ;; pos
            pos)))
```

Conversion de fermeture (55)


```
((lambda? expr)
 (let ((b (ev (lambda-body expr)
              (cte-extend cte
                          (lambda-params expr)))))
   (make-closure
    (lambda ($this rte)
      (make-closure
       (lambda ($this . args)
         (let (($clo (closure-ref $this 0))) ;; b
           ((closure-code $clo)
            $clo
            (rte-extend
             (closure-ref $this 1) ;; rte
             args))))
       (closure-ref $this 0)
       rte))
    b)))

...))
```

Lambda-lifting (1)

- Le **lambda-lifting** est une transformation qui permet d'**éviter de créer des fermetures** dans certains cas
- C'est utile pour réduire la quantité de mémoire allouée par un programme, et donc de réduire l'utilisation du garbage-collector
- S'applique aux lambda-expressions qui sont liées à des variables par des `let`, `let*`, `letrec`, ou définitions internes

Lambda-lifting (2)

- Exemple simple de programme transformable par lambda-lifting :

```
(define f
  (lambda (a b)
    (let ((g (lambda (x) (+ x a))))
      (g (g b)))))

(f 1 100)
(f 2 500)
```

- Normalement il faudrait créer une fermeture pour la lambda-expression `(lambda (x) (+ x a))` pour que la fonction se souvienne de la valeur de a dans chaque appel à `f`

Lambda-lifting (3)

- Mais si on passe explicitement la valeur de a en paramètre, on élimine le besoin de créer une fermeture :

```
(define f
  (lambda (a b)
    (let ((g (lambda (a x) (+ x a))))
      (g a (g a b)))))

(f 1 100)
(f 2 500)
```

Lambda-lifting (4)

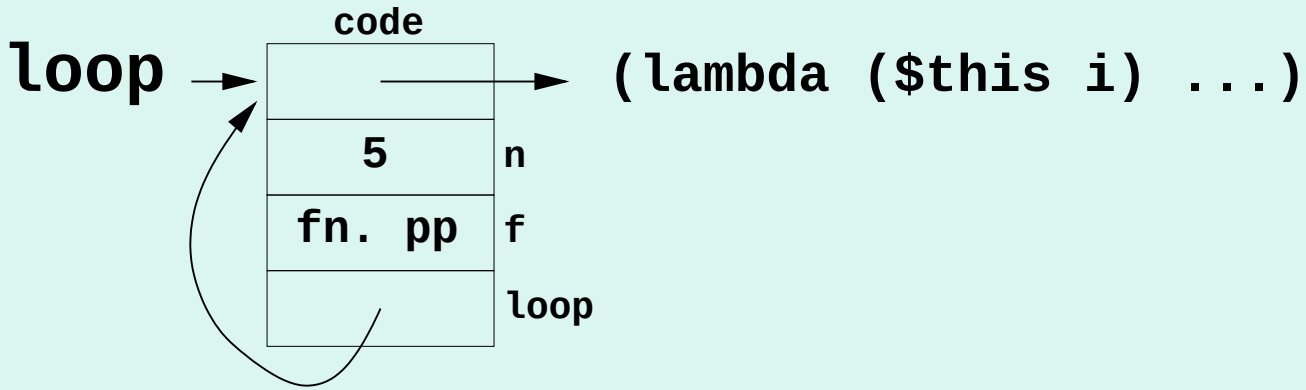
- Exemple de fonction récursive transformable par lambda-lifting :

```
(define repeter
  (lambda (n f)
    (letrec ((loop
              (lambda (i)
                (if (<= i n)
                    (begin
                      (f i)
                      (loop (+ i 1)))))))
      (loop 1))))

(repeter 5 pp)
```

- On aurait pu utiliser un `let`-nommé ou une définition interne (ce serait tout à fait équivalent)

Lambda-lifting (5)

- La fermeture créée pour `loop` permet d'accéder aux variables `n`, `f` et `loop` (à noter que dans ce cas particulier `loop` = `$this`) :




```
(define repeter
  (lambda (n f)
    (letrec ((loop
              (lambda (i)
                (if (<= i n)
                    (begin
                      (f i)
                      (loop (+ i 1)))))))
      (loop 1))))

(repeter 5 pp)
```

Lambda-lifting (6)

- Traduction par conversion de fermeture :

```
define repeter
(make-closure
 (lambda ($this n f)
  (let ((loop
         (make-closure
          (lambda ($this i)
           (if (<= i (closure-ref $this 0))
               (begin
                (let (($clo (closure-ref $this 1)))
                  ((closure-code $clo) $clo i))
                (let (($clo (closure-ref $this 2)))
                  ((closure-code $clo) $clo (+ i 1))))))
          n
          f
          #f)))
   (closure-set! loop 2 loop)
   (let (($clo loop))
     ((closure-code $clo) $clo 1))))))
```

Lambda-lifting (7)

- Peut-on éviter de créer une fermeture pour `loop`?
- Il s'agit de trouver une autre façon pour que le corps de `loop` soit informé de la valeur des variables `n`, `f` et `loop`
- Solution : **passer ces valeurs en paramètre** à `loop`!
- Cela fonctionne même pour l'affectation si on a fait une conversion d'affectation au préalable

Lambda-lifting (8)

- Nouveau code sans fermeture :

```
(define repeter
  (lambda (n f)
    (let ((loop
           (lambda (n f loop i)
             (if (<= i n)
                 (begin
                   (f i)
                   (loop n f loop (+ i 1)))))))
      (loop n f loop 1))))
```

- Remarques :
  - le `letrec` a été remplacé par un `let`
  - la fonction `loop` n'a plus de variables libres

Lambda-lifting (9)

- Idée de base de la transformation :
  - Pour toute variable v qui est liée à une lambda-expression l et qui apparait seulement dans des appels de fonction en position fonctionnelle, c'est-à-dire $(v \ \ldots)$:
    - Ajouter les variables libres de l au début de la liste de paramètres formels de l
    - Ajouter les variables libres de l au début de la liste de paramètres actuels de chaque appel de fonction $(v \ \ldots)$

Lambda-lifting (10)

- Cela ne fonctionne pas tout à fait lorsqu'il y a plus d'une fonction à transformer :

```
(define repeter
  (lambda (n z f)
    (letrec ((g
              (lambda (x) (* x z)))
             (loop
              (lambda (i)
                (if (<= i n)
                    (begin
                      (f (g i))
                      (loop (+ i 1)))))))
      (loop 1))))

(repeter 5 10 pp) => 10 20 30 40 50
```

Lambda-lifting (11)

- La transformation donne :

```
(define repeter
  (lambda (n z f)
    (let ((g
           (lambda (z x) (* x z)))
          (loop
           (lambda (n f loop g i)
             (if (<= i n)
                 (begin
                   (f (g z i))
                   (loop n f loop g (+ i 1)))))))
      (loop n f loop g 1))))
```

- `z` est maintenant une variable libre de `loop`

Lambda-lifting (12)

- Il faut répéter la transformation jusqu'à ce qu'il n'y ait plus de changement :

```
(define repeter
  (lambda (n z f)
    (let ((g
           (lambda (z x) (* x z)))
          (loop
           (lambda (z n f loop g i)
             (if (<= i n)
                 (begin
                   (f (g z i))
                   (loop z n f loop g (+ i 1)))))))
      (loop z n f loop g 1))))
```

Lambda-lifting (13)

- Le lambda-lifting ne s'applique pas lorsque les lambda-expressions ne sont pas directement liées à des variables, ou bien lorsque ces variables ne sont pas uniquement utilisées en position fonctionnelle

- Dans ces cas, il se peut que :
  - les sites d'appel ne soient pas dans la portée (lexicale) des variables libres
  - les sites d'appel de la fermeture soient aussi des sites d'appel de d'autres fermetures (donc il y a une possibilité de conflit d'ajout de paramètres actuels)

Lambda-lifting (14)

- Exemple de programme qui n'est pas transformable par lambda-lifting :

```
(define f
  (lambda (a b)
    (let ((g (lambda (x) (+ x a))))
      (list g
            (g (g b))))))

(define x (f 1 100))

(define h (car x))

(h 2) ;; comment transformer cet appel?
```

Lambda-lifting (15)

- Autre programme qui n'est pas transformable par lambda-lifting :

```
(define map
  (lambda (f lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              (map f (cdr lst))))))

(define g
  (lambda (a b c lst)
    (list (map (lambda (x) (* x (+ a b))) lst)
          (map (lambda (y) (+ y c)) lst))))
```

- Puisque 2 fermetures peuvent se faire appeler par l'appel `(f (car lst))` et que ces fermetures n'ont pas le même nombre de variables libres, comment transformer cet appel?

Gestion mémoire (1)

- La **gestion mémoire** c'est la tâche de
  - réserver (**allouer**) l'espace mémoire pour des données,
  - libérer (**récupérer**) l'espace mémoire des données qui ne sont plus requises,
  - gérer les lectures et écritures de données, ce qui demande d'établir une **représentation** des données et des méthodes pour **accéder** aux données

- Dans plusieurs langages de programmation, tels que C et C++, la gestion mémoire est **manuelle**

- C'est au programmeur de faire appel à des fonctions de librairie pour allouer et récupérer l'espace mémoire (`malloc`/`free`)

Gestion mémoire (2)

- Il est reconnu que la gestion mémoire manuelle est difficile pour les programmes complexes, et une **source importante de bogues** difficiles à résoudre (lieu où se manifeste le problème $\neq$ lieu du bogue de gestion mémoire)
- 2 types de bogues de gestion mémoire :
  - **Fuite de mémoire** : un espace mémoire devenu inutile n'est pas récupéré, ou est récupéré tardivement
  - **Pointeur fou** : le programme utilise un pointeur vers un espace mémoire qui ne contient plus la bonne donnée (car il a été récupéré et possiblement alloué pour une autre donnée), ou qui est à l'extérieur de l'espace mémoire du programme

Gestion mémoire (3)

- Avec la **gestion mémoire automatique** c'est le langage de programmation qui s'occupe de la gestion mémoire, ce qui permet d'éviter complètement le problème de pointeur fou, et plusieurs problèmes de fuites de mémoire (mais pas tous)

- Déf. : une **donnée vivante** c'est une donnée qui sera utilisée plus tard dans l'exécution du programme

- Déf. : une **donnée morte** c'est une donnée qui n'est pas vivante ("garbage")

Gestion mémoire (4)

- En principe, pour tout point t de l'exécution du programme, les données mortes au point t peuvent être récupérées sans que cela affecte le bon déroulement du programme (une donnée morte au point t ne peut pas devenir vivante plus tard)
- Corollaire : aussitôt qu'une donnée devient morte elle peut-être récupérée
- Questions :
  - Comment identifier les données mortes?
  - Quand et comment les récupérer?

Gestion mémoire (5)

- La vivacité d'une donnée est en général **indécidable**

```
(let ((x (cons 11 22))) ; vivante ou morte?
  (f 33)
  (write x))
```

- On doit se contenter d'une **approximation conservatrice**
- Il est acceptable de classifier comme "vivante" une donnée qui est en réalité "morte", mais pas l'inverse
- C'est-à-dire

	classification par le GC vivante	morte
en réalité vivante	OK	**PAS OK**
en réalité morte	OK	OK

Gestion mémoire (5)

- On décrète que toute donnée est classifiée vivante si elle est contenue dans une **racine** :
  - Variable globale, variable statique d'une classe, ou registre du processeur
  - Continuation, ou bloc d'activation sur la pile d'exécution
- De plus, toute donnée contenue dans une donnée classifiée vivante est classifiée vivante
- Les autres données sont classifiées mortes

Gestion mémoire (6)

- En d'autres termes, une donnée est classifiée vivante ssi elle est **accessible directement ou indirectement à partir des racines**

- Un **récupérateur mémoire automatique** (ou "ramasse miettes", ou "glaneur de cellules", ou "garbage collector") c'est une procédure ou processus qui récupère la mémoire occupée par les données mortes

- Lisp fut le premier langage à employer un GC (1960), puis SIMULA-67, Prolog, Java, ...

Gestion mémoire (7)

- Quelques termes utiles :
  - **mutateur** (“mutator”) : le module principal qui fait les requêtes d’allocation et les accès aux objets
  - **récupérateur** (“collector”) : le module qui détermine quels objets sont morts et qui les récupèrent
  - On parle souvent de ces modules comme s’ils étaient des **processus** séparés (parfois ils sont des processus concurrents, parfois des coroutines, ou bien le récupérateur est une procédure appelée par le mutateur)

Quelques types de GC

- **bloquant** : est activé lorsqu'une requête d'allocation échoue et libère tout l'espace occupé par les données mortes
- **incrémentiel** : distribue son travail en petites parcelles pendant l'exécution du programme (permet d'éviter de longues pauses du programme)
- **copiant** : déplace les données vivantes
- **compactant** : déplace les données vivantes pour qu'elles occupent une zone contiguë (permet d'éviter les problèmes de fragmentation de la mémoire)
- **générationel** : découpe la mémoire en zones contenant chacun des objets de même age approx.
- **conservateur** : connait peu sur le format de stockage des objets en mémoire

Représentation des données (1)

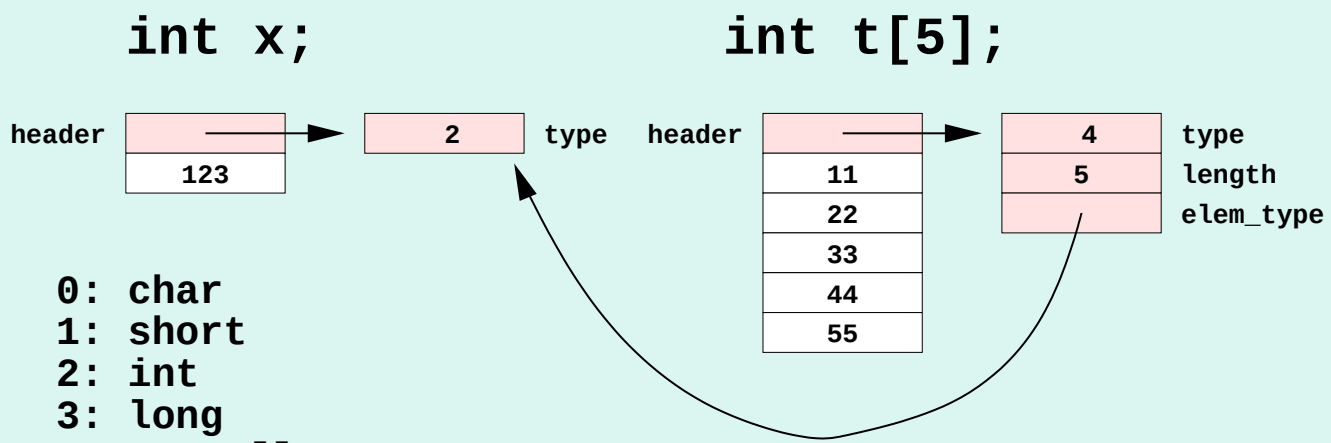
- Dans les langages statiques sans GC (e.g. FORTRAN, Pascal, C), la représentation des données correspond de près à celle qu'offre le CPU. Par exemple, en C :
  - Les types `char`, `short`, `int`, `long` sont normalement représentés par un groupe de 1, 2, 4, et 8 octets (i.e. entiers de 8, 16, 32, et 64 bits en complément à 2)
  - Le type `struct` et les tableaux sont représentés par un groupe d'octets qui est la concaténation de la représentation de ses champs/éléments
- **Le type d'une donnée n'est pas représenté en mémoire à l'exécution** (seul le compilateur a besoin d'associer un type aux variables pour lui permettre de savoir quel code générer pour y accéder)

Représentation des données (2)

- Dans les langages statiques avec GC (e.g. Java, Haskell) et les langages à typage dynamique (e.g. Smalltalk, Python, Ruby, Perl, JavaScript, Lisp, Scheme), il est nécessaire de **connaître le type des données à l'exécution** pour :
  - Le **typage dynamique** : `(car x)`
  - Les **prédicats de type** : `(pair? x)`
  - La **réflexion** : `foo.hasOwnProperty("bar")`
  - Le **GC**, pour qu'il puisse correctement manipuler les données allouées par le mutateur (ce qui est une forme de réflexion)

Représentation des données (3)

- Pour **associer l'information de type aux données à l'exécution** il faut que la représentation des données soit **augmentée avec de l'information de type**
- Par exemple, on pourrait ajouter à la représentation de tous les objets un champ d'entête (**header**) qui pointe vers un **descripteur de type** contenant un encodage de l'information de type
- Exemple simple avec Java (info de type en rose) :



Compteur de référence (1)

- Associe un **compteur** avec chaque objet qui indique le **nombre de références vers cet objet**
- Lors de l'exécution de l'affectation `v = E;` le compteur de **l'objet référé par** `E` **est incrémenté** et le compteur de **l'objet référé par** `v` **est décrémenté**
- Lorsque le compteur tombe à 0 l'objet est récupéré après avoir décrémenté le compteur des objets référés par celui-ci
- Il faut traiter les blocs d'activation comme des objets (lorsqu'ils sont retirés de la pile c'est comme si leur compteur tombait à 0)

Compteur de référence (2)

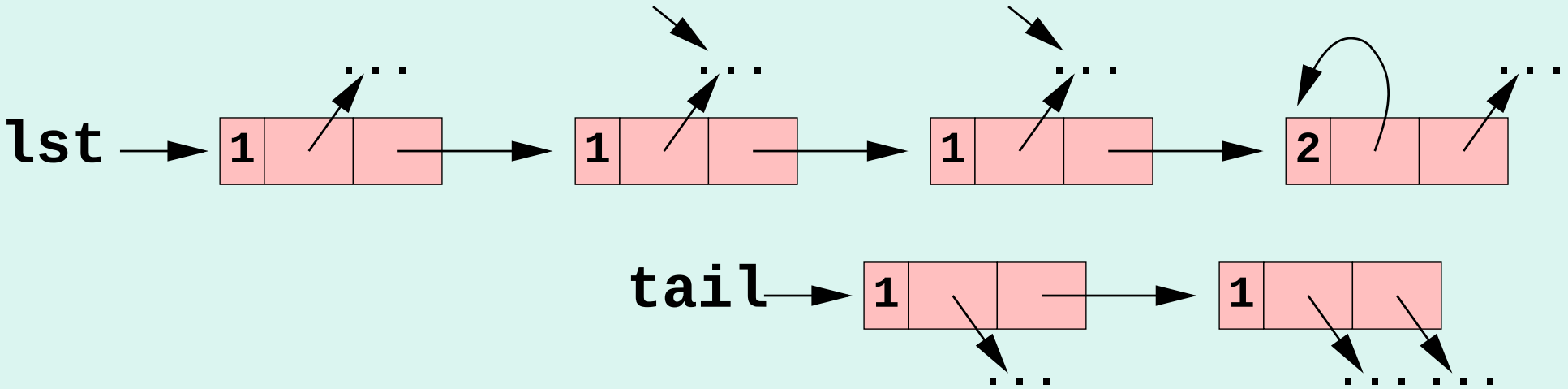

```
typedef struct object *ref;

struct object {
  int rc;
  ref car, cdr;
};

void inc(ref x) { x->rc++; }

ref cons(ref x, ref y)
{
  ref p = malloc(sizeof(struct object));
  p->rc = 1;
  inc(x); p->car = x;
  inc(y); p->cdr = y;
  return p;
}
```

Compteur de référence (3)


```
void dec(ref x)
{
  if (--x->rc == 0)
    {
      dec(x->car);
      dec(x->cdr);
      free(x);
    }
}

void set_car(ref p, ref x) { ... }

void set_cdr(ref p, ref x)
{
  inc(x); // ordre important!
  dec(p->cdr);
  p->cdr = x;
}
```

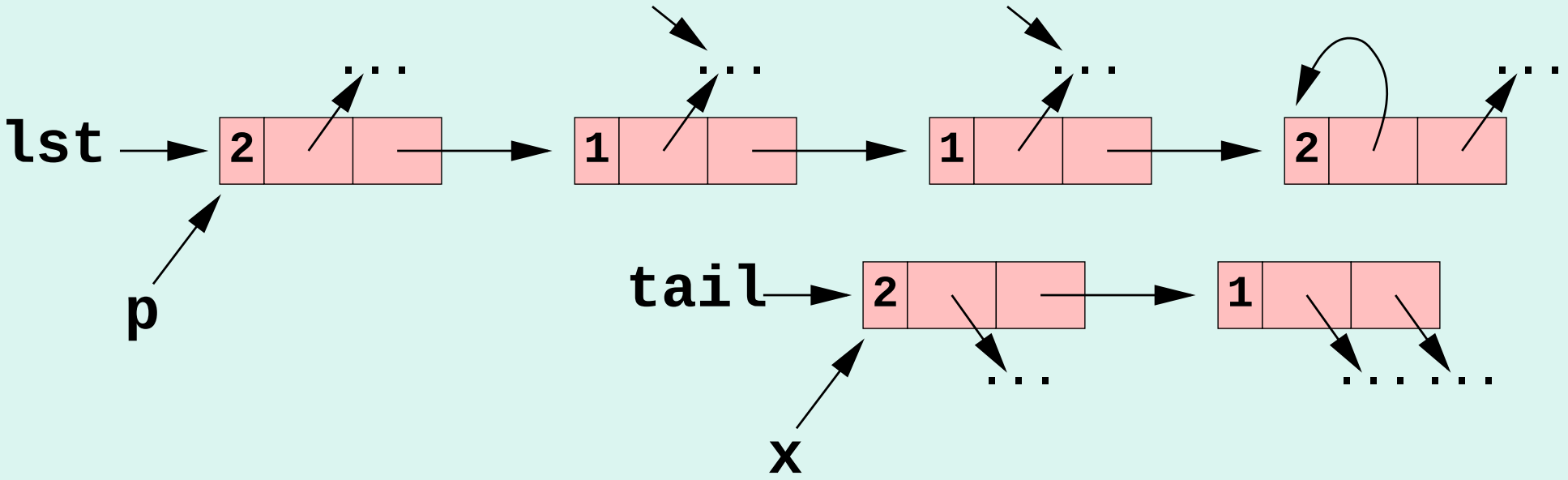
Compteur de référence (4)

- Trace de l'appel `set_cdr(lst,tail)`



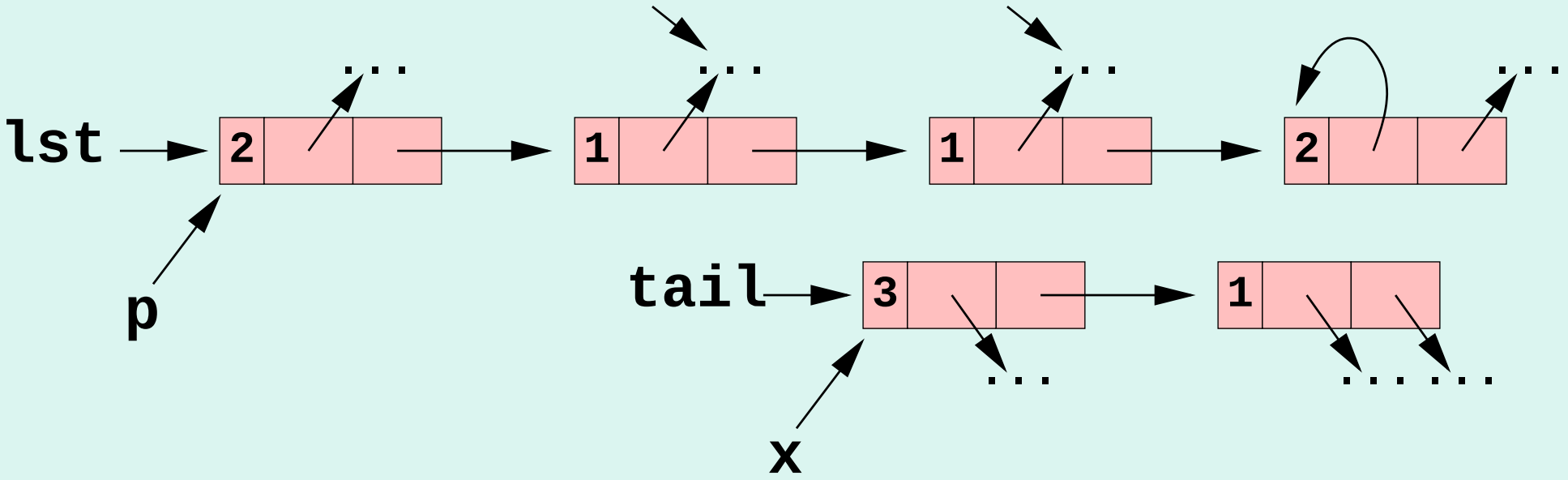
Compteur de référence (5)

- Trace de l'appel `set_cdr(lst,tail)`



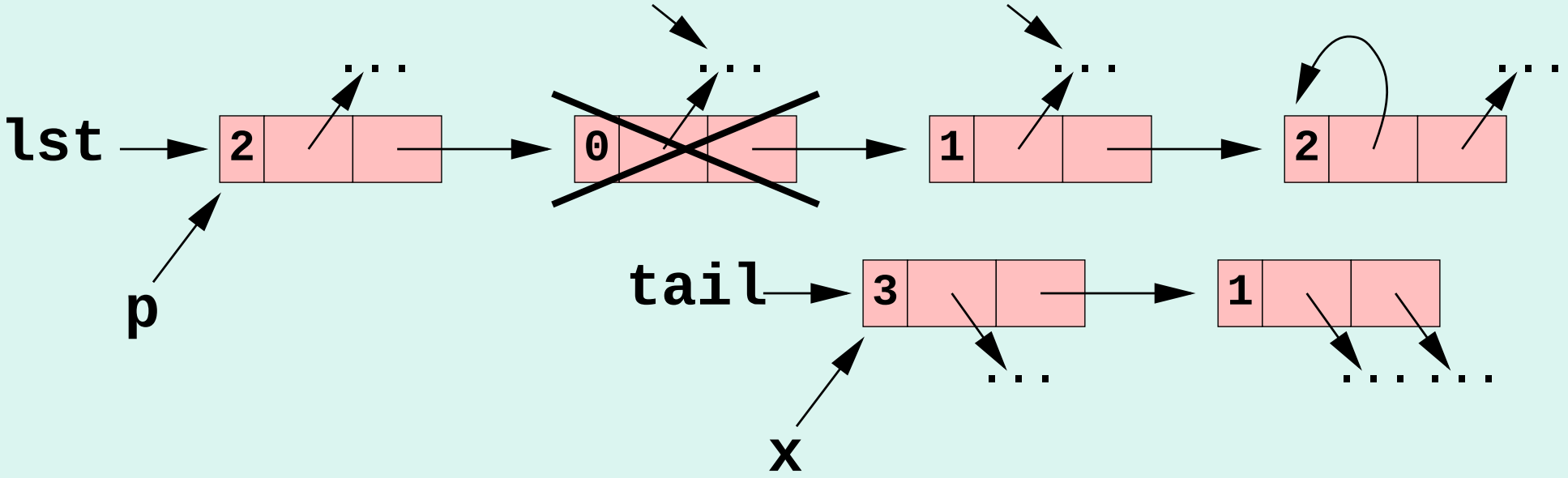
Compteur de référence (6)

- Trace de l'appel `set_cdr(lst,tail)`



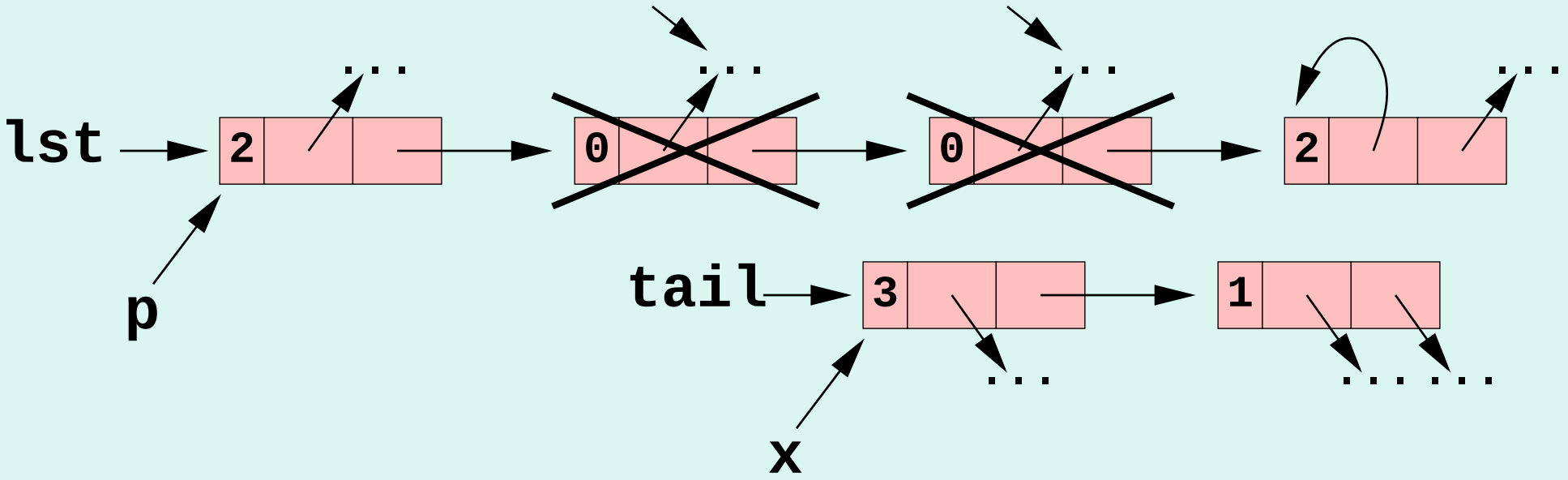
Compteur de référence (7)

- Trace de l'appel `set_cdr(lst,tail)`



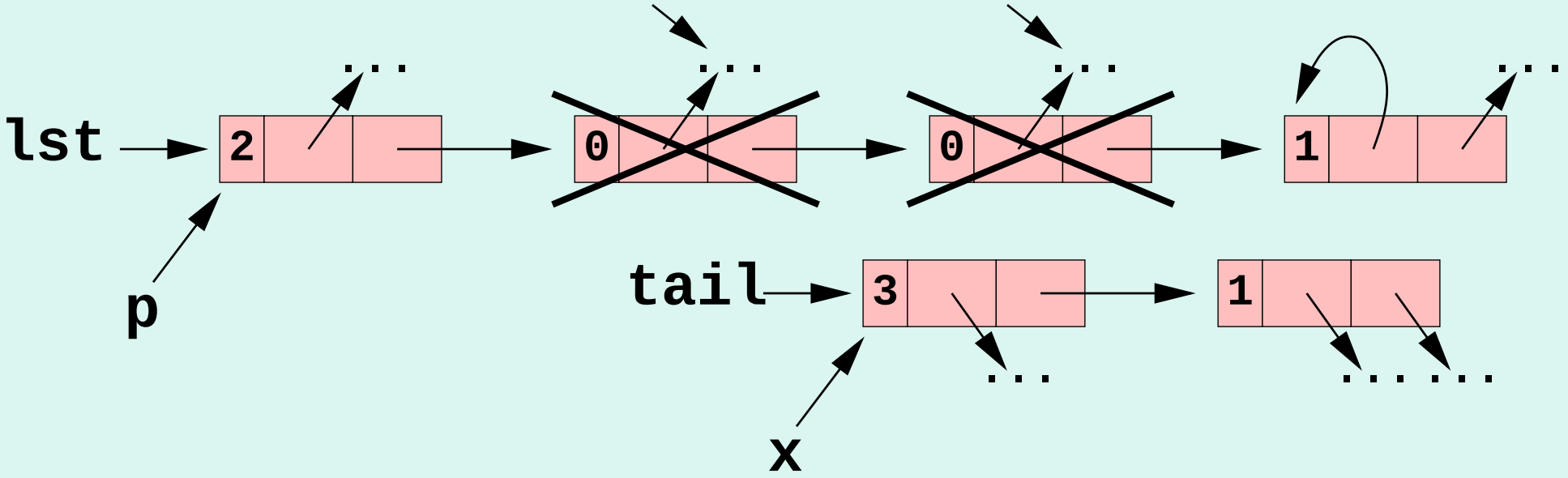
Compteur de référence (8)

- Trace de l'appel `set_cdr(lst,tail)`



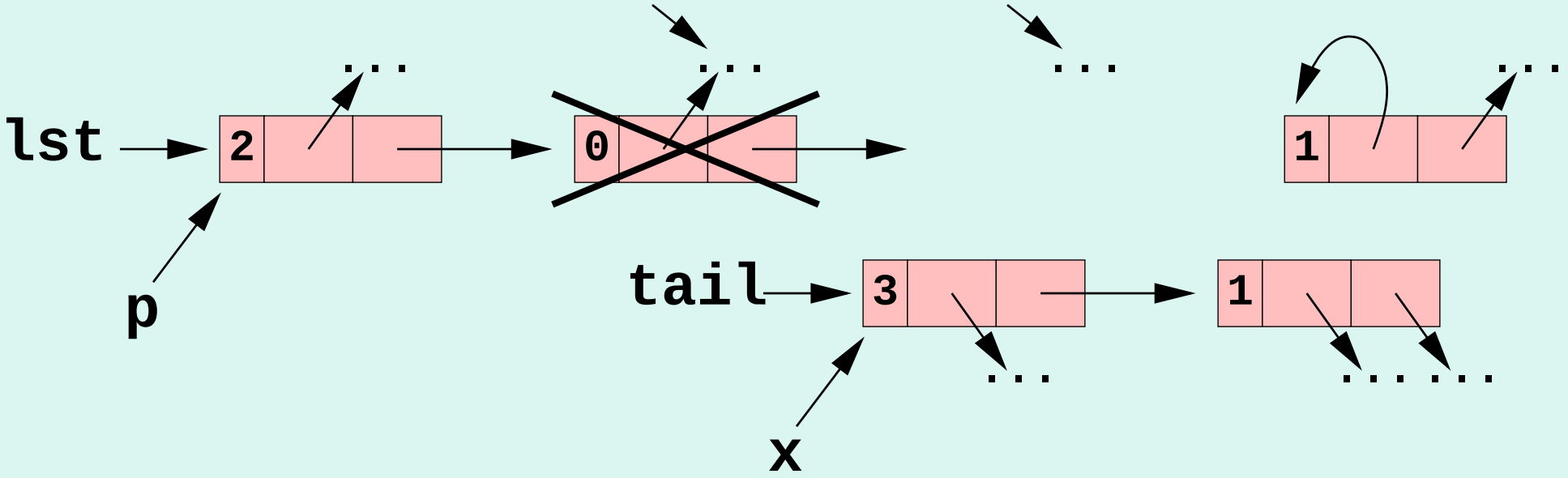
Compteur de référence (9)

- Trace de l'appel `set_cdr(lst,tail)`



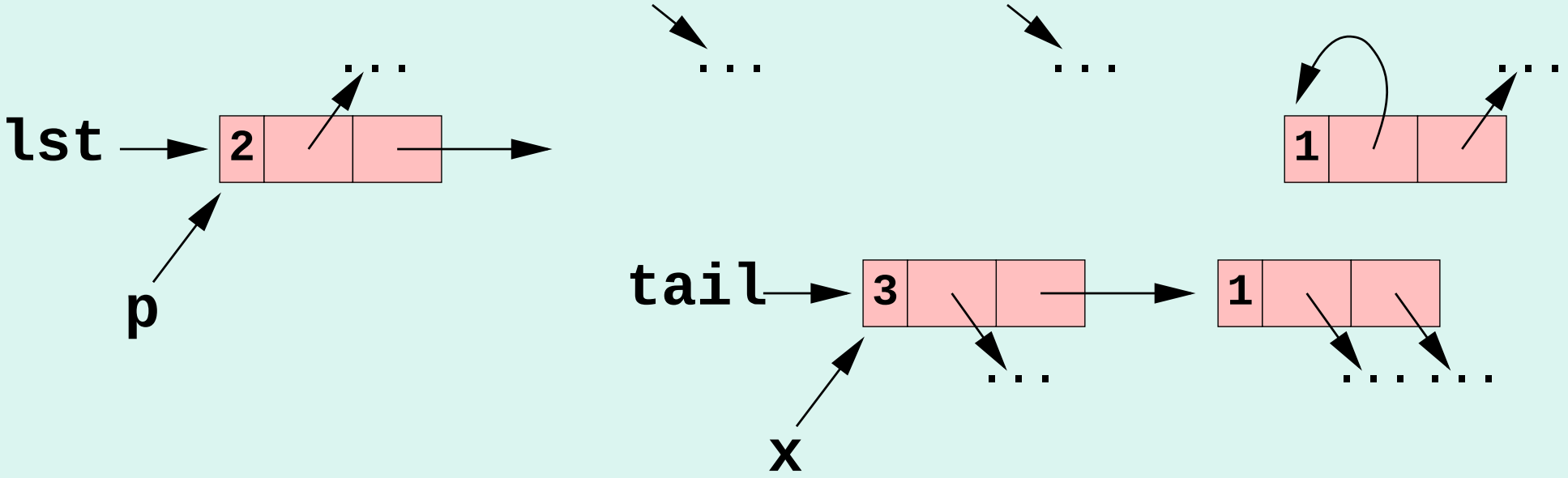
Compteur de référence (10)

- Trace de l'appel `set_cdr(lst,tail)`



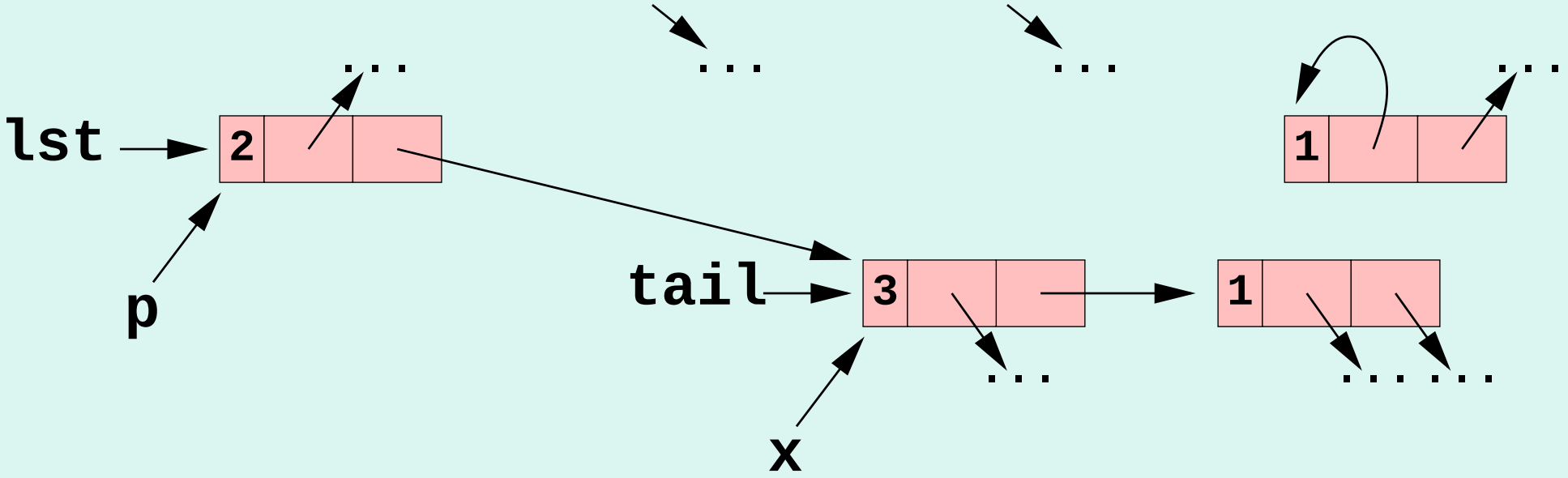
Compteur de référence (11)

- Trace de l'appel `set_cdr(lst,tail)`



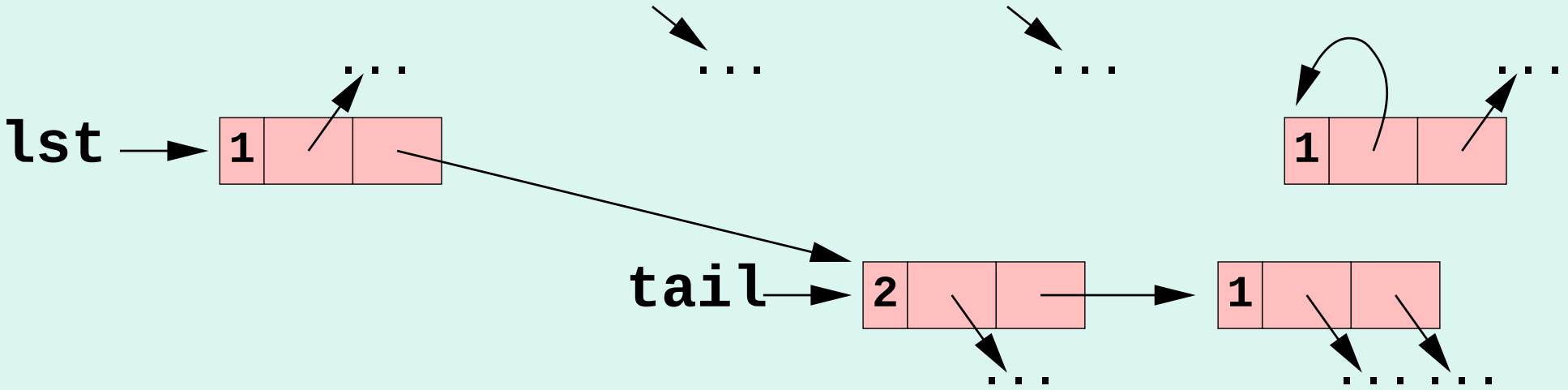
Compteur de référence (12)

- Trace de l'appel `set_cdr(lst,tail)`



Compteur de référence (13)

- Trace de l'appel `set_cdr(lst,tail)`



Compteur de référence (14)

- Cela donne un GC quasi-incrémentiel car le travail de récupération est entremêlé avec le mutateur
- Les structures contenant des cycles ne sont pas récupérées
- Technique simple, mais lente (car chaque affectation et appel/retour de fonction demande un travail supplémentaire)

Mark and sweep (1)

- Chaque objet contient un **bit de marquage**
- GC bloquant non-copiant à 2 phases :

1. traversée des objets vivants à partir des racines qui **marque** les objets visités
2. traversée de tous les objets alloués, les objets non-marqués sont récupérés (et les bits de marquage sont remis à zéro)

Mark and sweep (2)


```
typedef struct object *ref;

struct object {
  bool m;
  ref car, cdr;
};

#define HSIZE 1000

struct object heap[HSIZE];

ref freelist;

void init()
{ int i;
  freelist = NULL;
  for (i=0; i<HSIZE; i++)
    { heap[i].m = FALSE;
      heap[i].cdr = freelist;
      freelist = &heap[i];
    }
}
```

Mark and sweep (3)

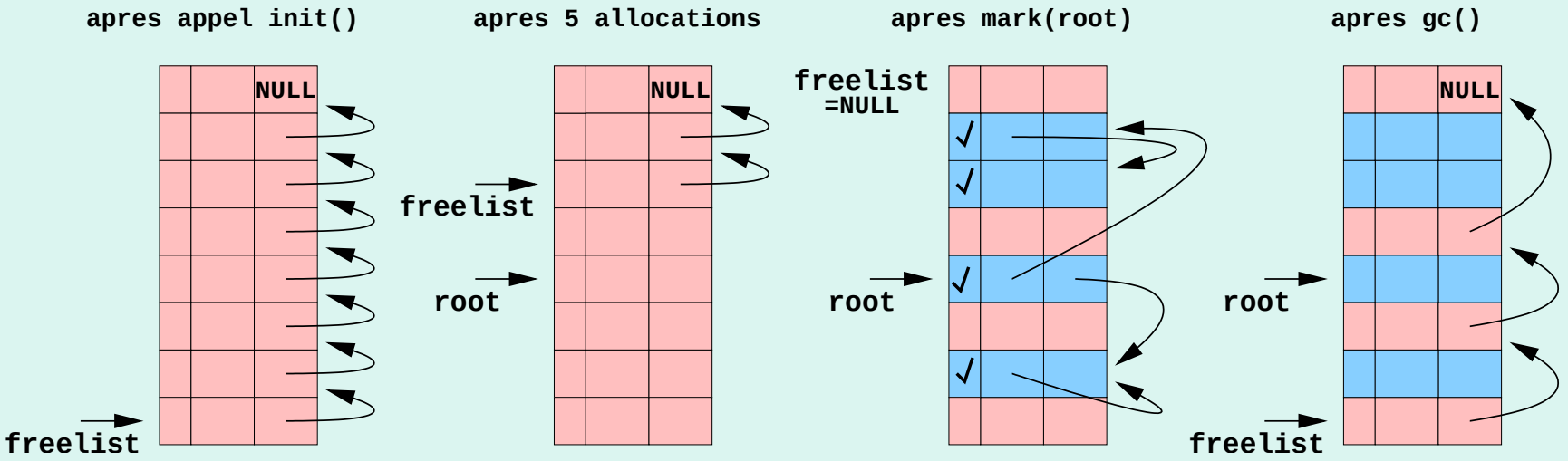

```
ref root; // racine(s)

void mark(ref x)
{ if (!x->m)
    { x->m = TRUE;
      mark(x->car);
      mark(x->cdr);
    }
}

void gc()
{ int i;
  mark(root);
  for (i=0; i<HSIZE; i++)
    if (heap[i].m)
      heap[i].m = FALSE;
    else
      { heap[i].cdr = freelist;
        freelist = &heap[i];
      }
  if (freelist == NULL) error("heap full");
}
```

Mark and sweep (4)


```
ref cons(ref x, ref y)
{
  ref p;

  if (freelist == NULL) gc();

  p = freelist;
  freelist = p->cdr;

  p->car = x;
  p->cdr = y;

  return p;
}
```

Mark and sweep (5)

- Trace :



Mark and sweep (6)

- L'algorithme de marquage est récursif
- On peut se servir d'une pile pour implanter les appels récursifs, mais cela demande dans le pire cas un espace mémoire de taille proportionnelle au tas
- Une autre approche c'est d'utiliser l'algorithme de marquage Deutsch-Schorr-Waite (1967)
- Cet algorithme représente la pile avec une liste de cellules chaînées **à l'intérieur des objets à marquer**

Mark and sweep (7)

- Le bit de marquage est remplacé par un petit champ entier (0..2)
- Ce champ indique l'état de marquage de l'objet

  0. objet pas encore marqué
  1. visite récursive du `car` débutée
  2. visite récursive du `cdr` débutée

- Dans l'état 1 le champ `car` est utilisé pour chaîner la pile
- Dans l'état 2 le champ `cdr` est utilisé pour chaîner la pile

Mark and sweep (8)

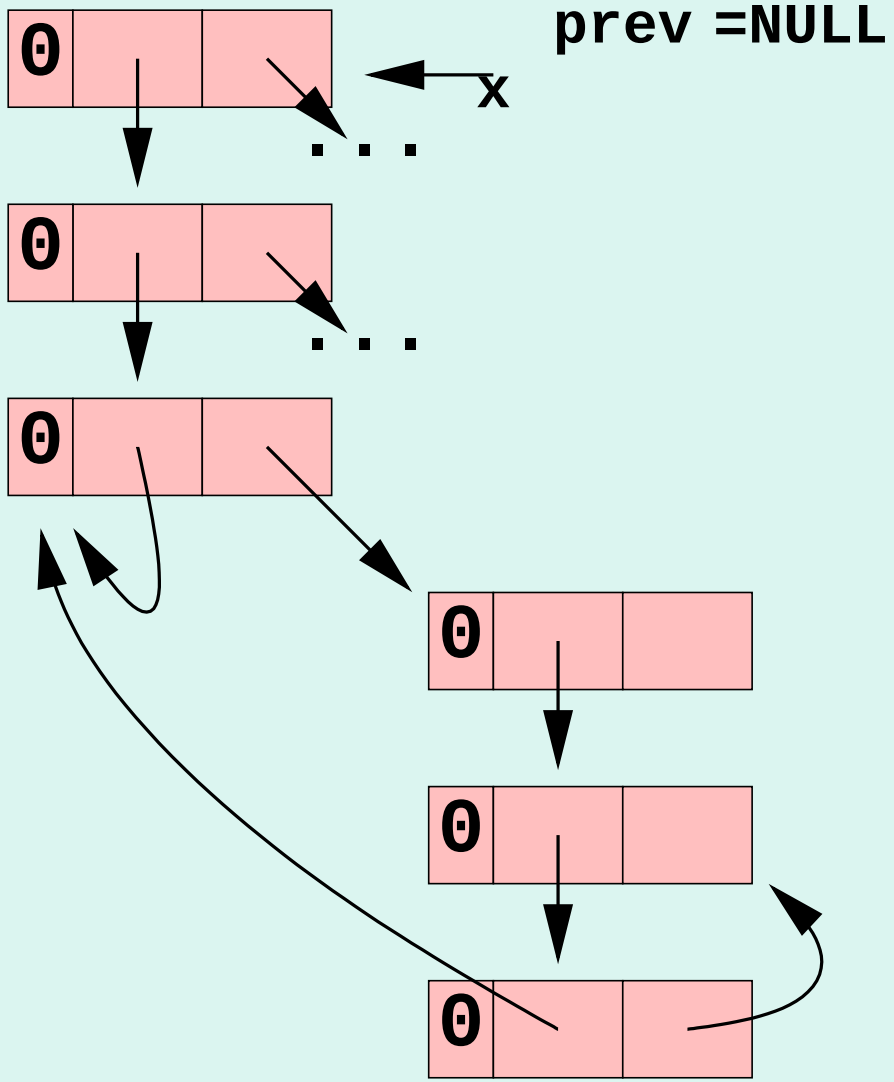

```
struct object {
  int state; // 0..2, initialisé à 0
  ref car, cdr;
};

void mark(ref x)
{
  ref temp;
  ref prev = NULL;

forward:
  if (x->state == 0)
    {
      x->state = 1;
      temp = x;
      x = x->car;
      temp->car = prev;
      prev = temp;
      goto forward;
    }
```

Mark and sweep (9)


```
backward:
  if (prev != NULL)
    switch (prev->state) {

      case 1:
        prev->state = 2;
        temp = prev->car;
        prev->car = x;
        x = prev->cdr;
        prev->cdr = temp;
        goto forward;

      case 2:
        temp = prev->cdr;
        prev->cdr = x;
        x = prev;
        prev = temp;
        goto backward;
      }
}
```

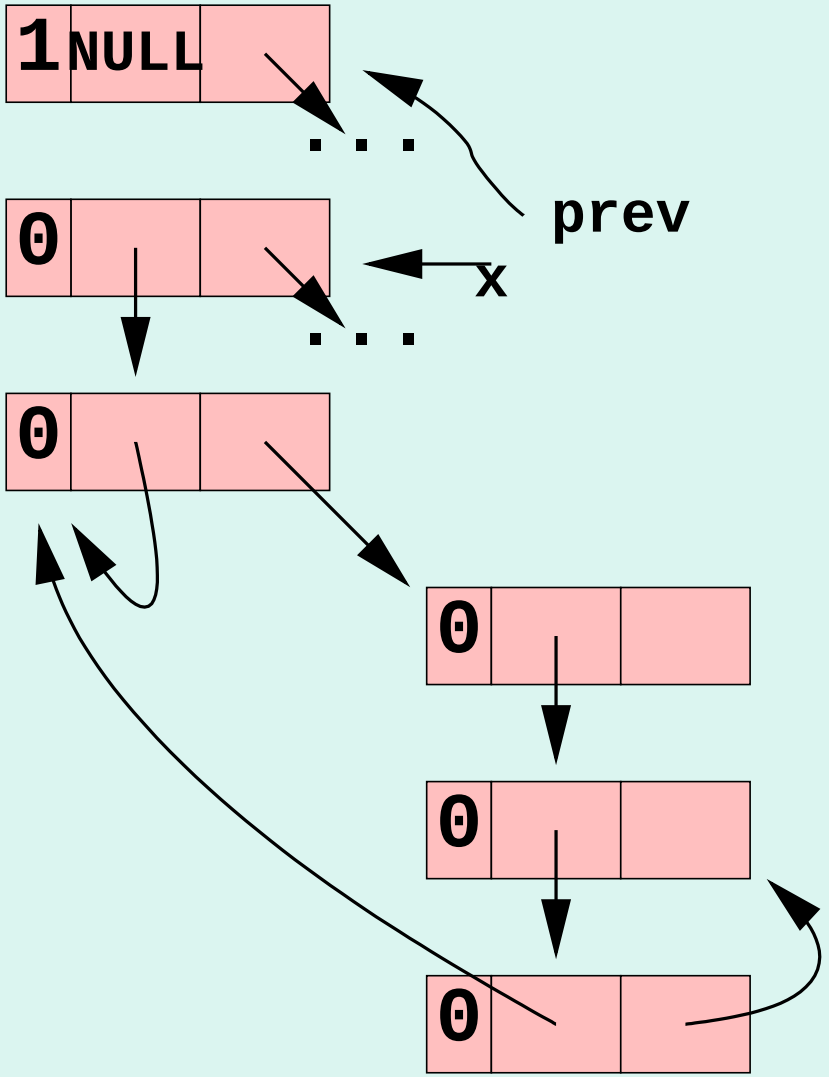
Mark and sweep (10)

- Trace :



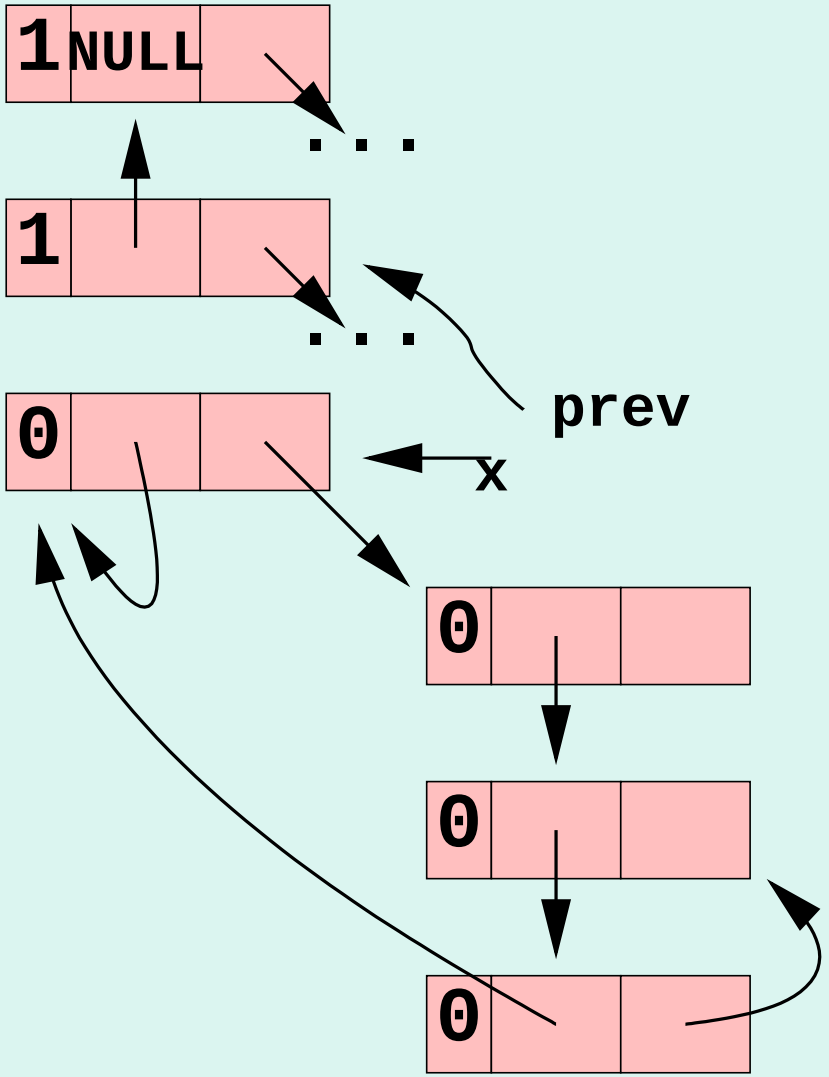
Mark and sweep (11)

- Trace :



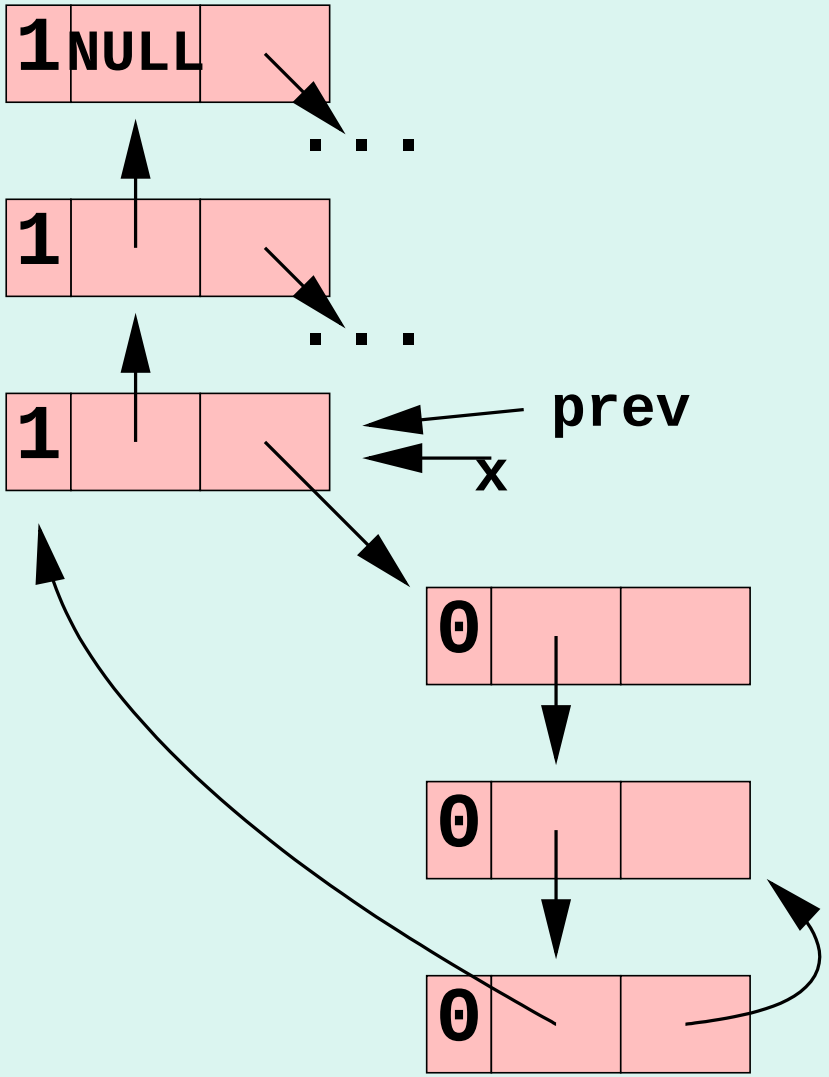
Mark and sweep (12)

- Trace :



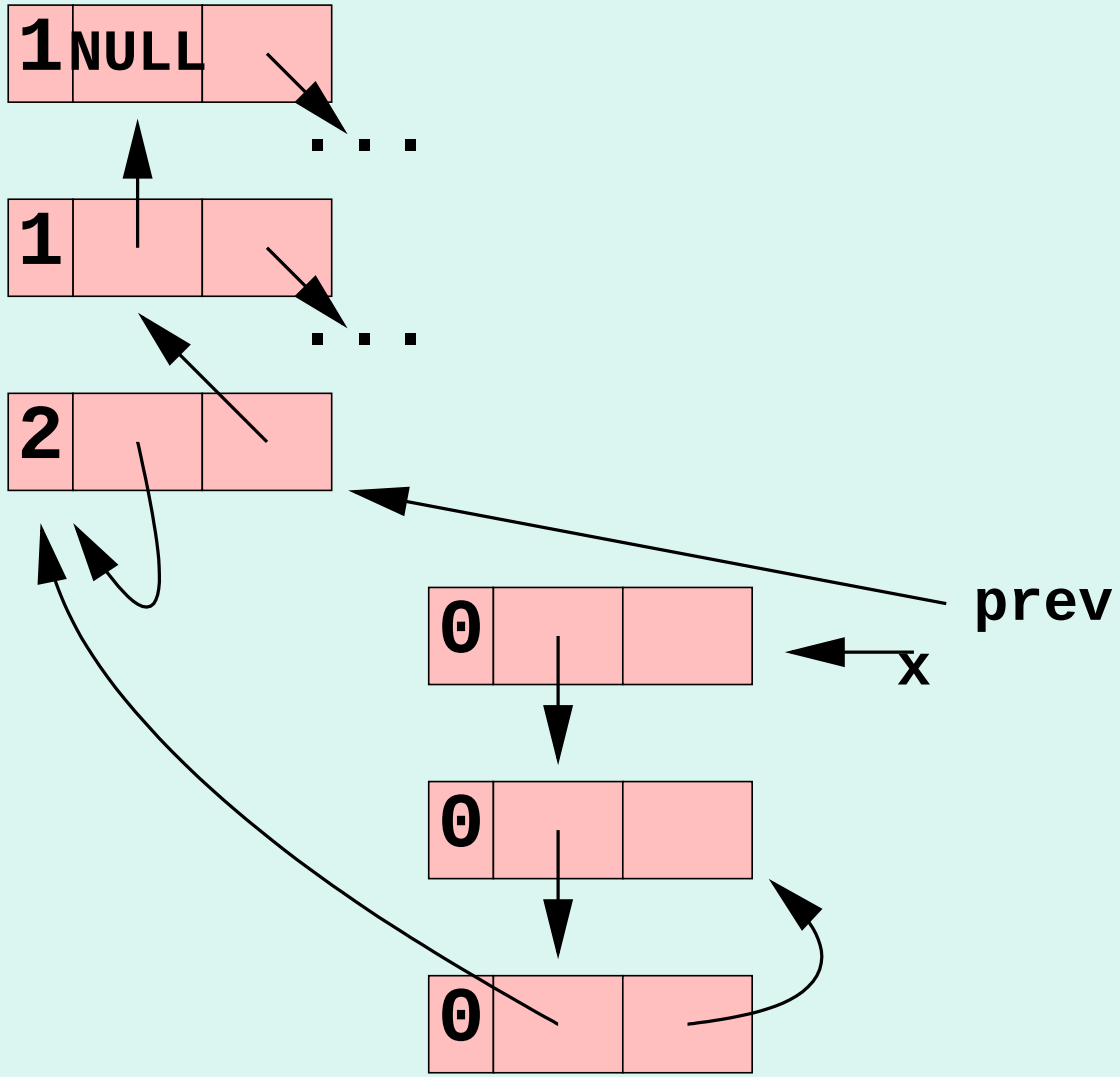
Mark and sweep (13)

- Trace :



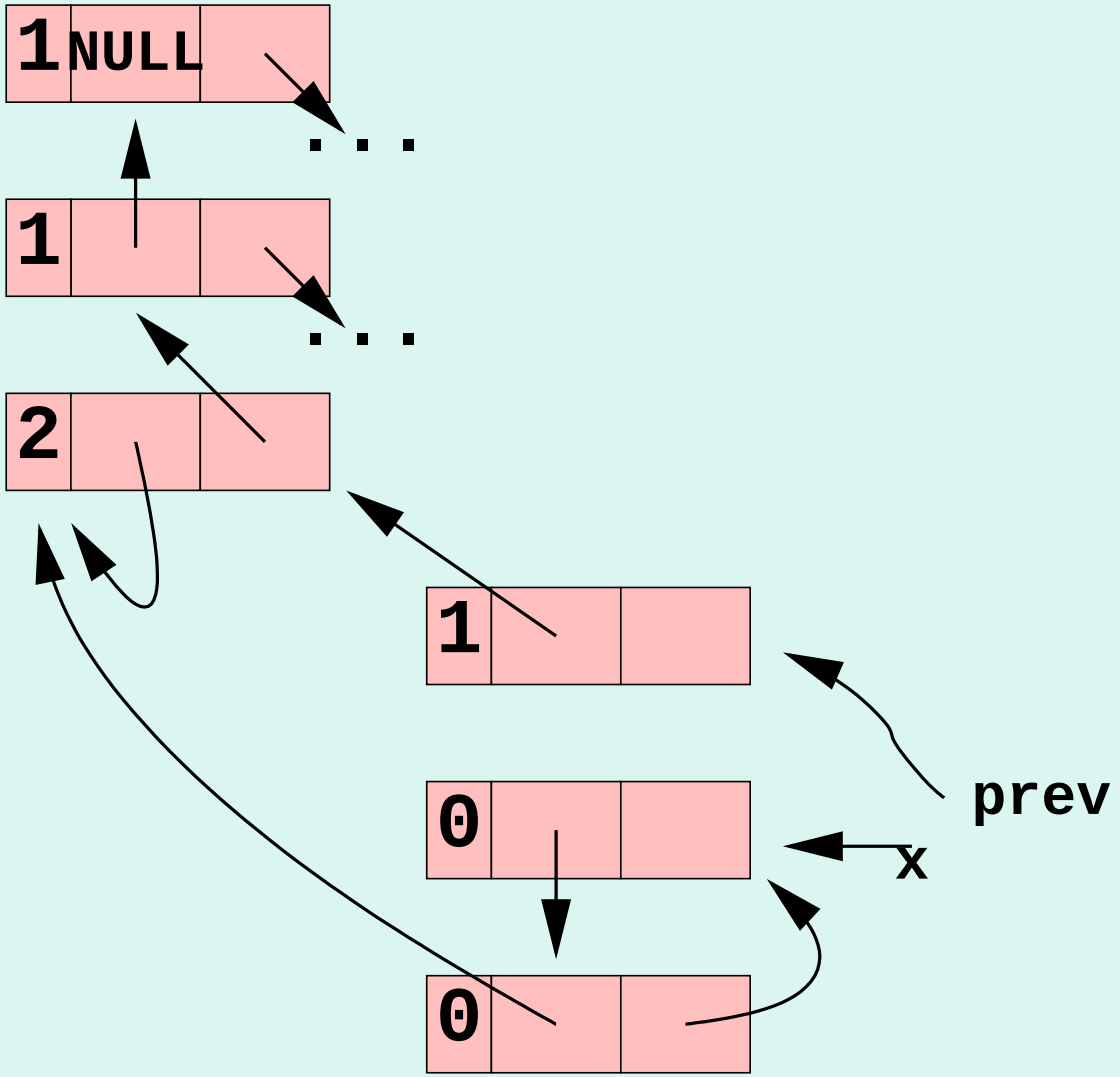
Mark and sweep (14)

- Trace :



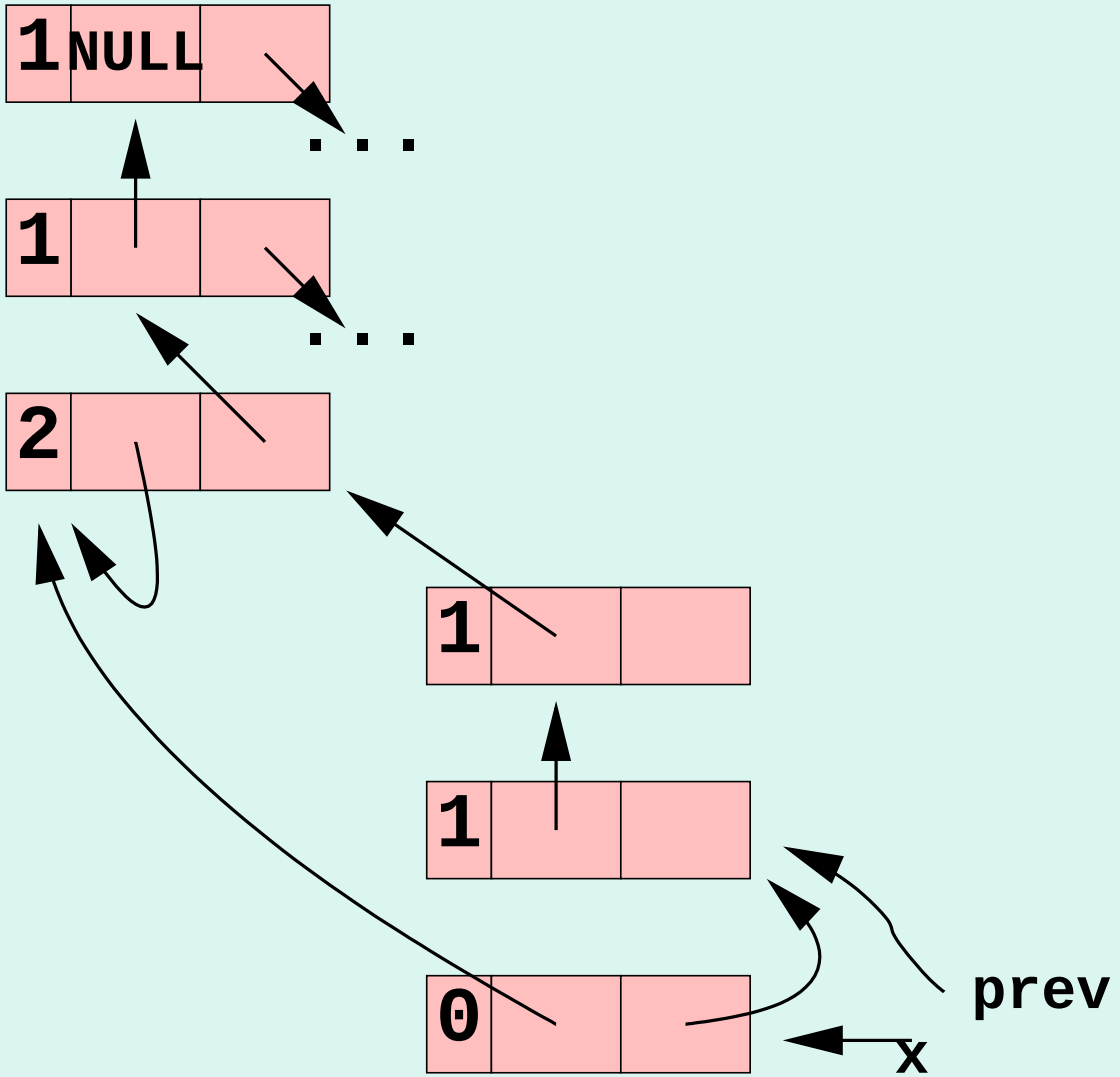
Mark and sweep (15)

- Trace :



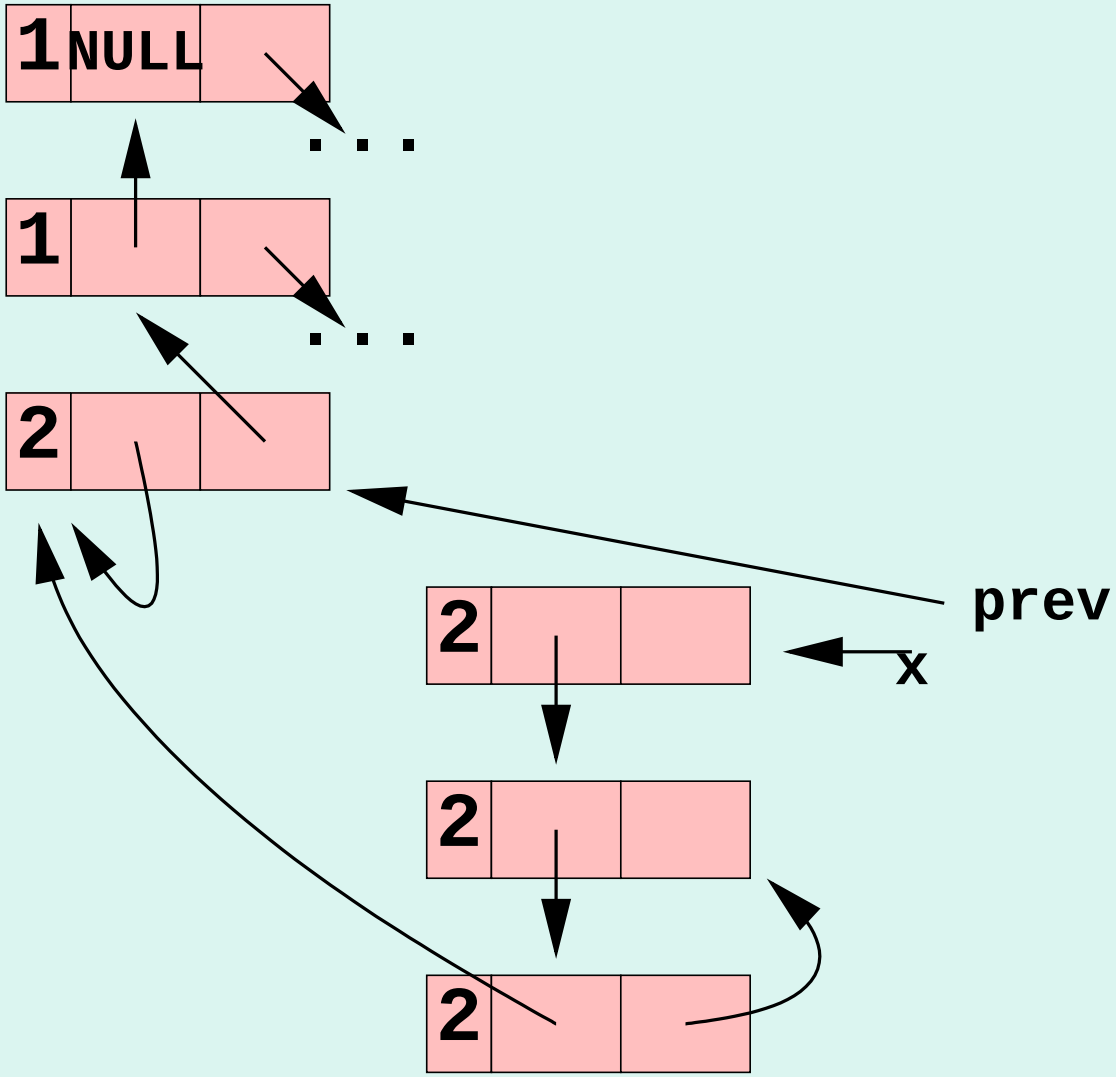
Mark and sweep (16)

- Trace :



Mark and sweep (17)

- Trace : (après quelques étapes)



Mark and sweep (18)

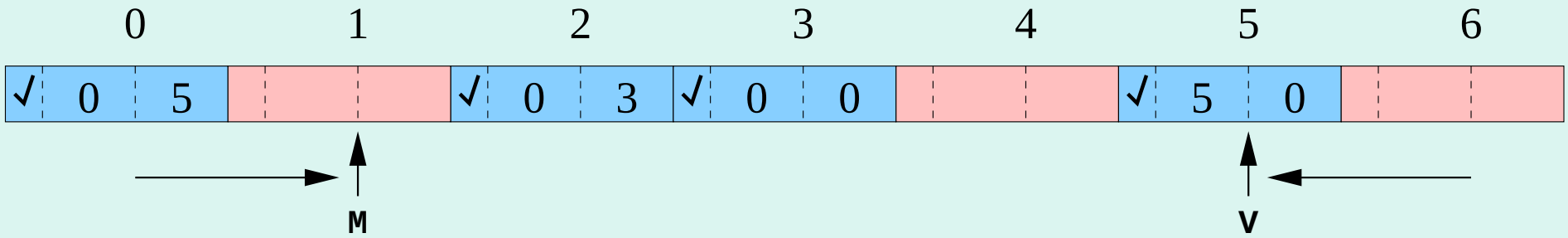
- L’algorithme Deutsch-Schorr-Waite se généralise à des objets plus grands
- Il faut avoir un code de plus pour le champ `state` que le nombre de références dans l’objet

Mark and compact (1)

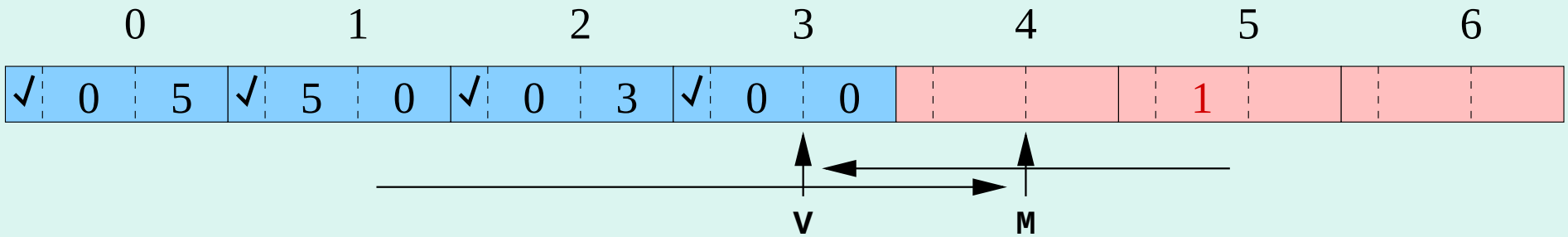
- Fait un marquage comme “mark and sweep” puis déplace les objets vivants pour qu’ils soient contiguës en mémoire
- La compaction se fait avec deux pointeurs partant aux extrèmes du tas (un cherche les objets morts, l’autre les objets vivants)
- Il faut une phase supplémentaire pour ajuster les références car certains objets ont été déplacés (il faut laisser un **“forwarding pointer”**)
- Simplifie l’algorithme d’allocation (on fait avancer un pointeur dans la zone libre)

Mark and compact (2)

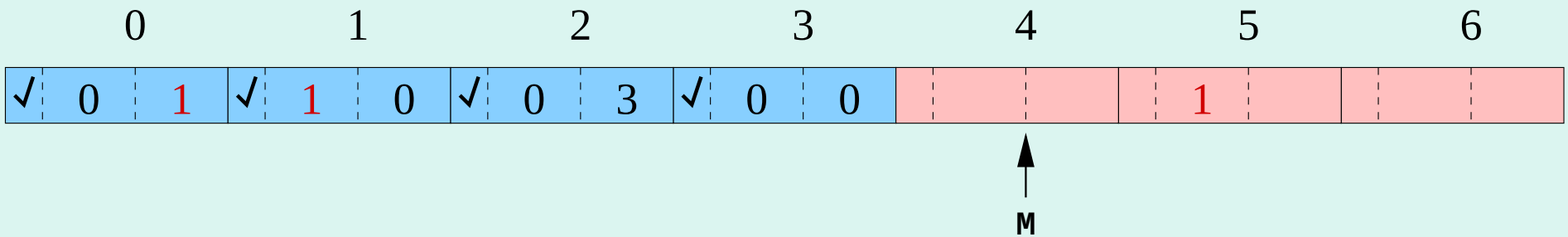
- Après marquage (racine = 2) et début de la recherche :



- Après première copie :



- Après mise à jour des références :

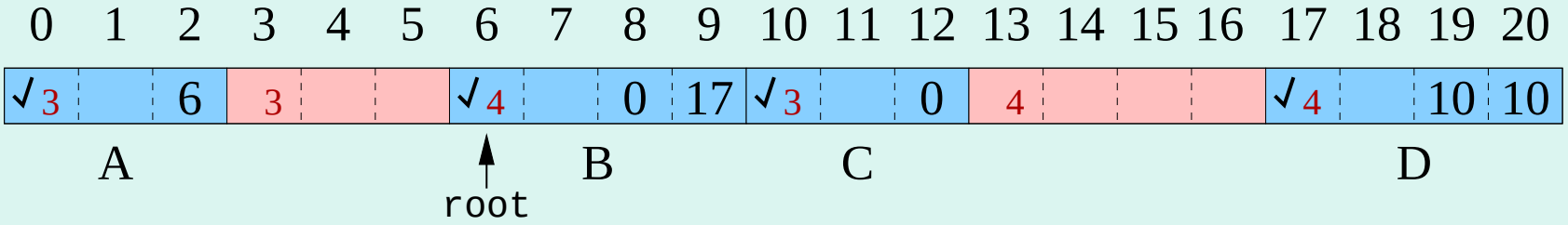


Mark and compact (3)

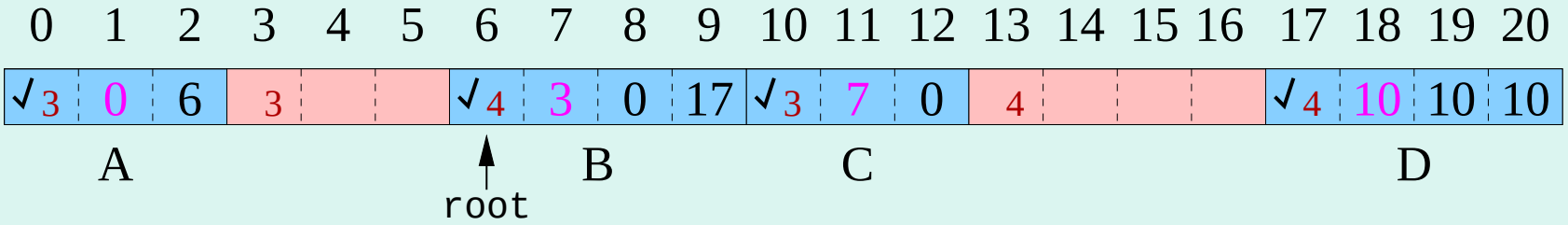
- Une variante de cet algorithme qui traverse le heap 3 fois, fonctionne avec des objets sont de taille variable
- Chaque objet a un champ dédié au “forwarding pointer”
- La phase de compaction traite les objets de gauche à droite et déplace le prochain objet vivant juste après le précédent (par “glissement”)
- Phases :
  1. Marquage
  2. Calcul des forwarding pointers
  3. Mise à jour des références
  4. Déplacement des objets

Mark and compact (4)

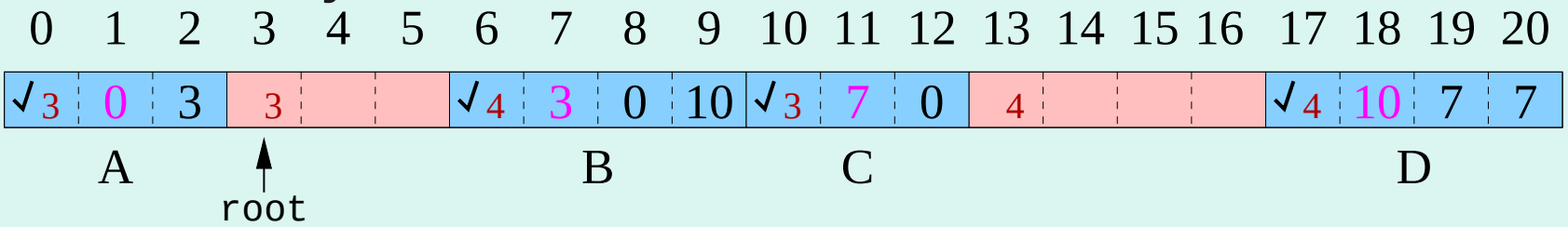
- Après marquage (racine = 6) :



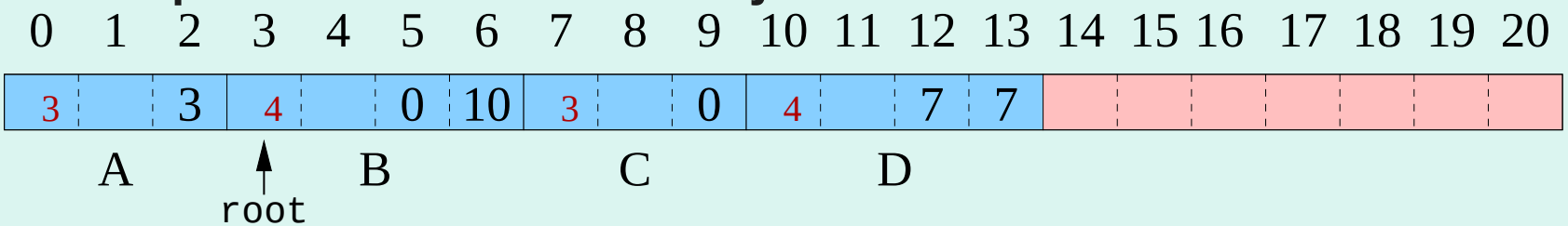
- Après calcul des forwarding pointers :



- Après mise à jour des références :



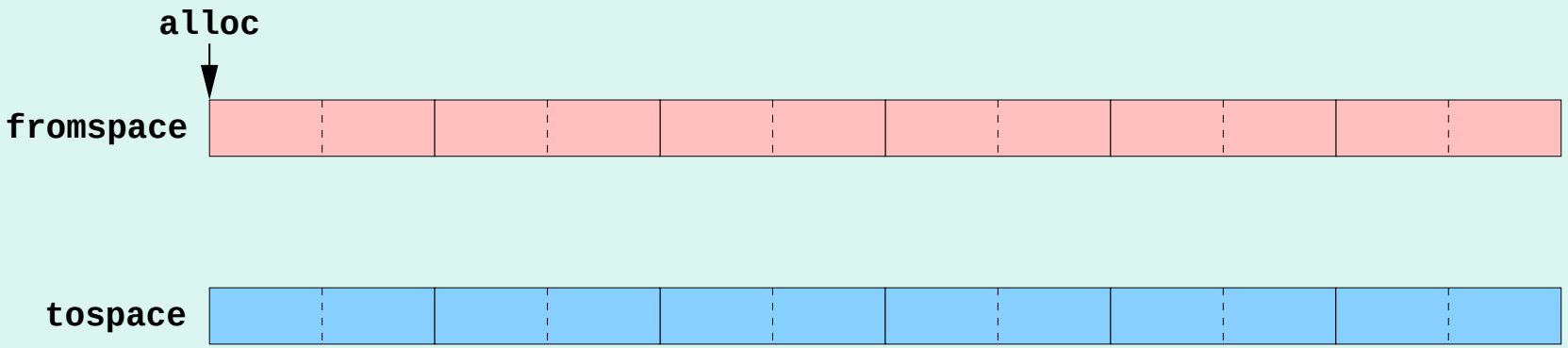
- Après déplacement des objets :



Stop and copy (1)

- Ce type de GC utilise l'algorithme de Cheney (1970) qui combine le marquage et la compaction

- Utilise 2 zones mémoire de même taille :
  - **fromspace** : zone contenant les objets
  - **tospace** : zone initialement vide dans laquelle sont copiés les objets vivants



Stop and copy (2)

- 2 pointeurs dans le tospace :
  - `copy` : où les objets vivants sont copiés
  - `scan` : limite des objets dont les références contenues ont été copiés

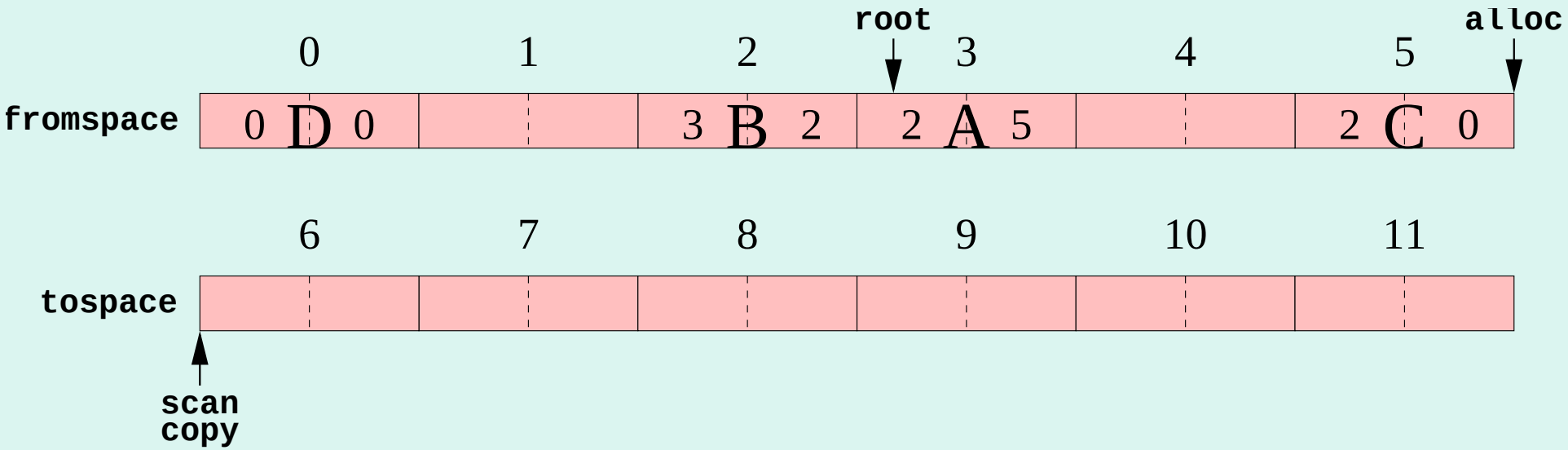
- Le GC termine lorsque `scan` = `copy`

- Aucun espace supplémentaire requis pour le marquage

- Besoin de **“broken heart”** et **“forwarding pointer”**

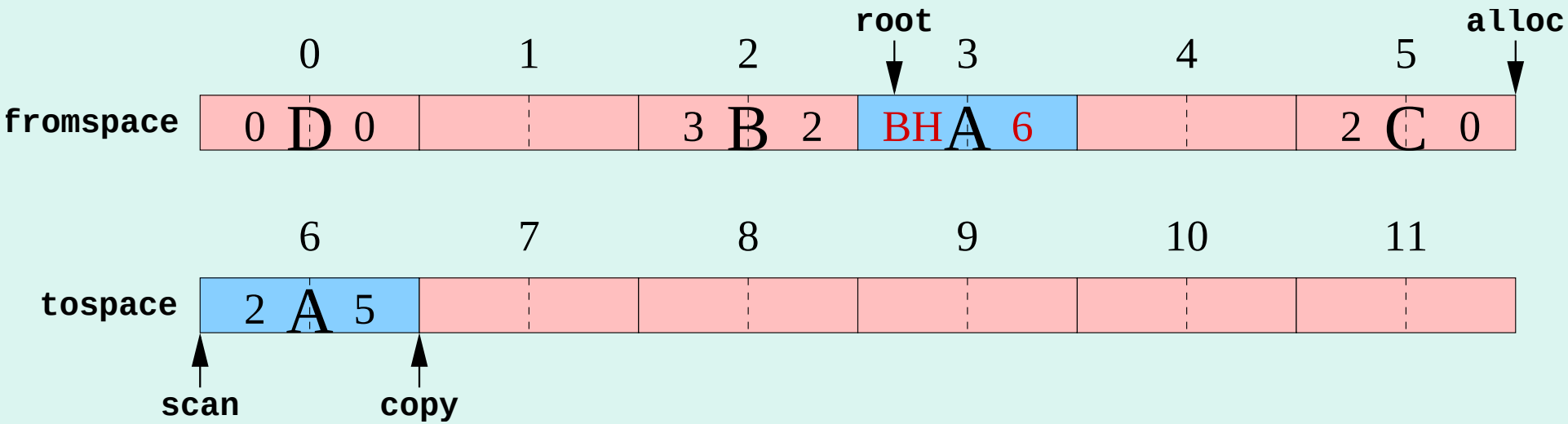
Stop and copy (3)

- Début du GC :



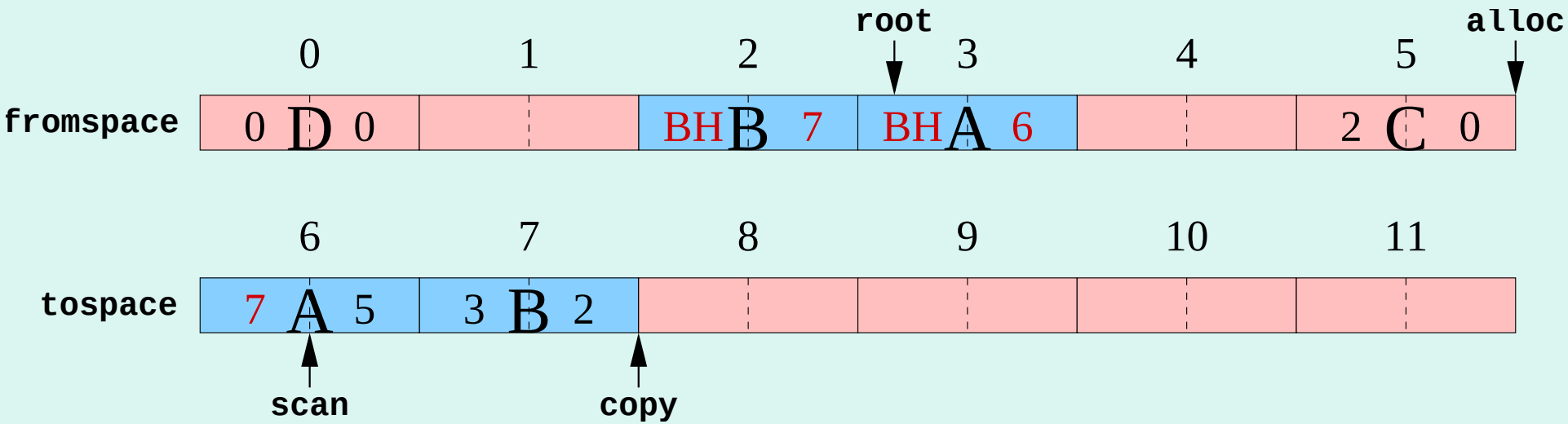
Stop and copy (3)

- Après prochaine étape :



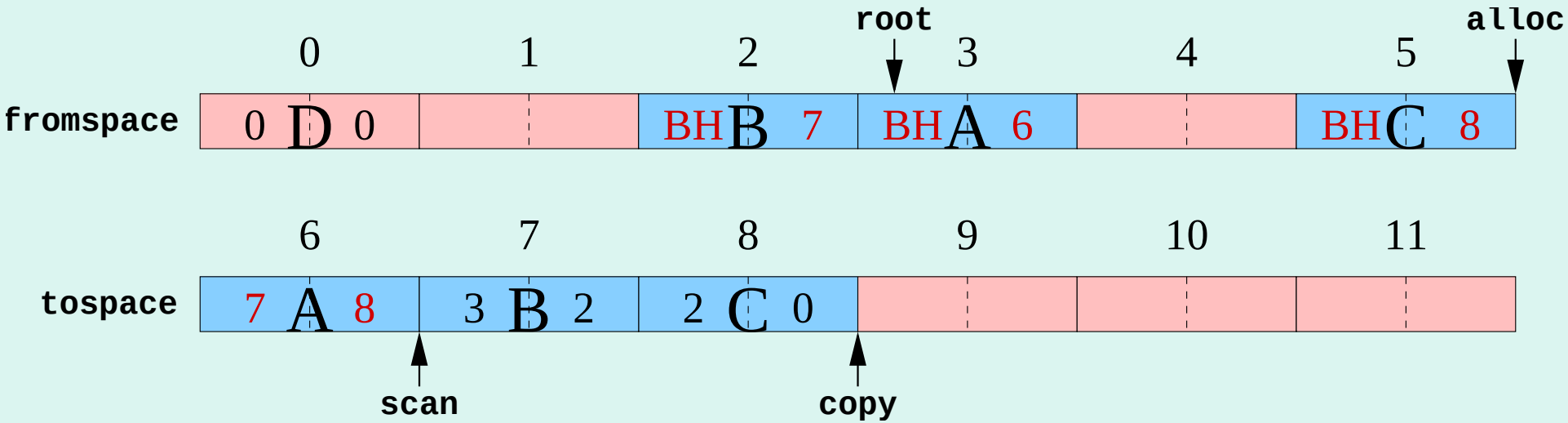
Stop and copy (4)

- Après prochaine étape :



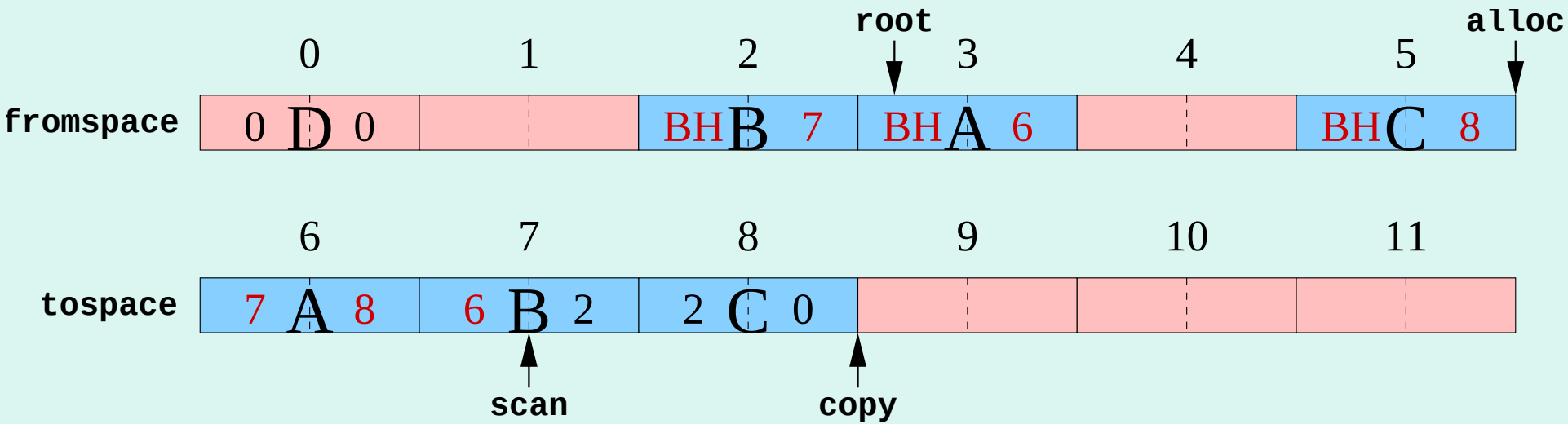
Stop and copy (5)

- Après prochaine étape :



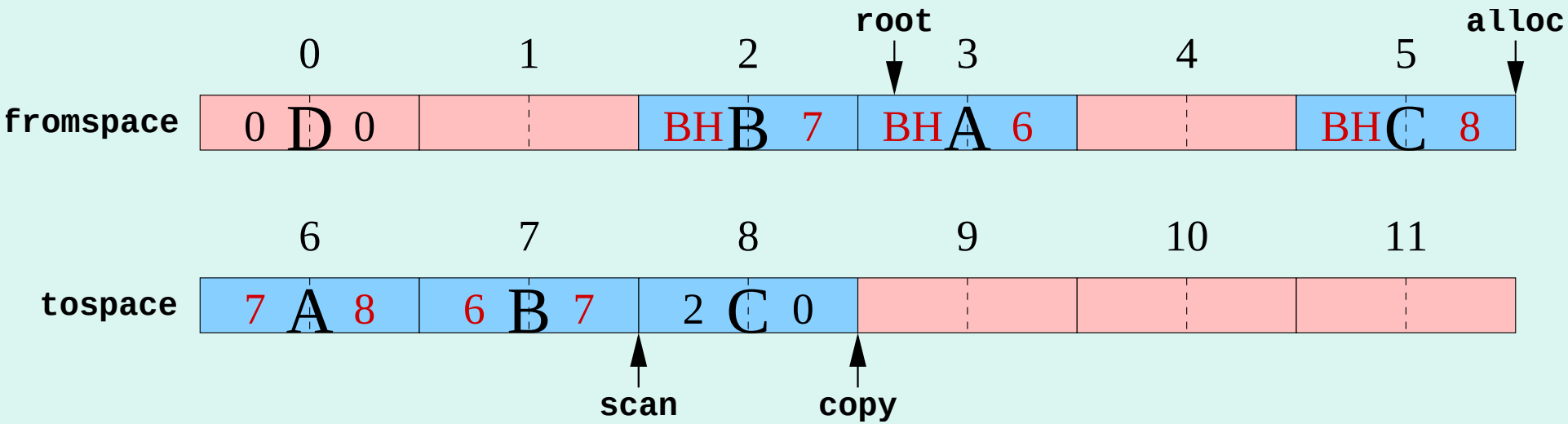
Stop and copy (6)

- Après prochaine étape :



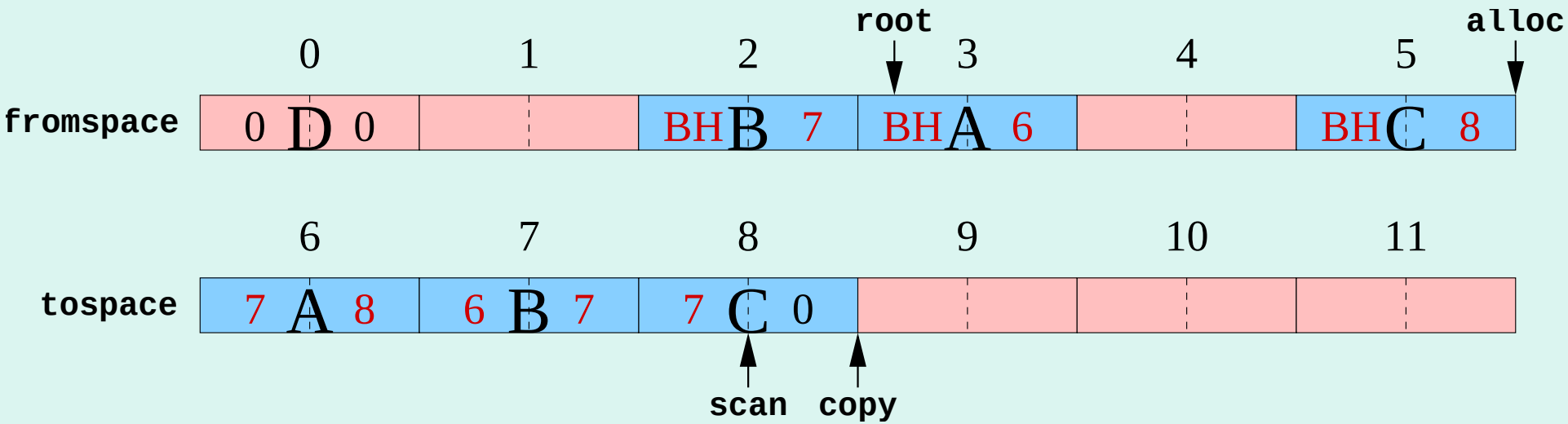
Stop and copy (7)

- Après prochaine étape :



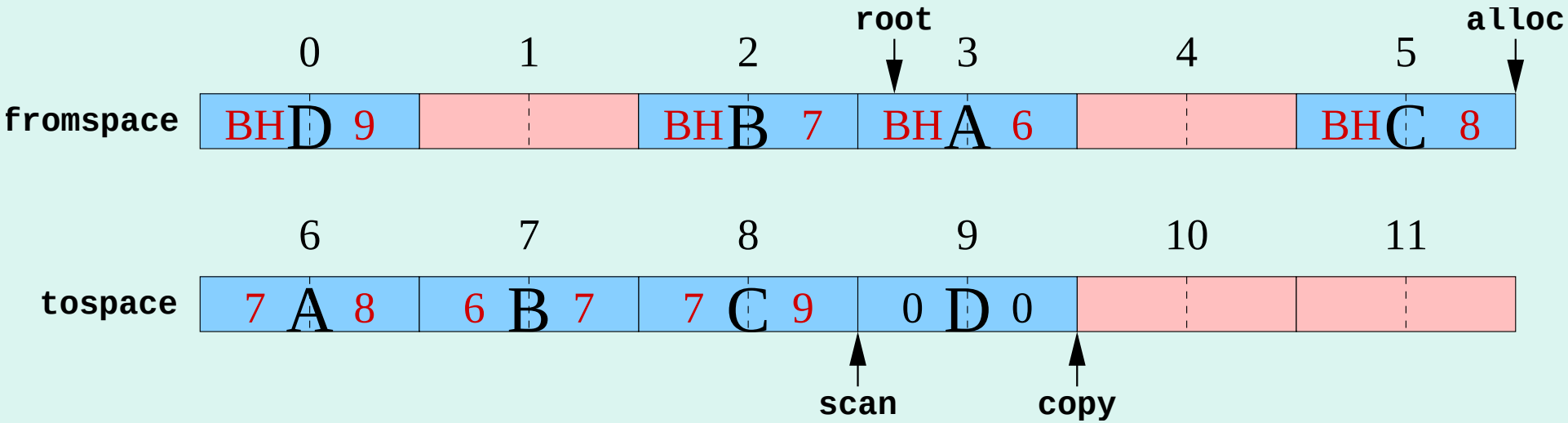
Stop and copy (8)

- Après prochaine étape :



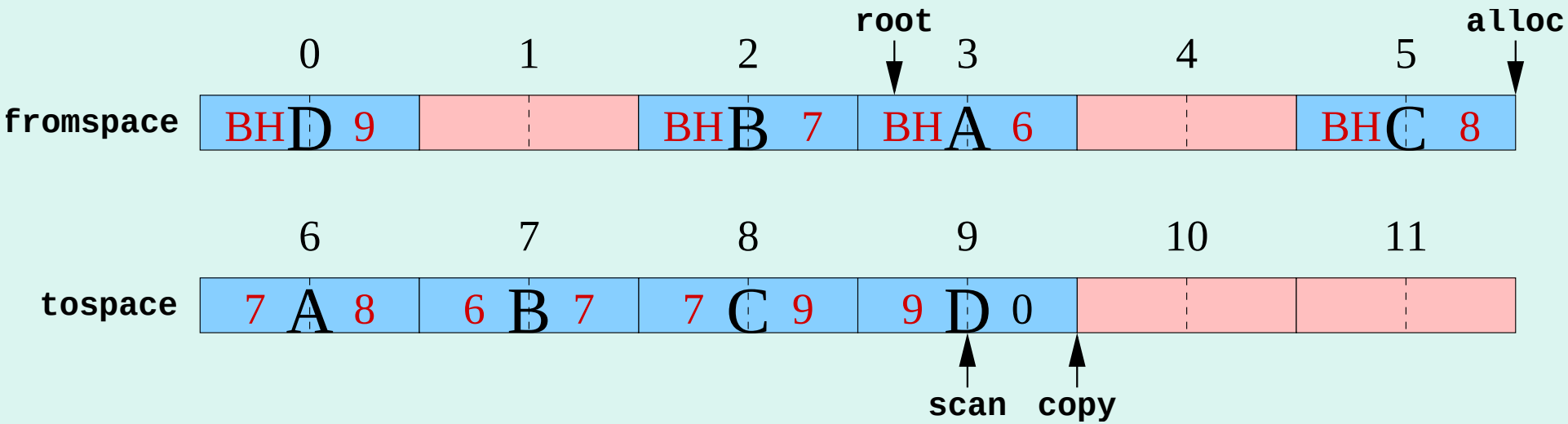
Stop and copy (9)

- Après prochaine étape :



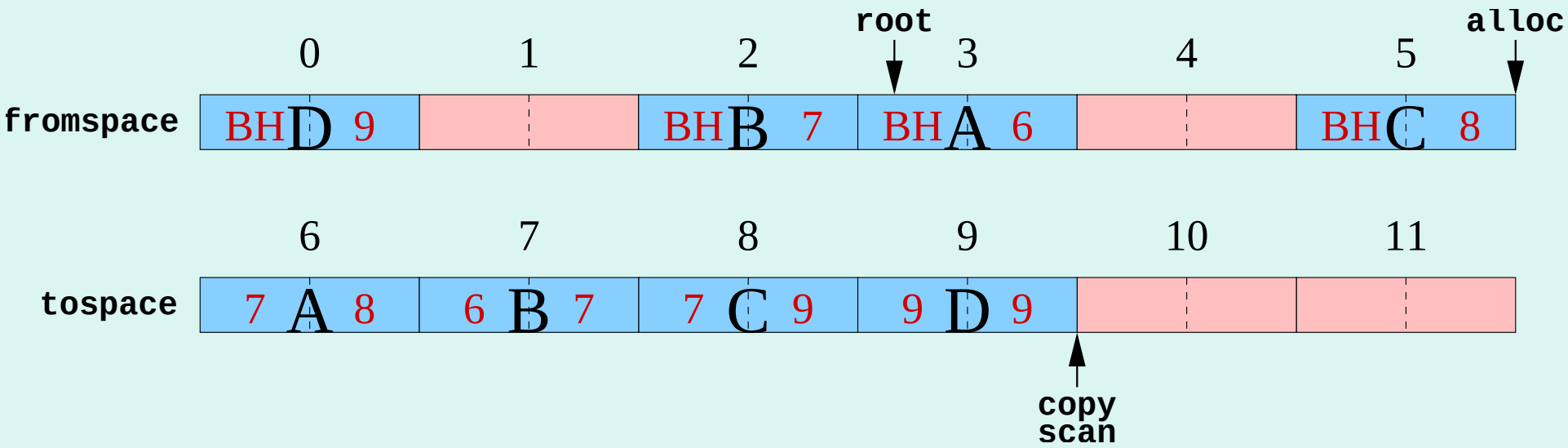
Stop and copy (10)

- Après prochaine étape :



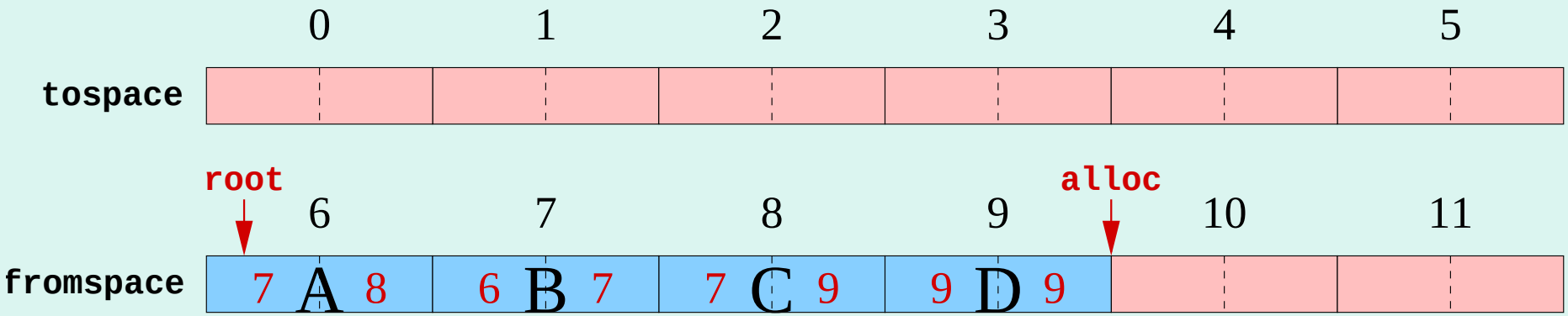
Stop and copy (11)

- Après prochaine étape :



Stop and copy (12)

- Après prochaine étape :



Performance de stop and copy

- Soit H la taille de la mémoire (“heap”) et L l’espace occupé par les objets vivants à la fin du GC (“live”)

- GC M&S et M&C
  - Temps d’allocation : $O(H)$
  - Temps de récupération : $O(H)$
  - grande constante cachée

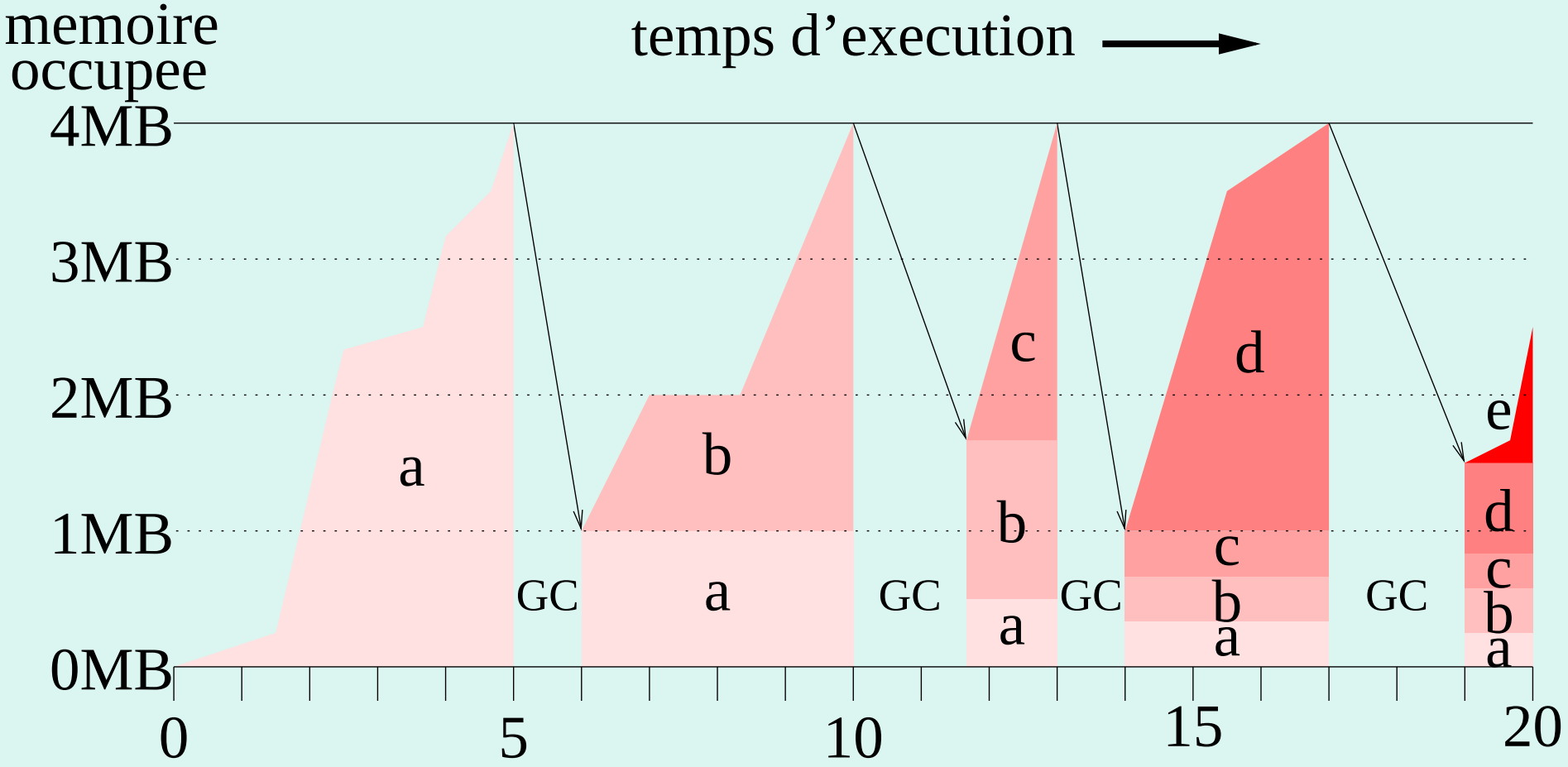
- GC S&C
  - Temps d’allocation : $O(H)$
  - Temps de récupération : $O(L)$
  - petite constante cachée
  - très bon lorsque L est une fraction de H

GC générationel

- Découpe l'espace mémoire en **générations**, qui sont des zones contenant chacune des objets de même age approximatif
- Le mutateur alloue les objets dans la première génération (**pouponnière** ou "nursery")
- Lorsque l'objet a atteint un certain age, il est **promu** à la prochaine génération (normalement cette opération se fait en allouant de l'espace dans la prochaine génération, ce qui peut entrainer de faire un GC de cette génération)

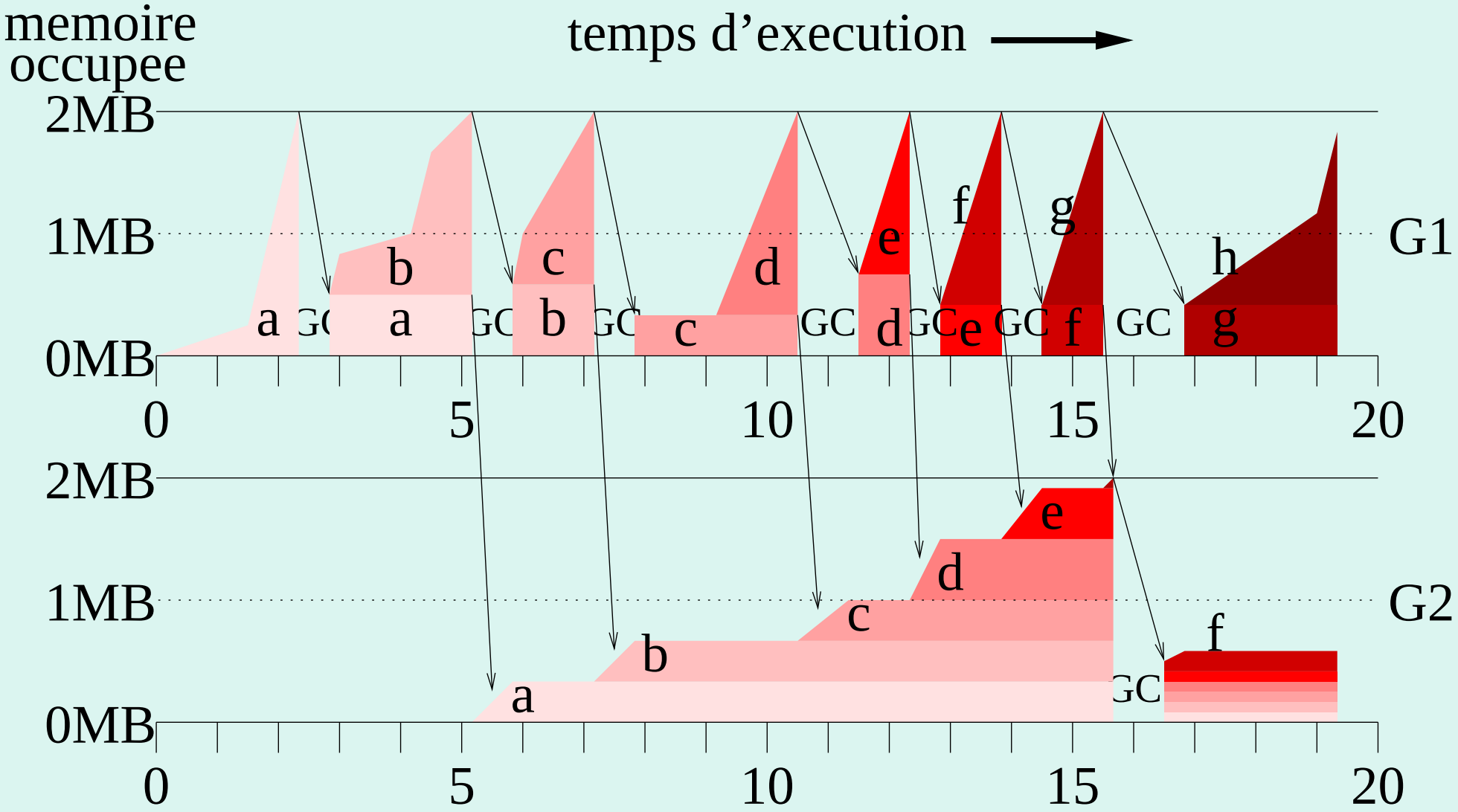
Profil d'allocation d'un GC compactant

- Supposons que le GC déplace les objets mais préserve leur ordre



Profil d'allocation GC générationel

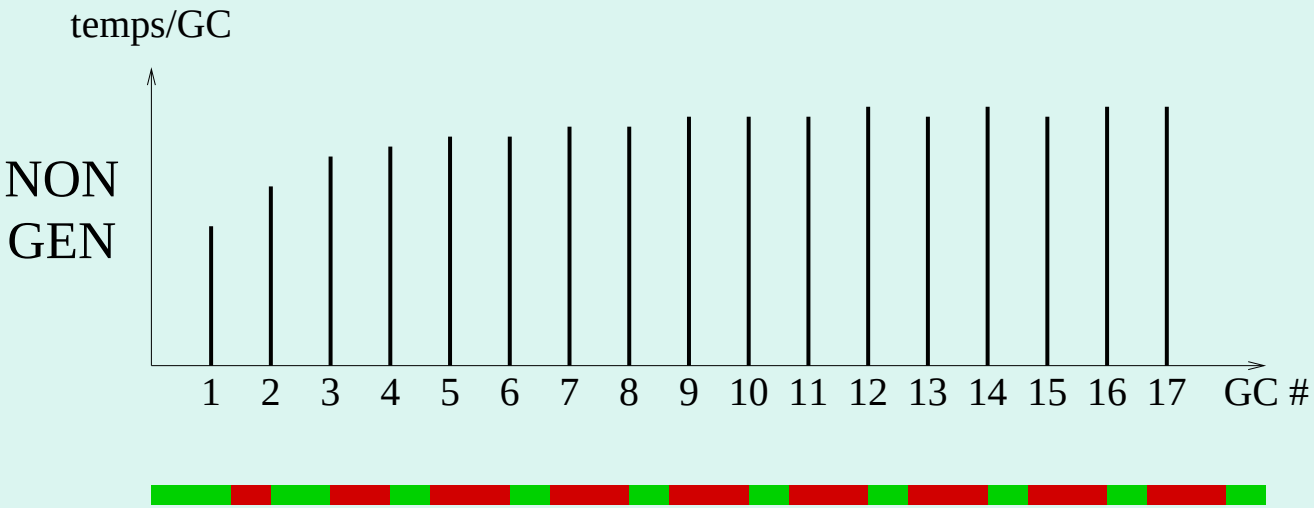
- GC générationnel à 2 générations

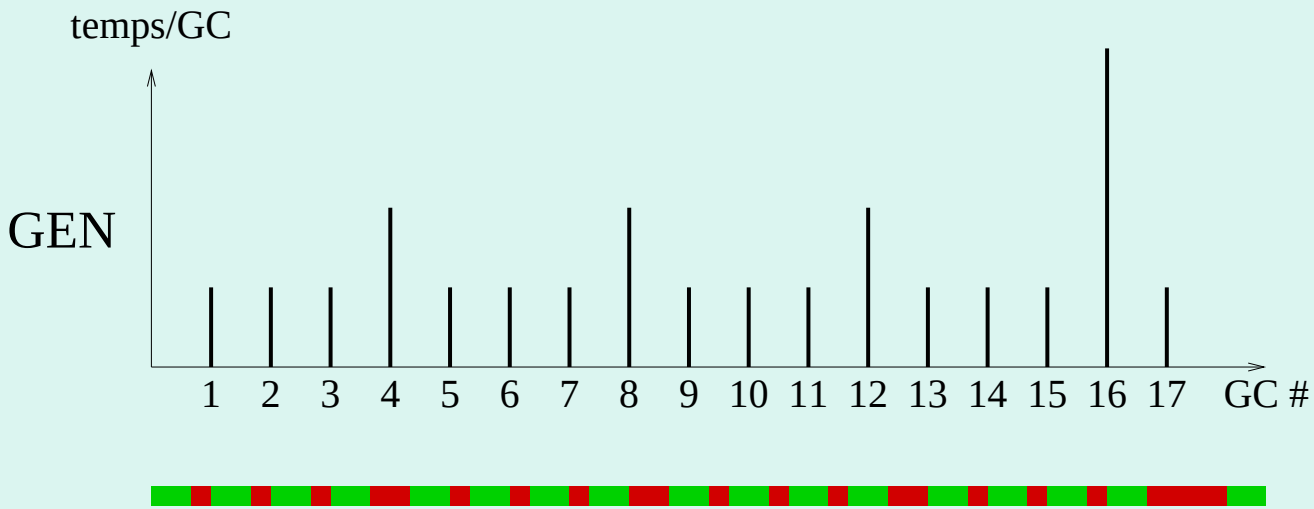


Considérations (1)

- Normalement les générations **plus jeunes sont plus petites que les plus vieilles**
- De plus, si la pouponnière est assez petite pour entrer **en cache** on améliore son efficacité
- Cela donne des **pauses de GC courtes en moyenne** (quasi-incrémentiel)
- Le pire cas donne une pause de GC plus longue qu'un GC non-générationel (i.e. GC de toutes les générations)

Considérations (2)





Considérations (3)

- Chaque génération peut avoir son propre algorithme d'allocation et de récupération

- Pour la pouponnière le **GC compactant S&C** est un bon choix car il est rapide (la moins bonne consommation d'espace de S&C est négligeable puisque la pouponnière est petite)

- Pour la plus vieille génération on peut utiliser un **GC non-copiant**, pour éviter de déplacer les objets "permanents" (ou de très longue vie) du programme

Politiques de promotion

- Simple : promotion de **tous les objets** à chaque GC d'une génération
- Le problème c'est que l'age des objets promus varie beaucoup (certains viennent juste d'être créés)
- L'age peut être **stocké dans un champ de l'objet** initialisé à 0 et incrémenté à chaque cycle de GC sans promotion
- **GC compactant par "glissement"** : l'age peut être maintenu avec une table qui indique les frontières entre chaque groupe
- **GC compactant S&C** : les objets qui ont survécu à 1 GC sont distinguables par leur adresse de ceux qui en on survécu plus qu'un

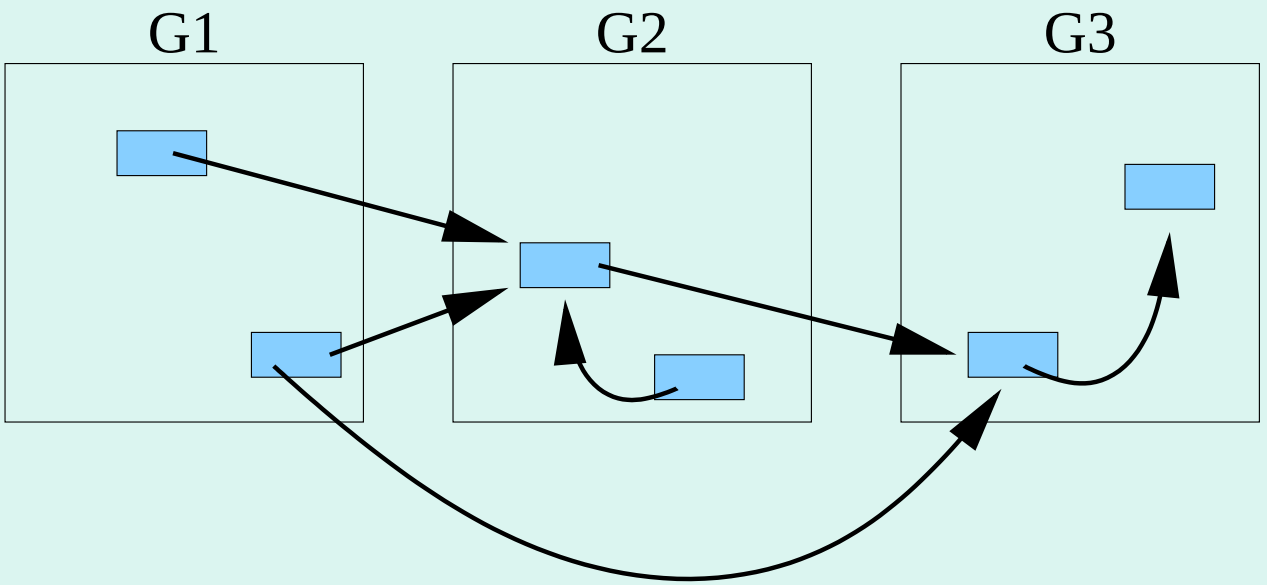
Efficacité du GC générationel

- Est-ce efficace?
- Des études ont démontré que souvent
  - Les objets **meurent jeunes** (i.e. 80-98% des objets meurent avant qu'un MB de mémoire soit alloué)
  - Une forte proportion des **objets qui survivent un cycle de GC en survivent plusieurs**
- Exemple : `(map sin (map sqrt lst))`
- Puisque les objets jeunes meurent rapidement, il est avantageux de concentrer les efforts de récupération sur les jeunes générations

Références dans un GC générationel (1)

- Si on suit un style purement fonctionnel (pas de mutations) alors un objet X dans une génération N peut seulement contenir des références vers des objets dans les générations $\geq N$

- Les racines peuvent référer à n'importe quelle génération (i.e. en quelquesorte les racines sont dans la génération 1)



Références dans un GC générationel (2)

- Pourquoi?

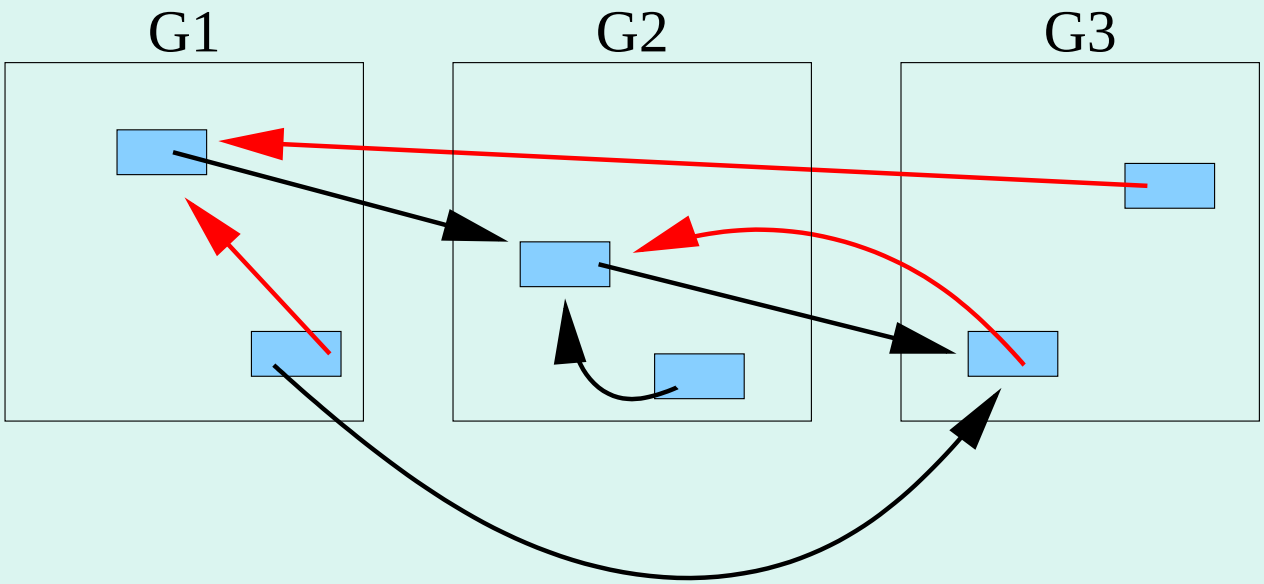
- Considérons le code suivant qui construit une paire :

```
(let* ((a ...) (b ...))
  (let ((x (cons a b)))
    ...))
```

- Pour construire un objet X il faut déjà avoir alloué les objets auxquels X va référer, c'est-à-dire **un objet est toujours plus jeune que les objets auxquels il réfère**

Références dans un GC générationel (3)

- Il se pourrait qu'il y ait des **références inverses** (un objet d'une génération N qui réfère à un objet d'une génération $< N$) lorsque
  - L'affectation est permise
  - La promotion ne respecte pas l'age exactement



Références inverses (1)

- Les **références inverses** sont problématiques car
  - Pour savoir quels objets de la génération 1 sont toujours vivants, **il faut examiner les autres générations**
  - En général pour faire le GC de la génération N, il faut tenir compte des générations $> N$
- Il y aurait peu de gains en performance s'il fallait parcourit toutes les générations
- Le GC peut se limiter à examiner les références inverses des générations $> N$ en **supposant quelles sont toutes dans des objets vivants** (supposition conservatrice)

Références inverses (2)

- Il faut donc conserver avec chaque génération un ensemble des emplacements contenant (possiblement) des références inverses
  - Pour les traiter comme des racines pour le GC des générations plus jeunes
  - Pour les mettre à jour lorsqu'on déplace les objets référés
- Cet ensemble est normalement mis-à-jour lors des mutations (d'où le nom "write barrier")

Write barrier

- Mise-à-jour des ensembles de références inverses :

```
write_barrier(Obj,Field,Val) // do: Obj.Field = Val
{
  Obj.Field = Val
  if is_object(Val) then
    if gen(Val) < gen(Obj) then
      add &Obj.Field to inverse_refs[gen(Obj)]
}

set_car(Pair,Val) { write_barrier(Pair,Car,Val) }
```

- Attention : **cette mise-à-jour alloue de l'espace!** Il faut donc prendre le point de vue que les affectations sont des opérations d'allocation qui peuvent causer un GC!

Variantes (1)

- Conserver l'ensemble des objets contenant possiblement des références inverses (**"remembered sets"**)

```
write_barrier(Obj,Field,Val) // do: Obj.Field = Val
{
  Obj.Field = Val
  if is_object(Val) then
    if gen(Val) < gen(Obj) then
      add Obj to remembered_set[gen(Obj)]
}
```

- Cela est plus **conservateur** mais demande moins d'espace

Variantes (2)

- Conserver toutes les mutations inconditionnellement dans une table de taille fixe (**“sequential store buffer”**)

- Lorsque la table déborde faire un **GC de la table** pour éliminer les entrées qui ne correspondent pas à des références inverses ou qui sont dupliquées

```
write_barrier(Obj,Field,Val) // do: Obj.Field = Val
{
  Obj.Field = Val
  SSB[ssb_alloc++] = &Obj.Field
  if ssb_alloc == ssb_size then SSB_gc();
}
```

- Si la table est trop pleine après le GC de la table, les ajouter au “remembered set”

Variantes (3)

- **“Card marking”** : utilise un bitmap pour savoir quels emplacements en mémoire peuvent contenir des références inverses
- Chaque “carte” est une zone de taille 2^k octets
- Toute affectation lève le bit approprié du bitmap; le GC traverse le bitmap; chaque bit à 1 causera un scan des références dans la carte
- k **petit** : traverse du bitmap **lent**, scan **rapide**
- k **grand** : traverse du bitmap **rapide**, scan **lent**

Variantes (4)

- **Write-barrier de Chambers** utilise un byte map :

```
st  [%obj + offset], %ptr      ; code SPARC
add %obj, offset, %temp
sll %temp, k, %temp
st  %g0, [%byte_map + %temp]
```

- **Write-barrier de Holzle** est moins précis mais plus rapide :

```
st  [%obj + offset], %ptr      ; code SPARC
sll %obj, k, %temp
st  %g0, [%byte_map + %temp]
```

Modèle trois-couleur (1)

- Abstraction du processus de GC inventé par Dijskstra pour expliquer/comparer les algorithmes de GC incrémentiels à un haut-niveau

- On associe l'une de 3 couleurs aux objets :
  - **Noir** : l'objet et ses descendants immédiats ont été visité par le GC et l'objet n'a plus besoin d'être visité à nouveau
  - **Gris** : l'objet a besoin d'être visité par le GC
  - **Blanc** : l'objet n'a pas encore été visité par le GC; s'il est toujours blanc à la fin du GC c'est qu'il est **mort**

Modèle trois-couleur (2)

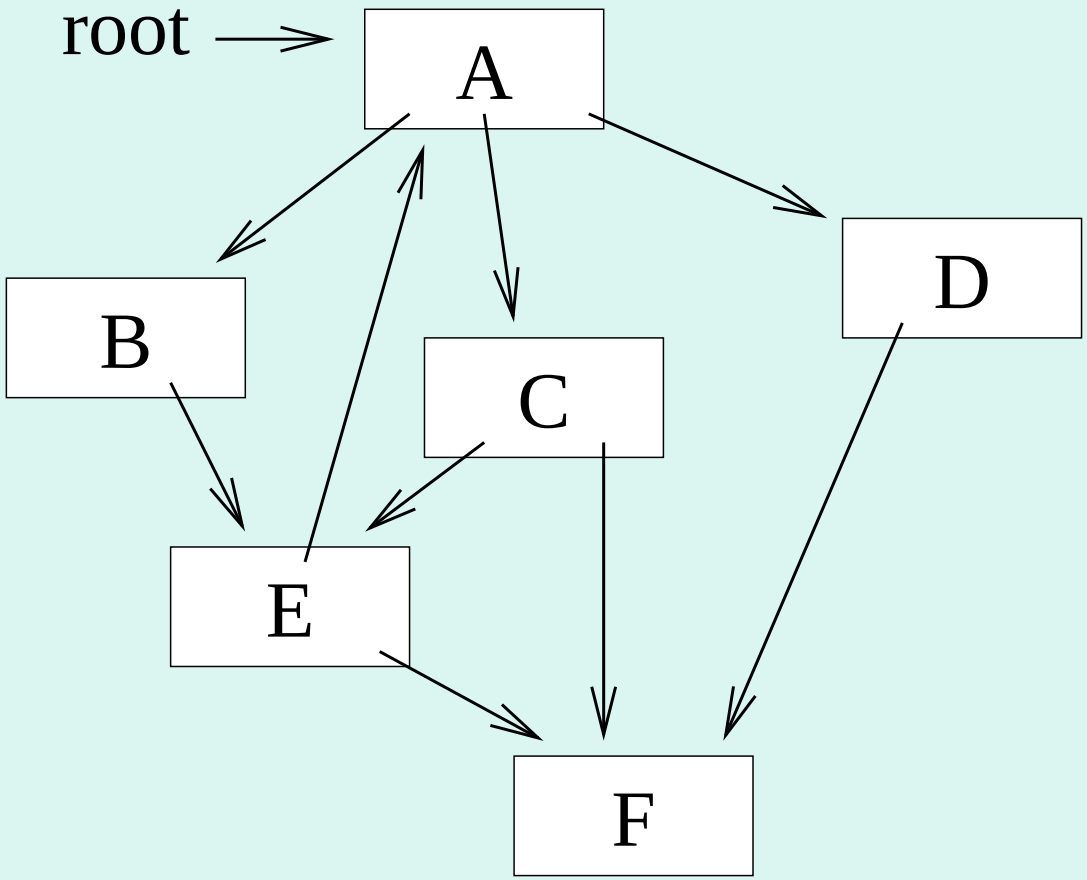
- Algorithme de GC “générique” :

  1. Tous les objets débutent en **blanc**
  2. Colorier les racines en **gris**
  3. Tant qu’il y a des objets gris :
     - Choisir un objet gris X
     - S’il y a un descendant Y de X qui est **blanc** alors colorier Y en **gris**
     - Sinon colorier X en **noir**

- Les objets **gris** sont donc ceux pour lesquels il reste du travail à faire (on peut imaginer qu’on les garde sur une “work list”)

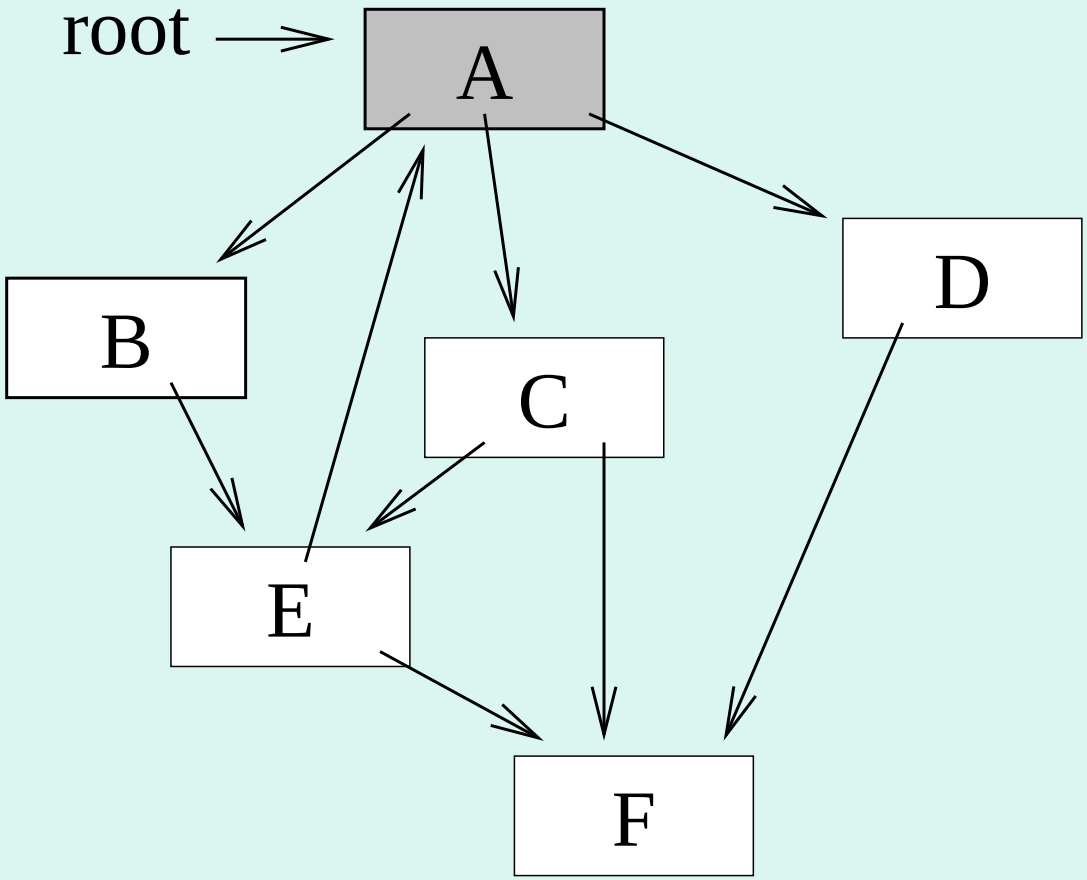
Modèle trois-couleur (3)

- Exemple :



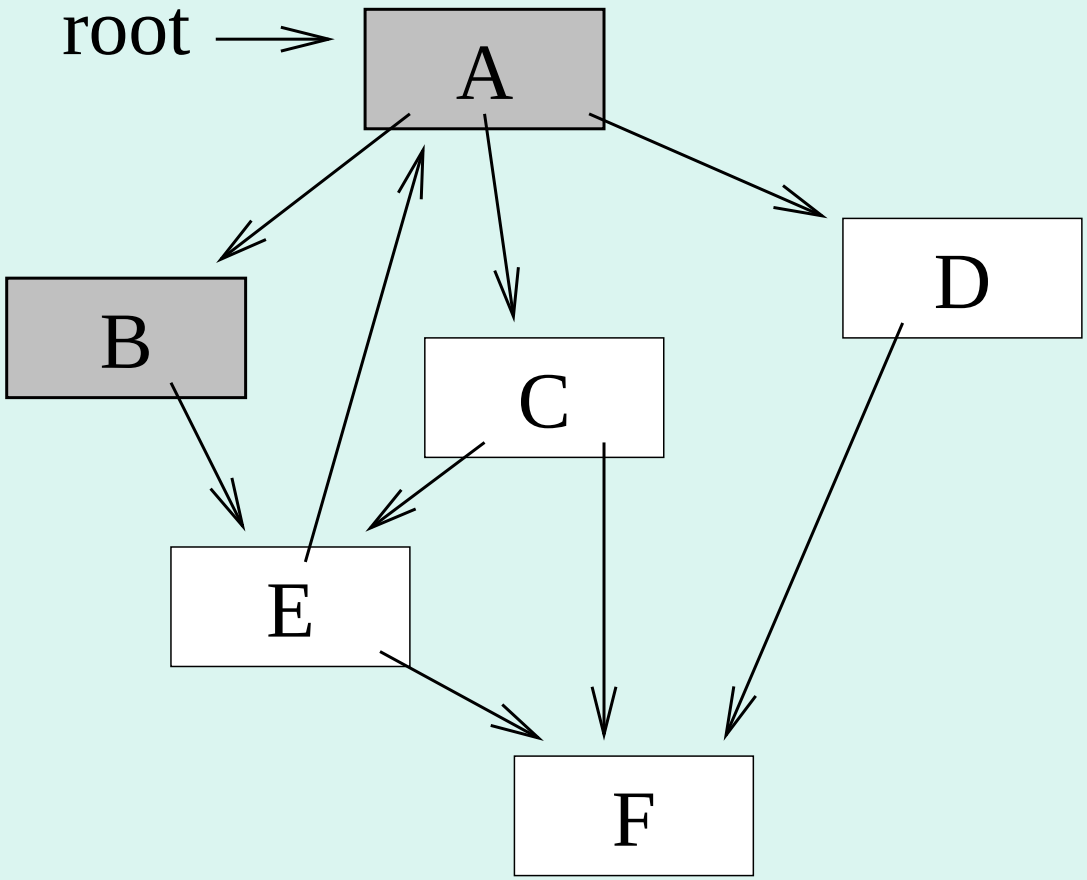
Modèle trois-couleur (4)

- Exemple :



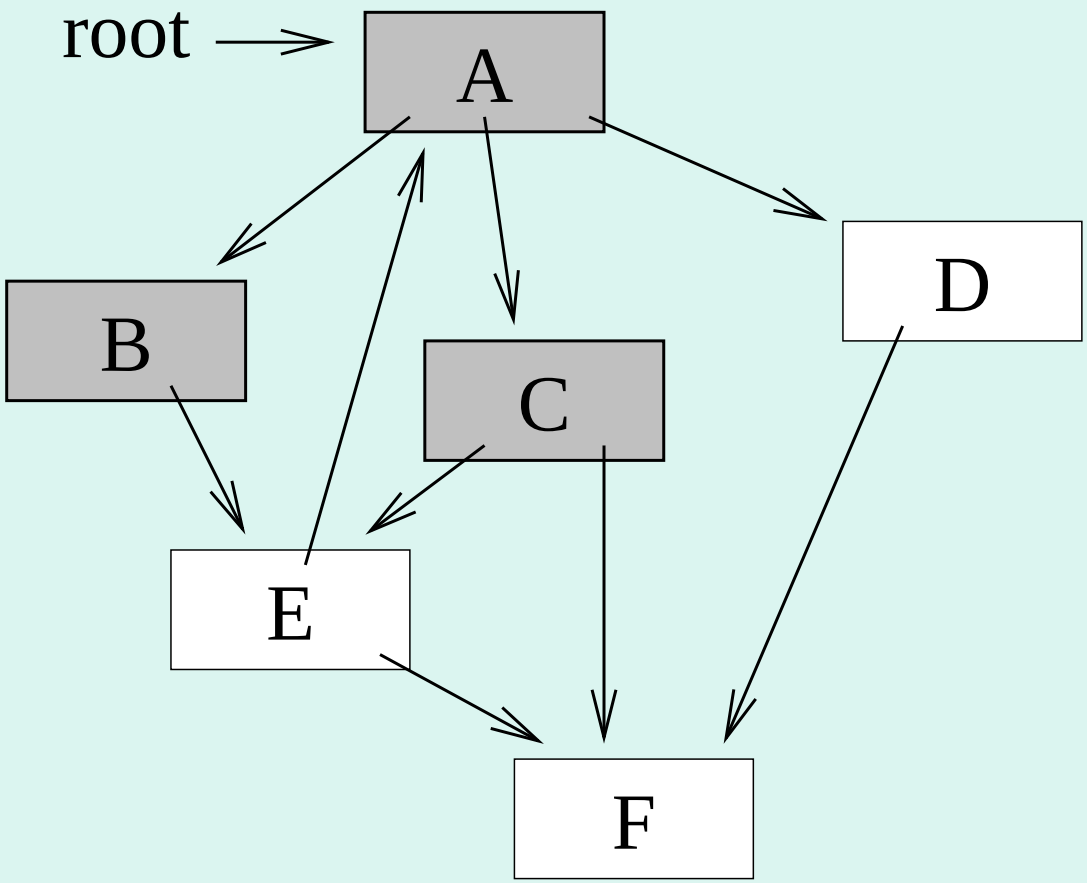
Modèle trois-couleur (5)

- Exemple :



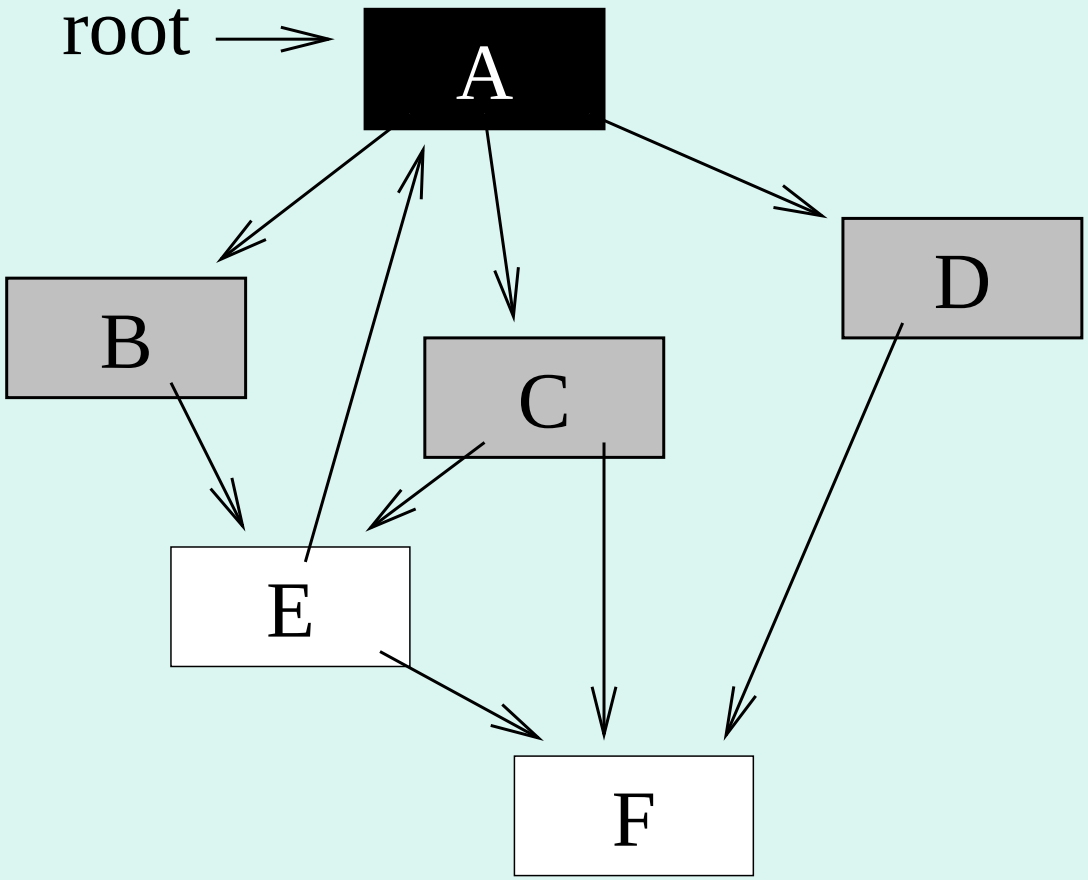
Modèle trois-couleur (6)

- Exemple :



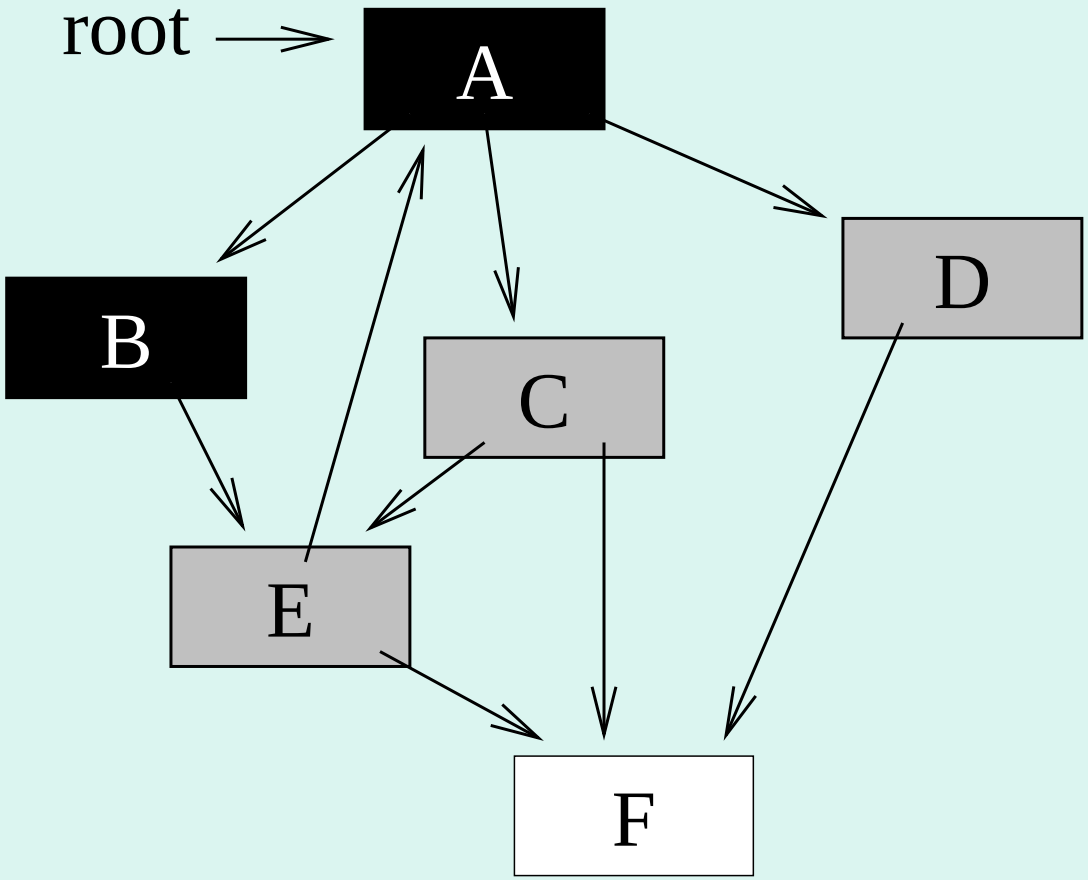
Modèle trois-couleur (7)

- Exemple :



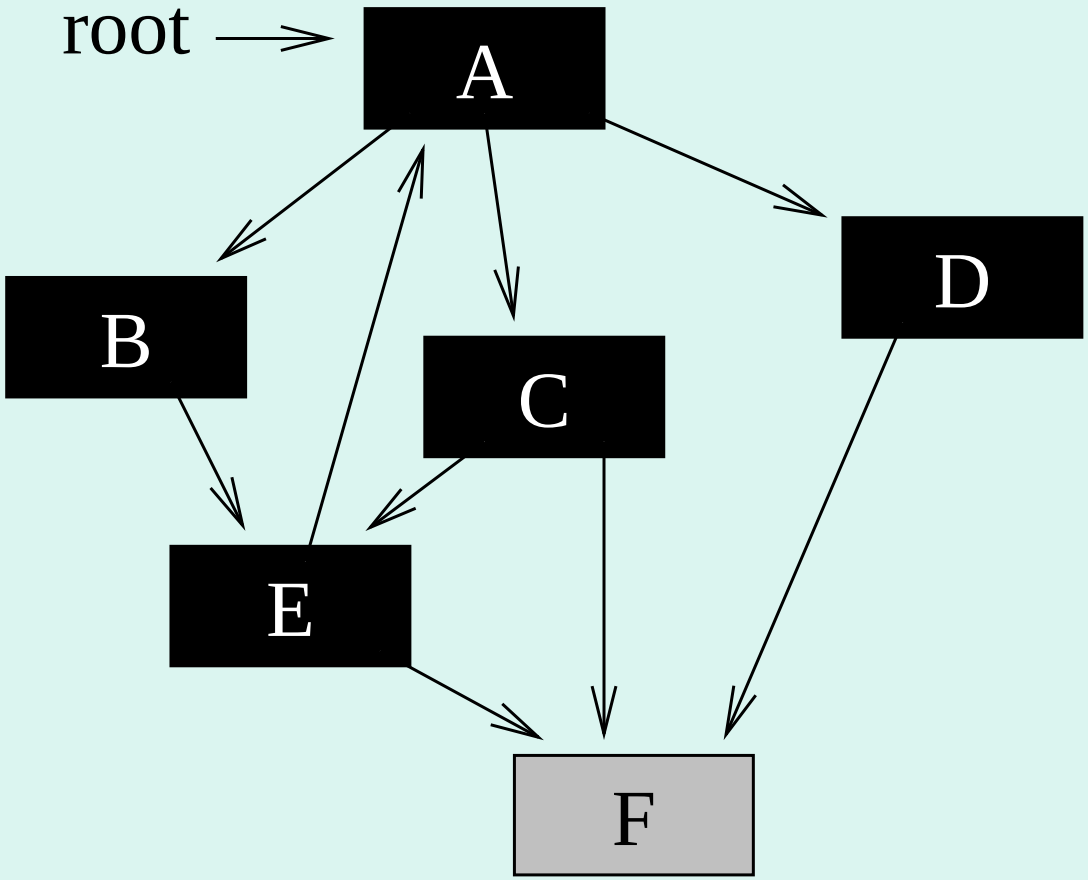
Modèle trois-couleur (8)

- Exemple :



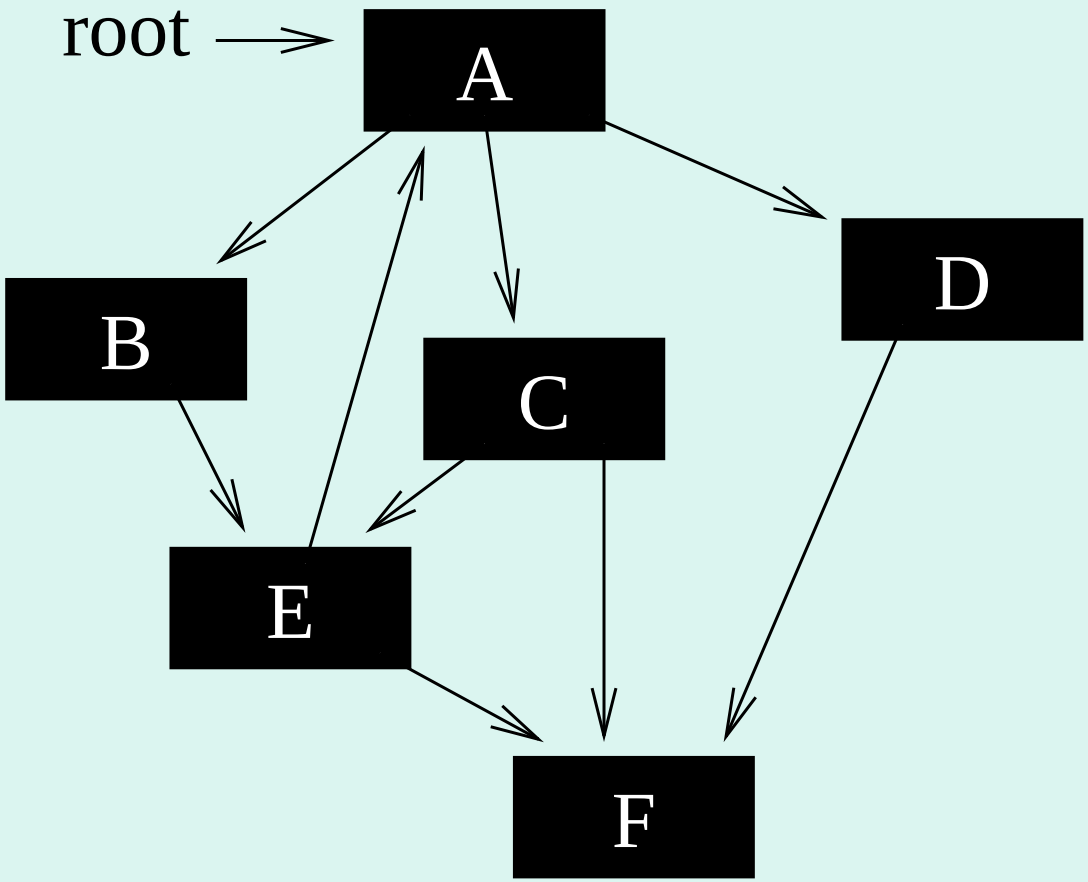
Modèle trois-couleur (9)

- Exemple :



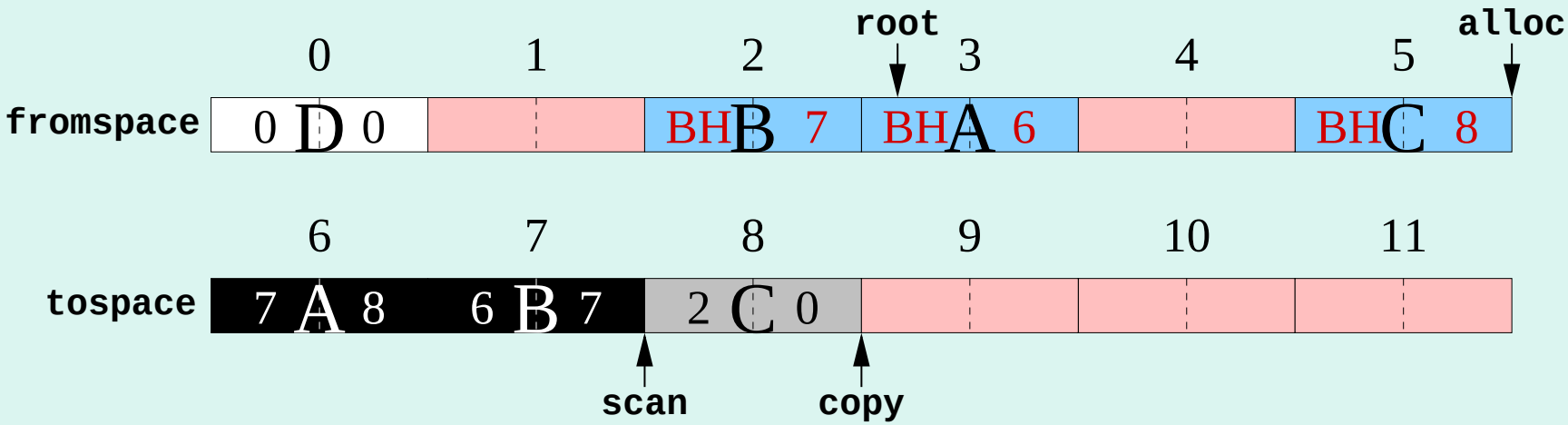
Modèle trois-couleur (10)

- Exemple :



Modèle trois-couleur (11)

- Le détail de l'implantation des "couleurs" varie d'un GC à un autre
- Pour le GC S&C :

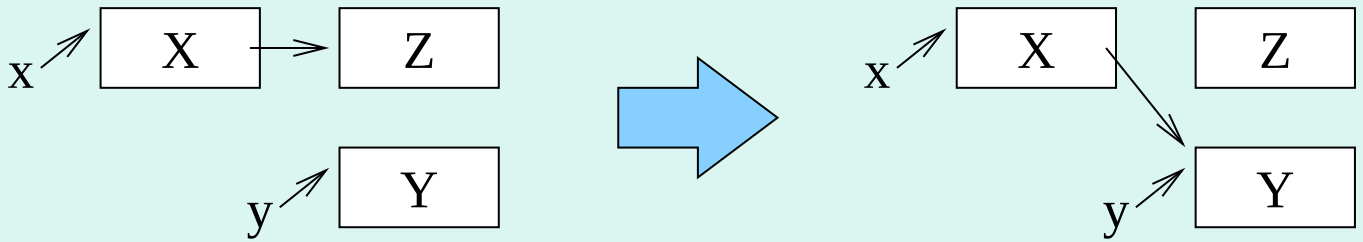


GC incrémentiels (1)

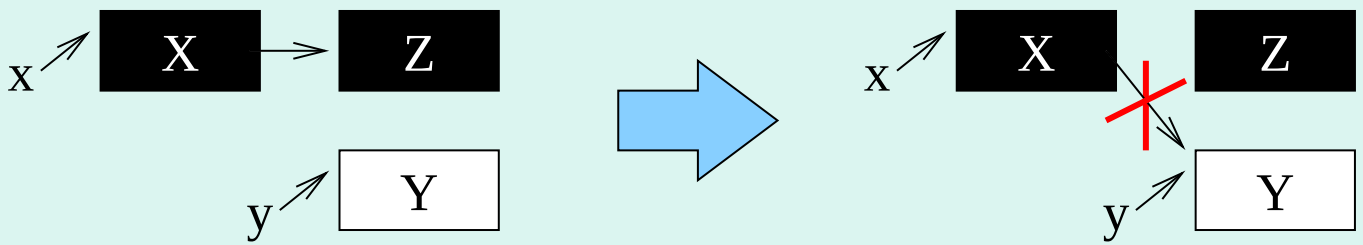
- Dans un GC incrémentiel le mutateur et récupérateur sont conceptuellement ou réellement **concurrents**
- Donc en général le mutateur **modifie le graphe des objets pendant son parcours par le récupérateur**

GC incrémentiels (2)

- Cela peut modifier l'accessibilité des objets et il ne faut pas que cela viole l'**invariant de coloration**
- `set_cdr(x,y)` : OK

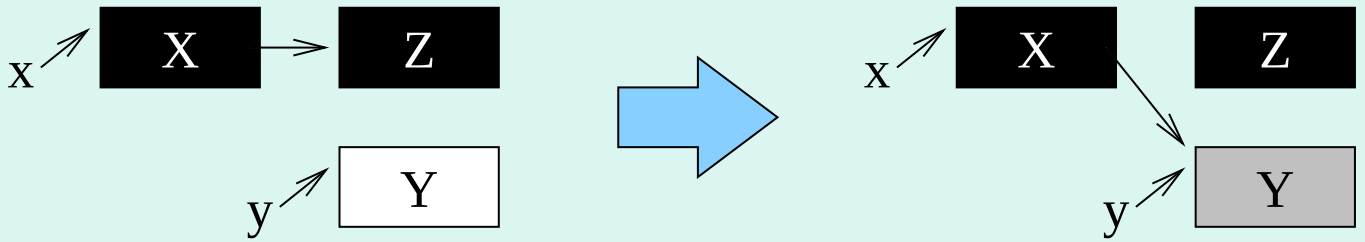


- `set_cdr(x,y)` : PAS OK car cacherait Y du GC



GC incrémentiels (3)

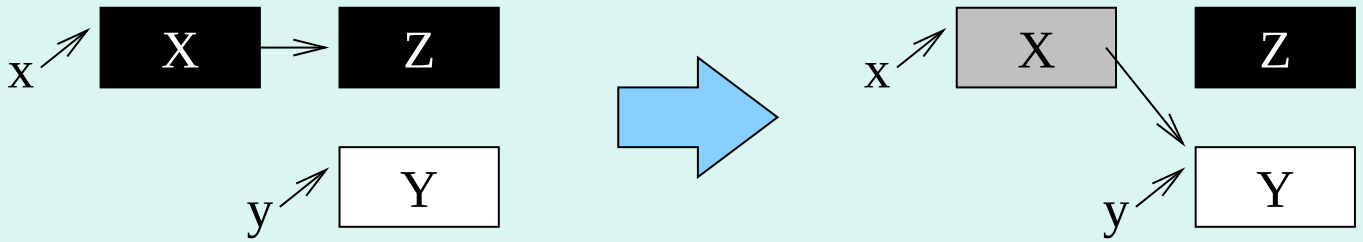
- **Write-barrier de Dijskstra** : colorier y en **gris**
- `set_cdr(x,y)`



- Y se fait donc ajouter à la work list du GC car il est clairement vivant présentement

GC incrémentiels (4)

- **Write-barrier de Steele** : colorier x en **gris**
- `set_cdr(x,y)`



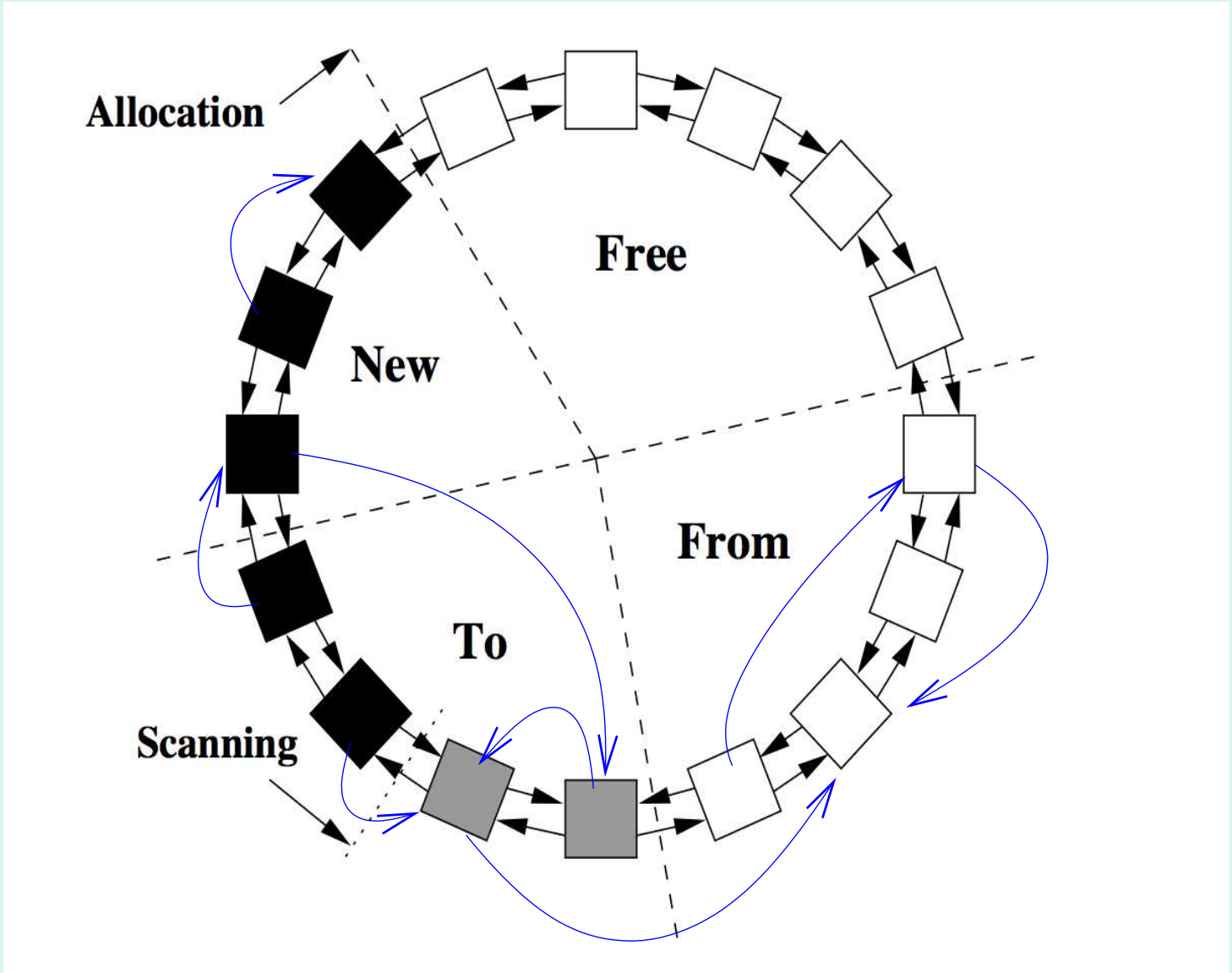
- X se fait remettre sur la work list du GC (négatif : plus de travail)
- Y a le temps de mourir avant que X se fasse traiter par le GC (positif : récupération d'un plus grand nombre d'objets morts)

GC incrémentiels (5)

- Un read-barrier peut être utilisé au lieu d'un write-barrier
- **Read-barrier de Baker** : si le mutateur lit une référence x vers un objet blanc, le mutateur colorie l'objet en **gris**
- Corollaire : dans `set_cdr(x,y)` les objets `x` et `y` sont soit gris ou noirs
- Le mutateur ne peut donc jamais violer l'invariant de coloration

Baker's Treadmill

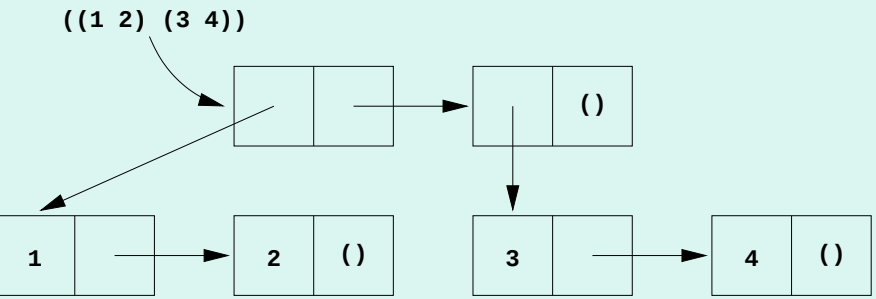
- GC incrémentiel trois-couleur "in situ"

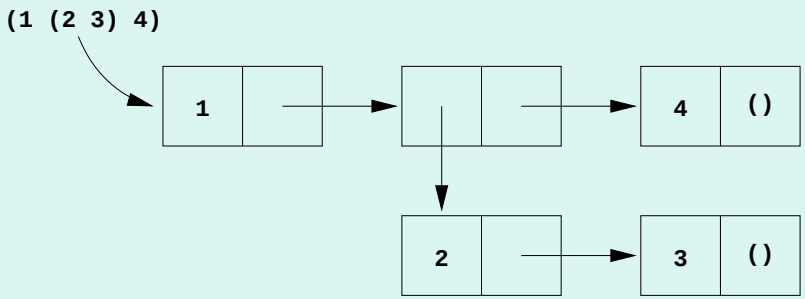


Continuations (1)

- Le problème de **“same fringe”** consiste à vérifier si deux arbres ont les mêmes feuilles (dans le même ordre dans un parcours gauche à droite)
- Les arbres peuvent avoir des formes différentes
- En Scheme nous représenterons les arbres avec des listes

```
(same-fringe? '((1 2) (3 4))
              '(1 (2 3) 4)) => #t
```





Continuations (2)

- Nous cherchons aussi une implantation efficace : on aimerait faire une quantité de travail qui est proportionnelle à la position de la première différence
- Solution inefficace :

```
(define (same-fringe? x y)
  (equal? (flatten x) (flatten y)))

(define (flatten t)
  (cond ((null? t)
         '())
        ((pair? t)
         (append (flatten (car t))
                 (flatten (cdr t))))
        (else
         (list t))))
```

Continuations (3)

- Type de `flatten` ?

  `flatten` :: $arbre \rightarrow liste$
  $liste$ = (`(`$objet$ `.` $liste$`)` | `()`

- Complexité de `flatten` si n est le nombre de feuilles dans l'arbre?

- Rappel :

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

- Donc (`append` x y`)` est $O(|x|)$

Continuations (4)

- Pire cas : `(flatten '((((1) 2) ...)` n`))`
- $T(n) = T(n-1) + (n-1)$
- $T(n) = (n-1) + (n-2) + \ldots + 1$
- $T(n) = O(n^2)$
- Il faut éviter d'utiliser `append`
- Idée : combiner l'opération "flatten" avec "append"

```
(append-flatten t r) = (append (flatten t) r)

(flatten t) = (append-flatten t '())
```

Continuations (5)


```
(define (append-flatten t r)
  (cond ((null? t)
         r)
        ((pair? t)
         (append-flatten
          (car t)
          (append-flatten (cdr t) r)))
        (else
         (cons t r))))

(define (flatten t)
  (append-flatten t '()))
```

- Type de `append-flatten` ?

  `append-flatten` :: $arbre\ liste \rightarrow liste$

  $liste$ = ($objet$. $liste$) | ()

Continuations (6)


```
(define (same-fringe? x y)
  (same-list? (flatten x)
              (flatten y)))

(define (same-list? x y)
  (if (null? x)
      (null? y)
      (and (not (null? y))
           (equal? (car x) (car y))
           (same-list? (cdr x) (cdr y)))))
```

- Ce n'est pas efficace car le `flatten` des deux arbres se fait entièrement avant les comparaisons
- Idée : faire le `flatten` **incrémentalement** en retardant le calcul de la queue de la liste

Continuations (7)

- Utiliser une liste paresseuse :

  $listepar$ = ($objet$. $\rightarrow listepar$) | ( )

- Une liste paresseuse non-vide est une paire qui contient le premier élément de la liste (`car`) et une fonction sans paramètre qui retourne le reste de la liste paresseuse (`cdr`)

- Nouveau type de `append-flatten` et `flatten` ?

  `append-flatten` :: $arbre \rightarrow listepar \rightarrow listepar$
  `flatten` :: $arbre \rightarrow listepar$

Continuations (8)

append-flatten :: $arbre \rightarrow listepar \rightarrow listepar$

flatten :: $arbre \rightarrow listepar$

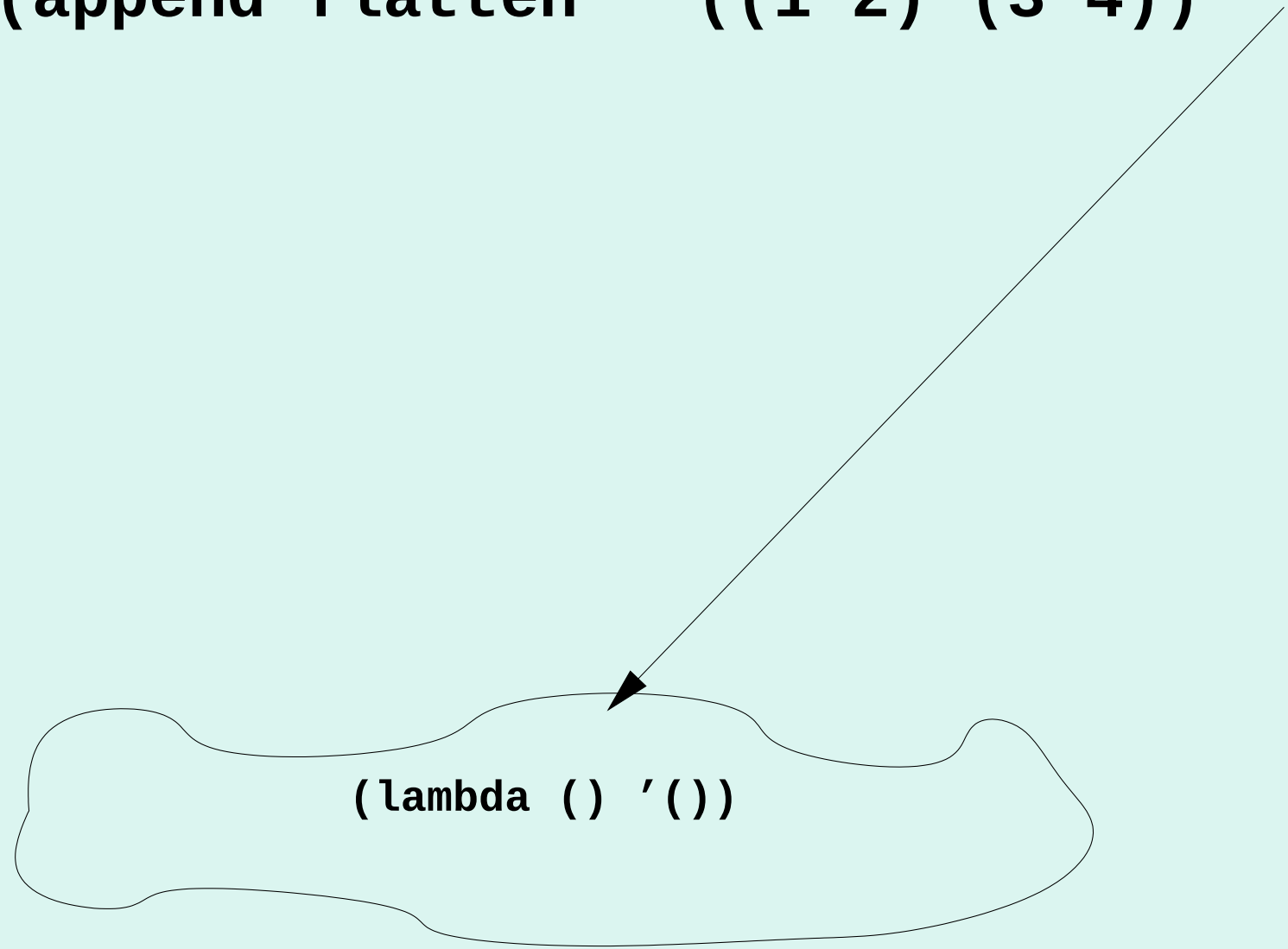
$listepar$ = ($objet$. $\rightarrow listepar$) | ()

```
(define (append-flatten t r)
  (cond ((null? t)
         (r))
        ((pair? t)
         (append-flatten
          (car t)
          (lambda ()
            (append-flatten (cdr t) r))))
        (else
         (cons t r))))

(define (flatten t)
  (append-flatten t (lambda () '())))
```

Continuations (9a)

- Trace de `(flatten '((1 2) (3 4)))`

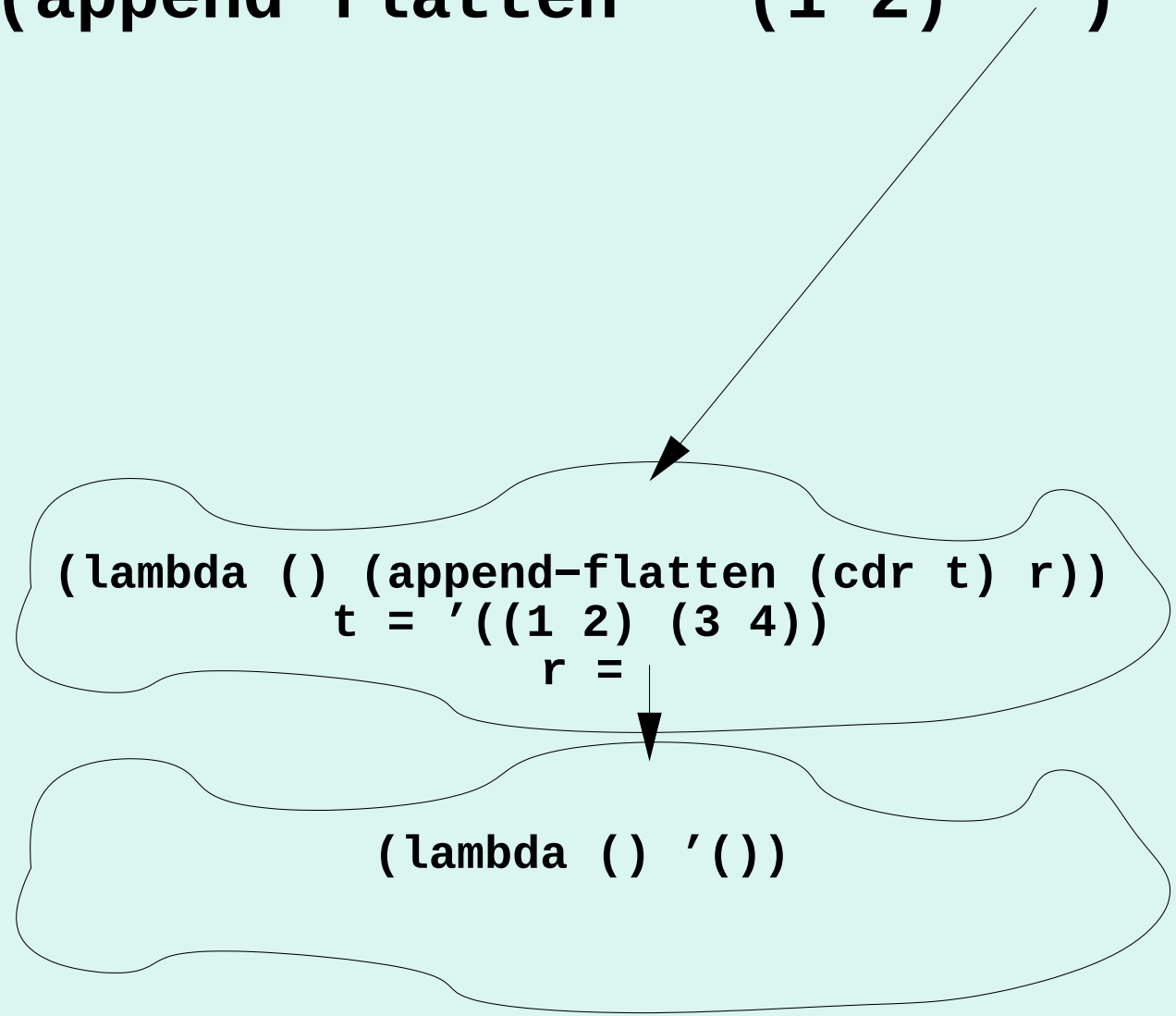
`(append-flatten  '((1 2) (3 4))  )`



Continuations (9b)

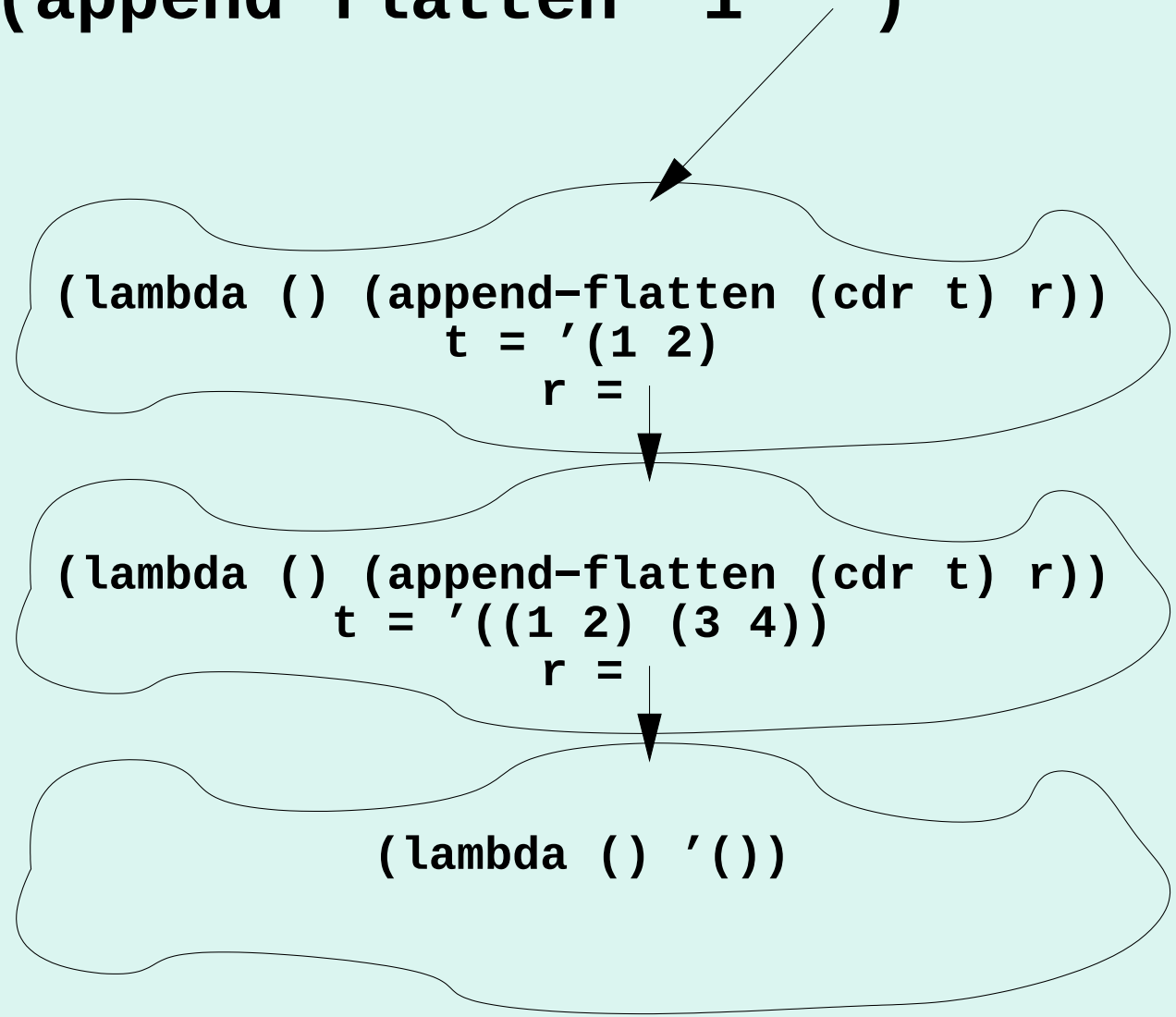
- Trace de `(flatten '((1 2) (3 4)))`

`(append-flatten '(1 2) )`



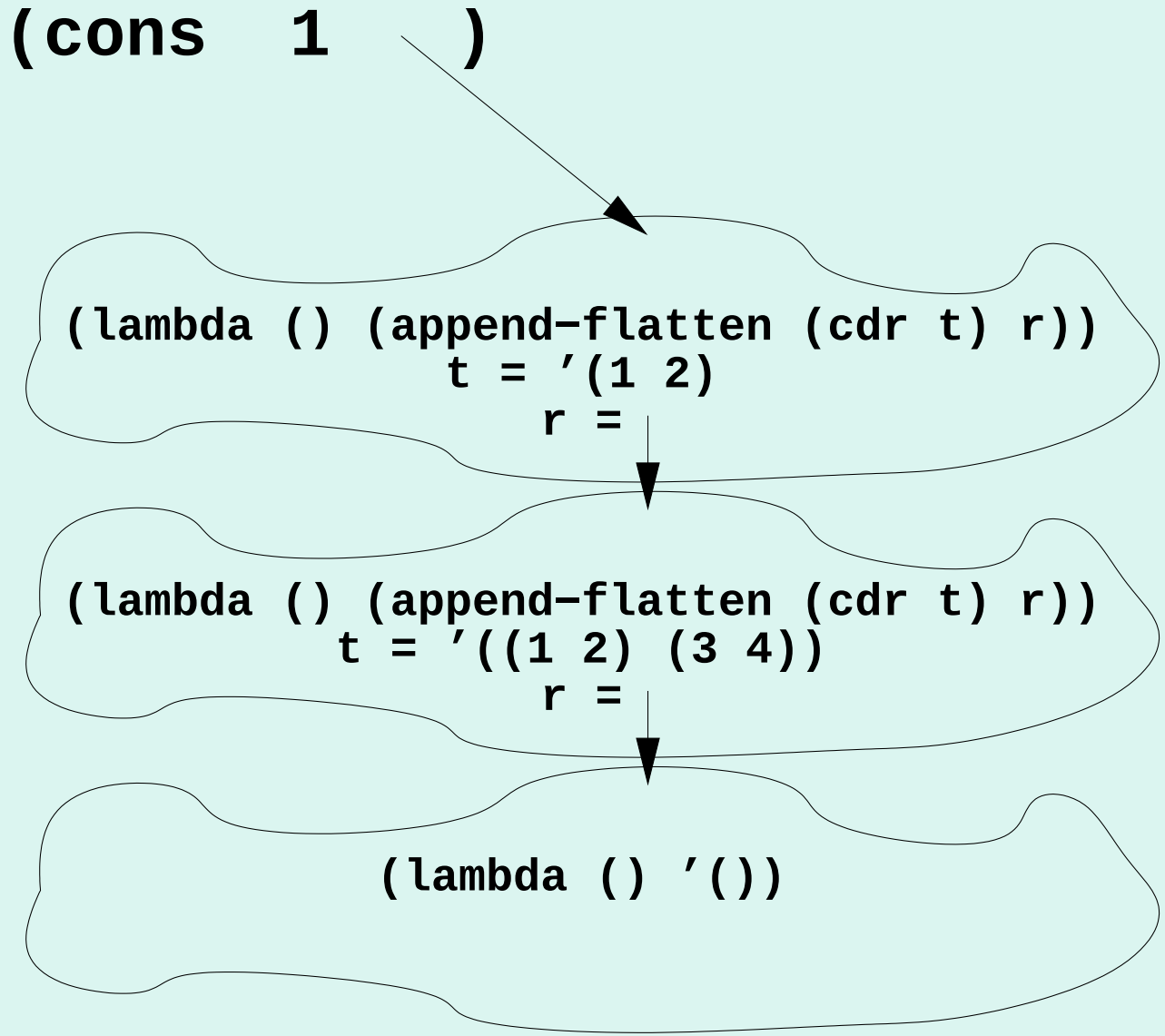
Continuations (9c)

- Trace de `(flatten '((1 2) (3 4)))`



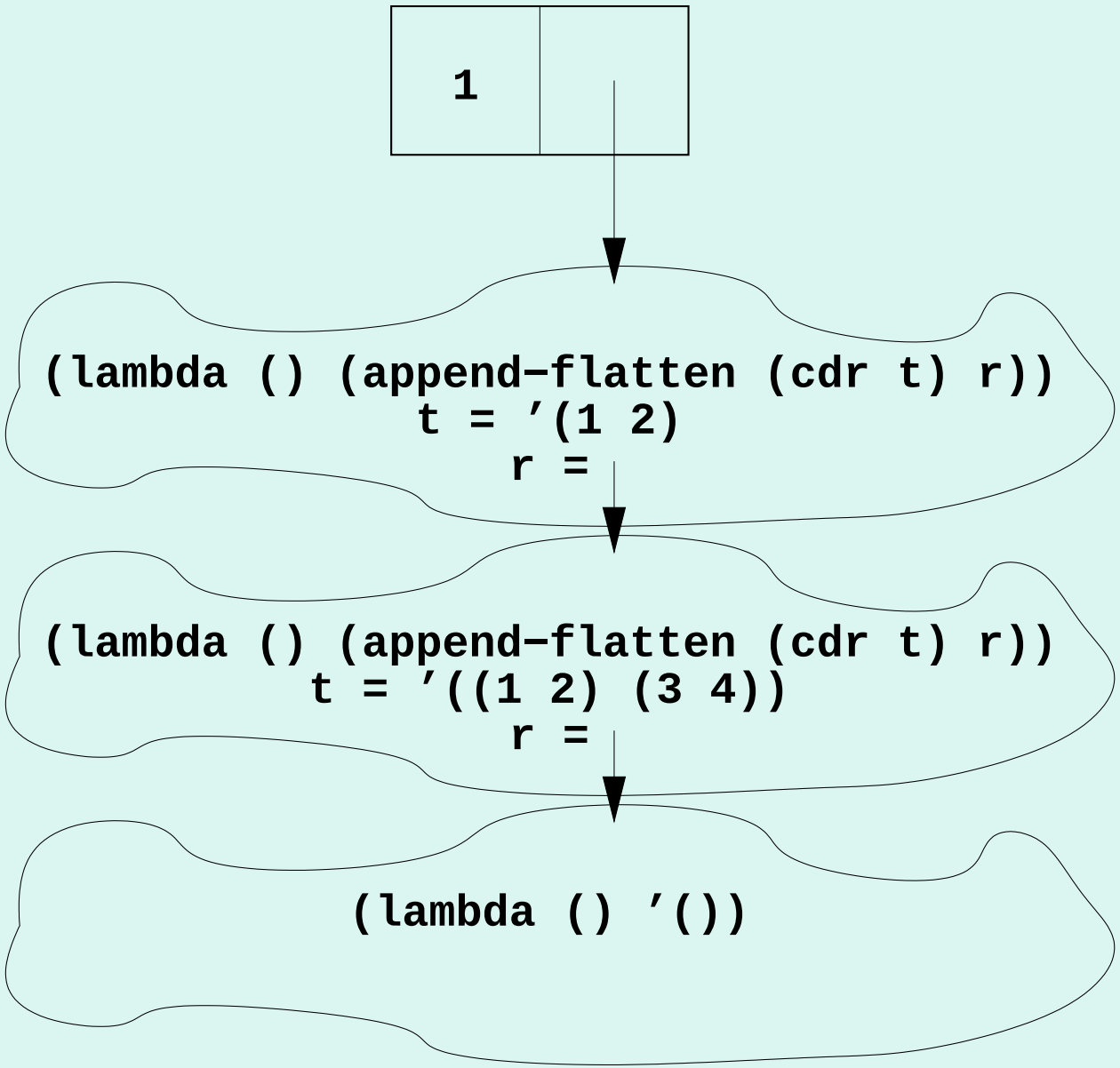
Continuations (9d)

- Trace de `(flatten '((1 2) (3 4)))`



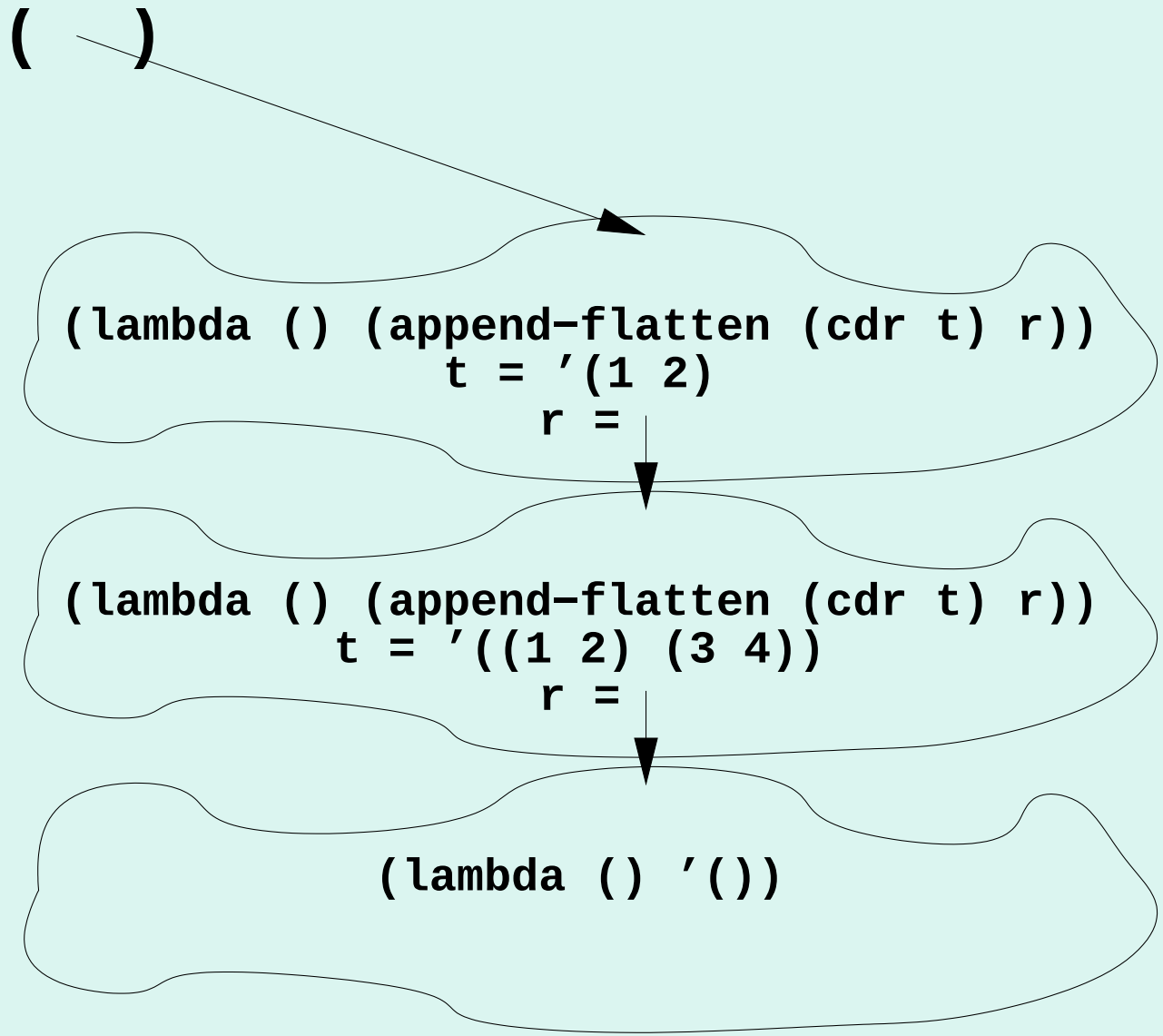
Continuations (9e)

- Trace de `(flatten '((1 2) (3 4)))`



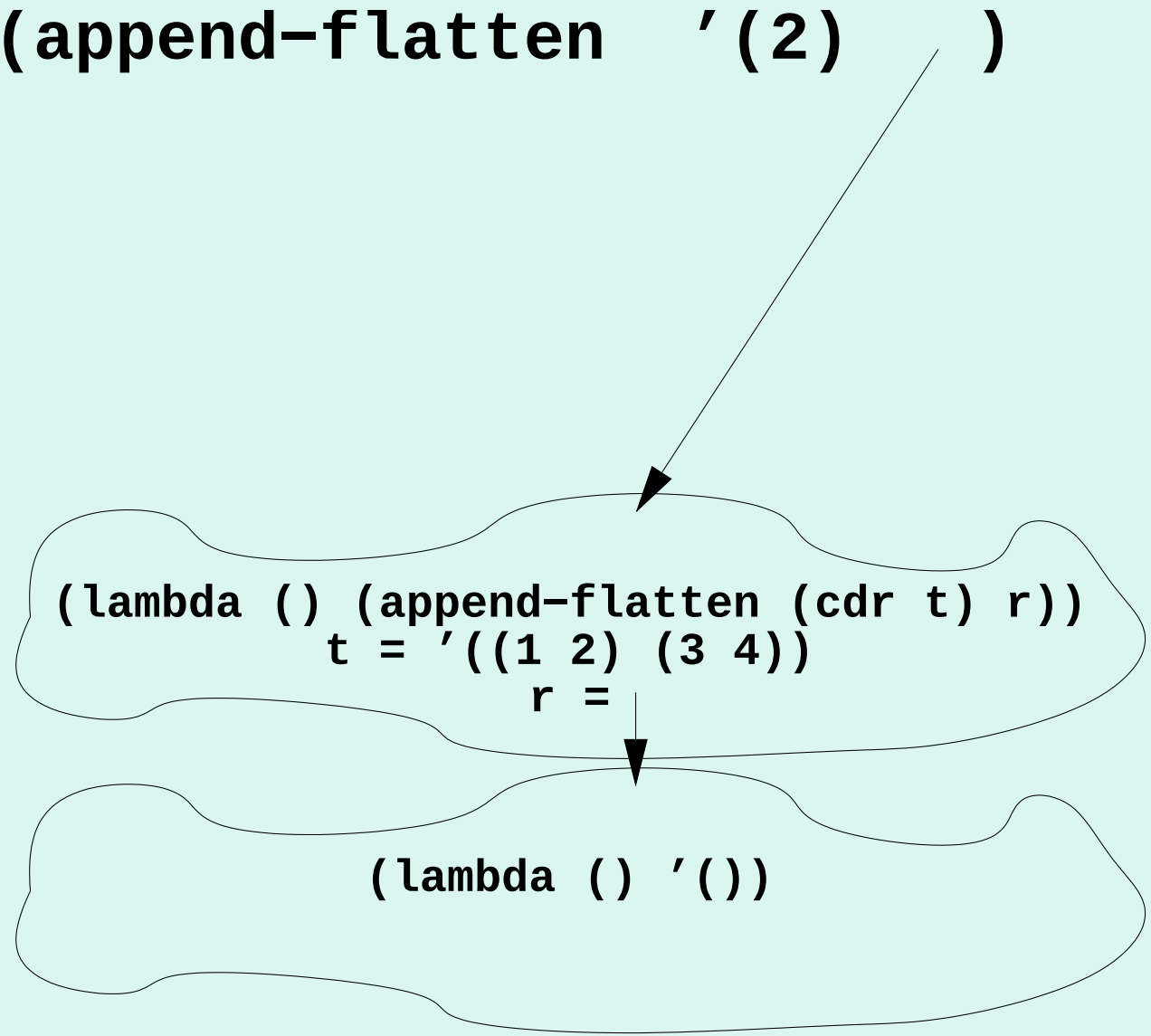
Continuations (9f)

- Trace de `((cdr (flatten '((1 2) (3 4)))))`



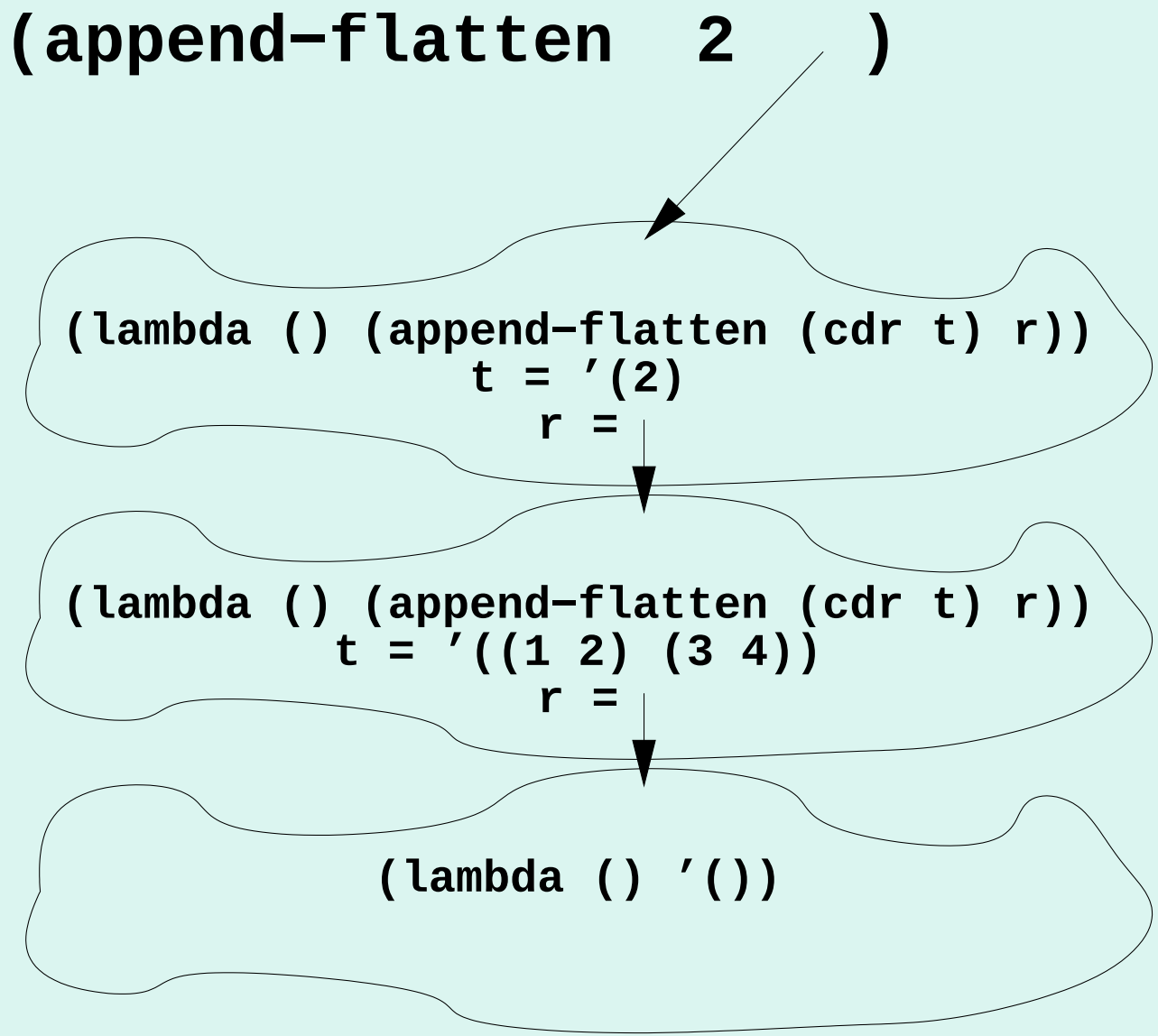
Continuations (9g)

- Trace de `((cdr (flatten '((1 2) (3 4)))))`



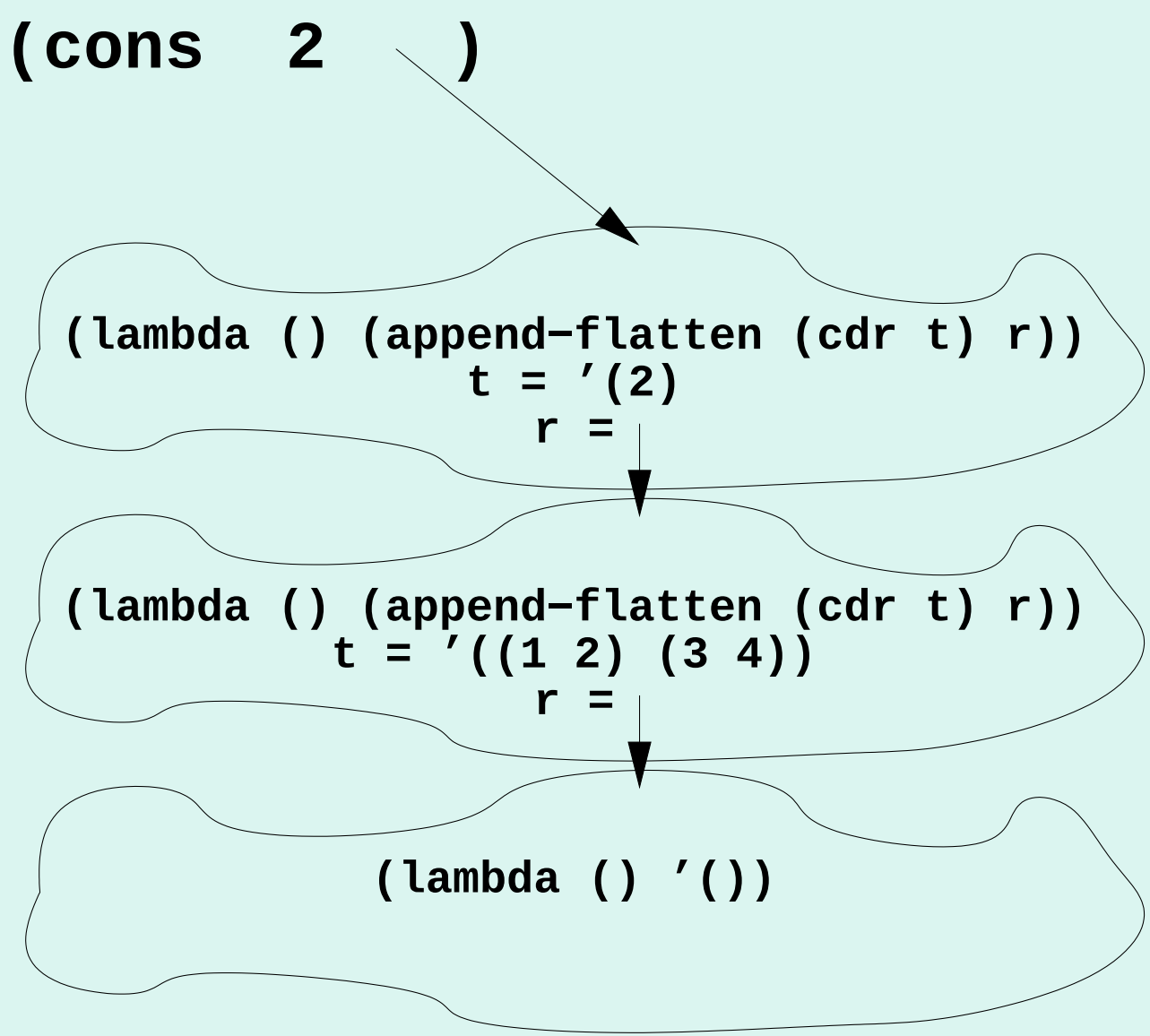
Continuations (9h)

- Trace de `((cdr (flatten '((1 2) (3 4)))))`



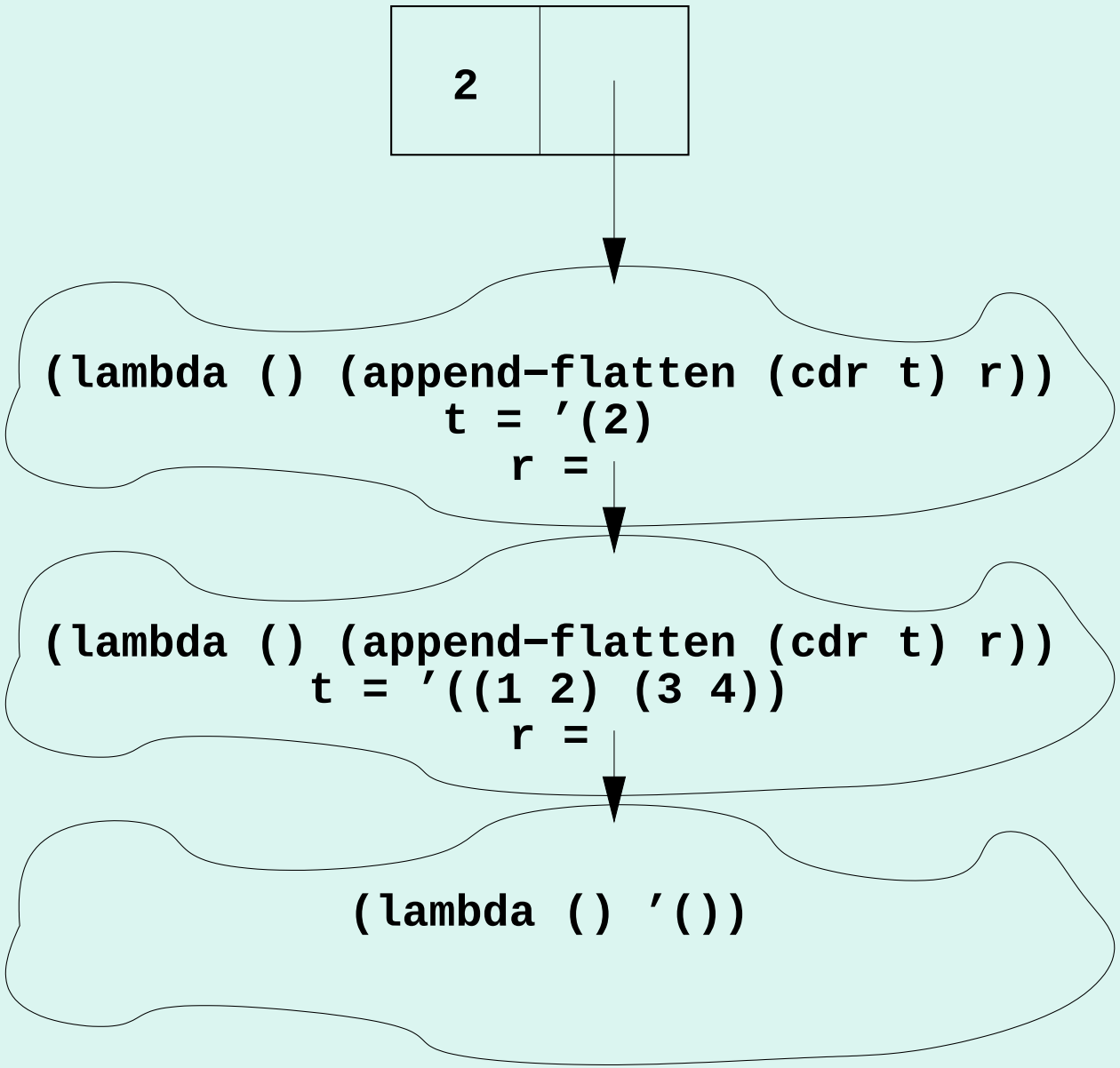
Continuations (9i)

- Trace de `((cdr (flatten '((1 2) (3 4)))))`



Continuations (9j)

- Trace de `((cdr (flatten '((1 2) (3 4)))))`



Continuations (10)


```
(define (enumerer x)
  (if (pair? x)
      (begin
        (display (car x))
        (enumerer ((cdr x))))))

(enumerer (flatten '(1 (2 (3 4)) (5))))
   => affiche 12345



(define (same-fringe? x y)
  (same-list? (flatten x)
              (flatten y)))

(define (same-list? x y)
  (if (null? x)
      (null? y)
      (and (not (null? y))
           (equal? (car x) (car y))
           (same-list? ((cdr x)) ((cdr y))))))
```

Continuations (11)

- Le paramètre `r` de `append-flatten` représente un **calcul suspendu** sous forme d'une fonction
- Pour continuer le calcul suspendu on appelle la fonction
- Pour cette raison on donne le nom **"continuation"** à cette fonction
- Une autre façon d'exprimer cet algorithme est d'utiliser des processus ou des coroutines, ce qui est équivalent à suspendre un calcul

Continuations (12)

- Ce concept de **suspension du calcul** se retrouve aussi dans la façon d'exécuter un programme
- En particulier l'exécution des **appels de fonction**
- Par exemple, dans l'expression

```
(* 2 (read))
```

la multiplication par 2 se fait **après** l'appel à la fonction `read`

Continuations (13)

- On peut donc voir la multiplication par 2 comme un **calcul suspendu** qui sera continué par la fonction `read` lorsque le résultat de cette fonction est connu

- Dans ce contexte, la continuation d'une fonction f est une fonction k à un paramètre, le **résultat de la fonction** f

- Si on écrit une fonction en explicitant la fonction qui dénote la continuation, on dit qu'elle est en **"Continuation Passing Style"** (CPS)

- Sinon, c'est le **style direct**

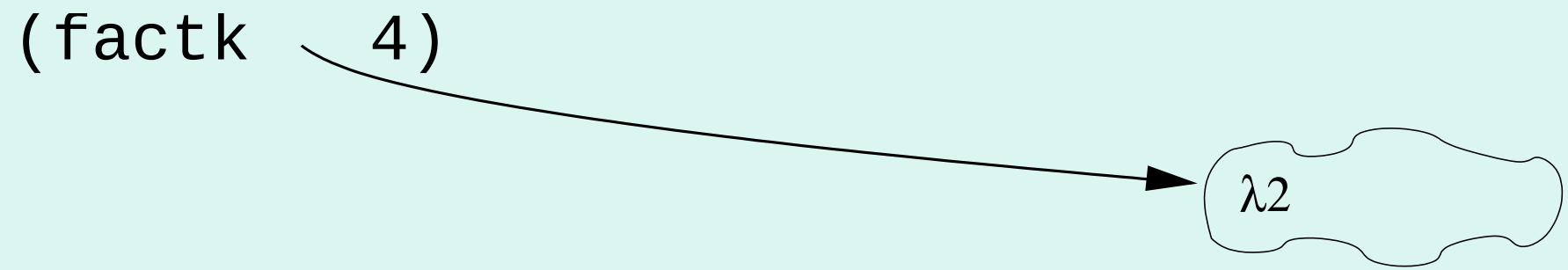
Continuations (14)


```
; en style direct :

(define (fact n)
  (if (<= n 1)
      1
      (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24



; en CPS :

(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda (r) (k (* n r)))
             (- n 1))))

(factk (lambda (r) (write r))
       4)
```

Continuations (15a)

- Trace de `(factk (lambda (r) (write r)) 4)`

```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```

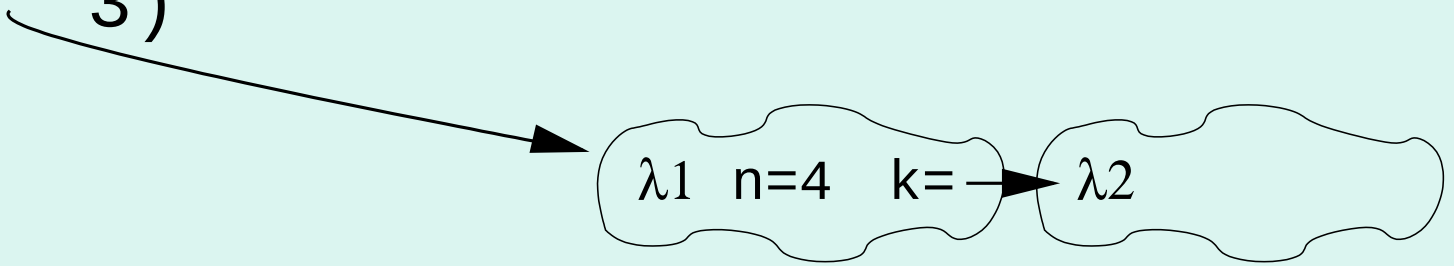


Continuations (15b)

- Trace de `(factk (lambda (r) (write r)) 4)`

```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```

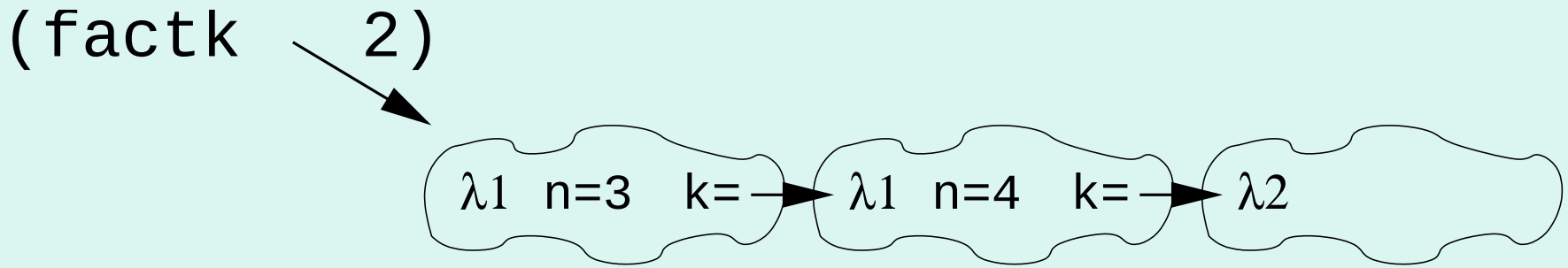
```
(factk   3)
```



Continuations (15c)

- Trace de `(factk (lambda (r) (write r)) 4)`

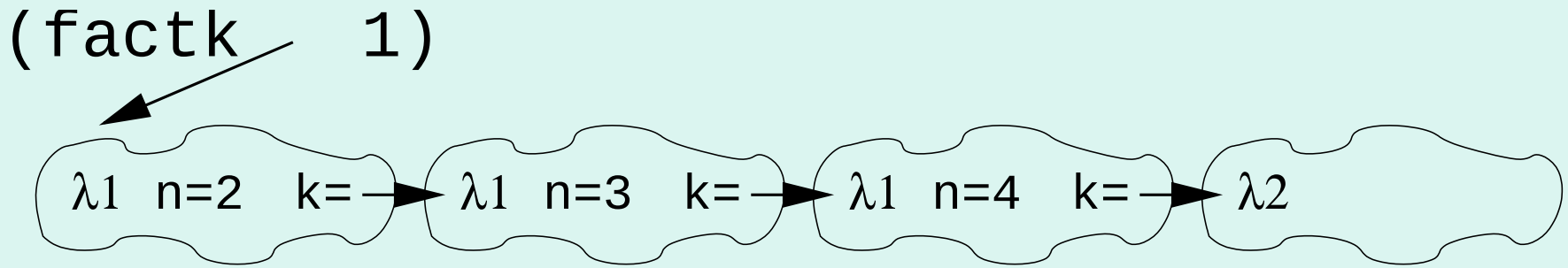
```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```



Continuations (15d)

- Trace de `(factk (lambda (r) (write r)) 4)`

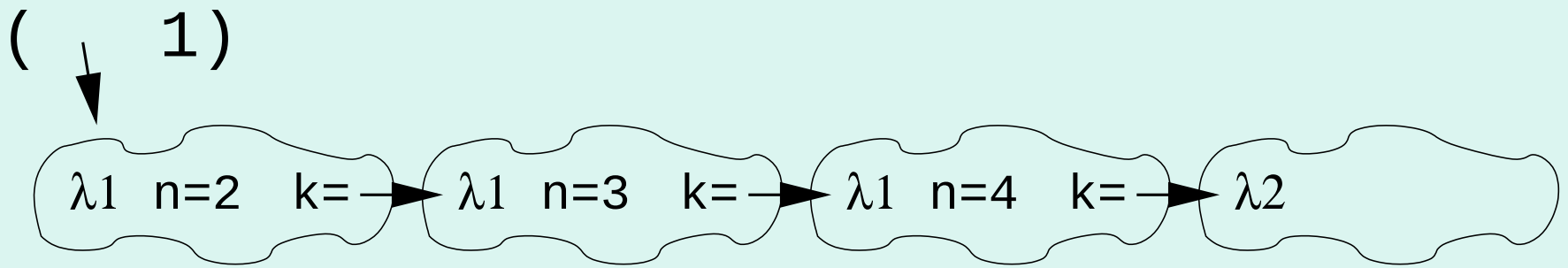
```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```



Continuations (15e)

- Trace de `(factk (lambda (r) (write r)) 4)`

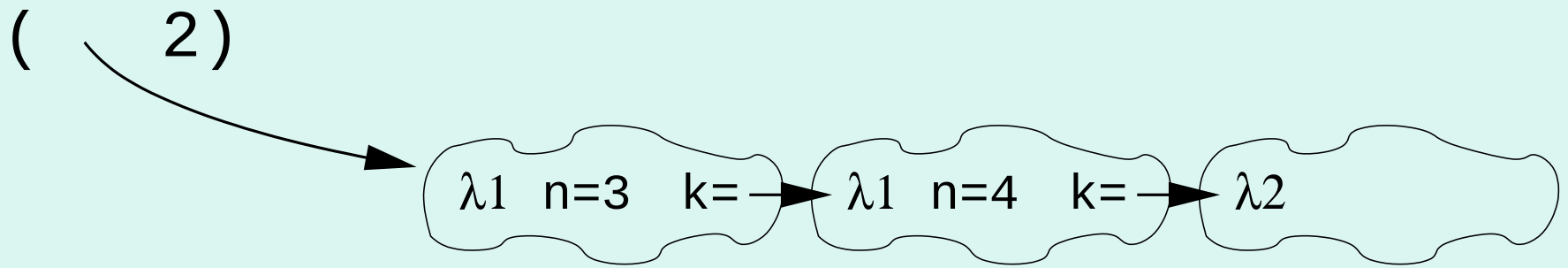
```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```



Continuations (15f)

- Trace de `(factk (lambda (r) (write r)) 4)`

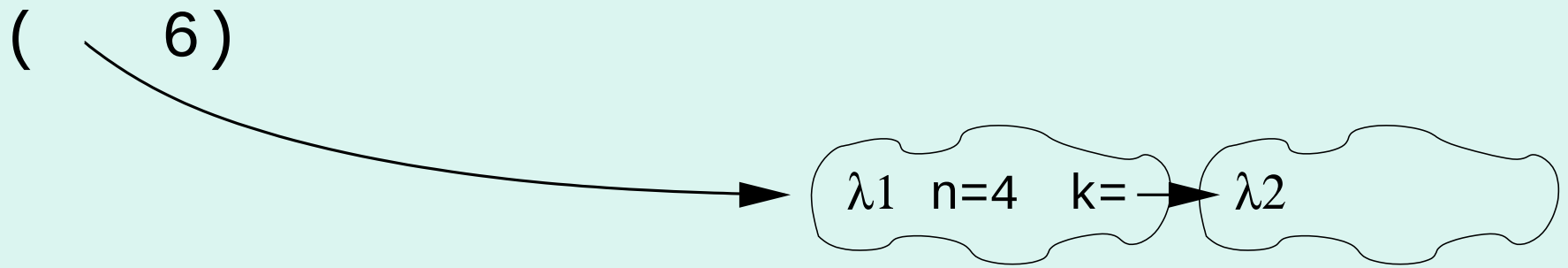
```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```



Continuations (15g)

- Trace de `(factk (lambda (r) (write r)) 4)`

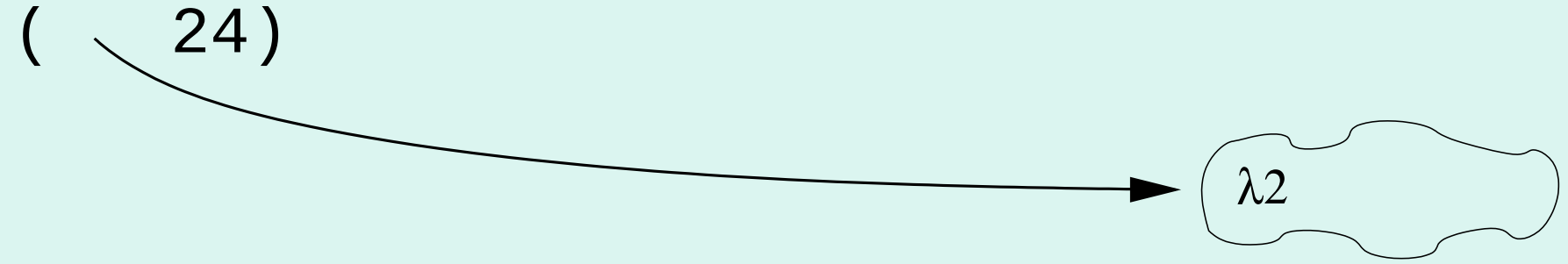
```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```



Continuations (15h)

- Trace de `(factk (lambda (r) (write r)) 4)`

```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```



Continuations (15i)

- Trace de `(factk (lambda (r) (write r)) 4)`

```
(define (factk k n)
  (if (<= n 1)
      (k 1)
      (factk (lambda1 (r) (k (* n r)))
             (- n 1))))

(factk (lambda2 (r) (write r))
       4)
```

```
(write 24)
```

Continuations (16)

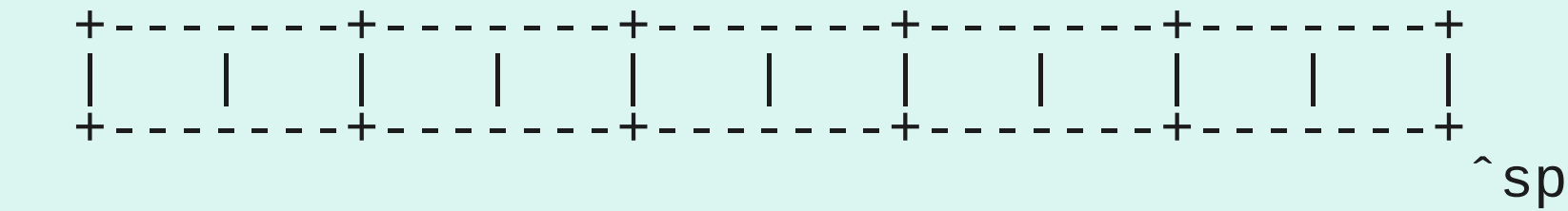
- Le déroulement du calcul en style CPS suit étroitement le déroulement du calcul en style direct
- Les fermetures qui représentent les continuations forment une pile semblable à la pile d'exécution
- Nous allons le démontrer en convertissant le programme en pseudocode machine

Continuations (17a)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```

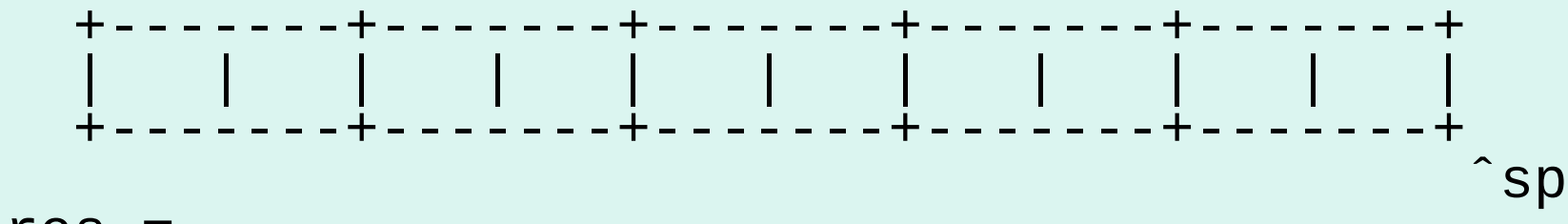


Continuations (17b)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```

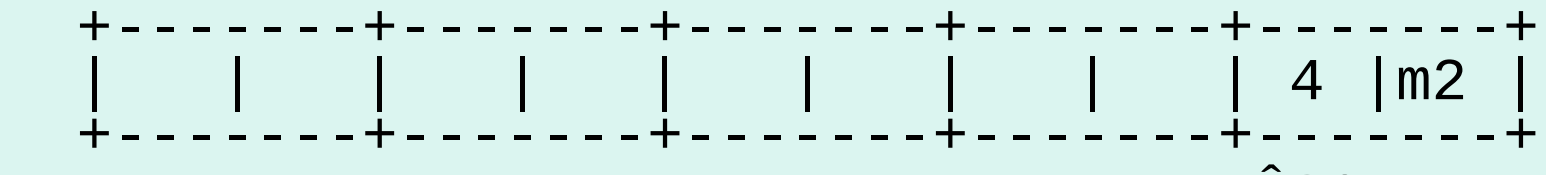


Continuations (17c)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```



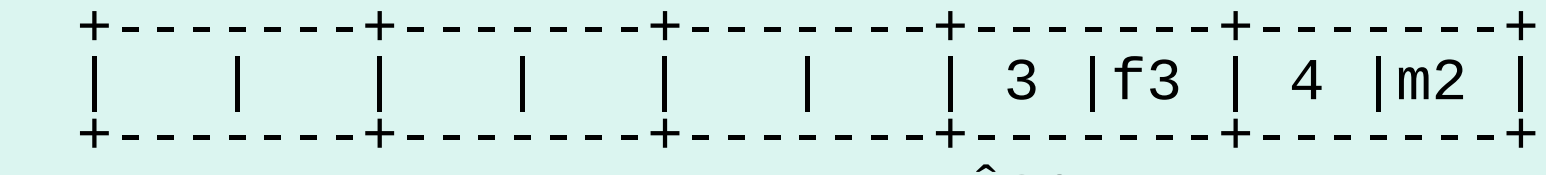
res =

Continuations (17d)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```

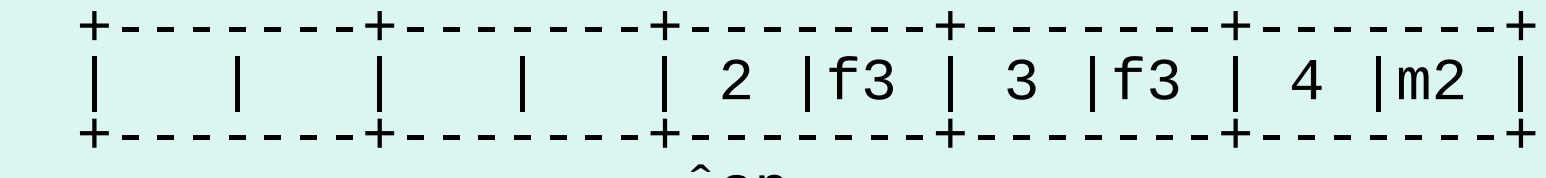


Continuations (17e)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```



Continuations (17f)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```

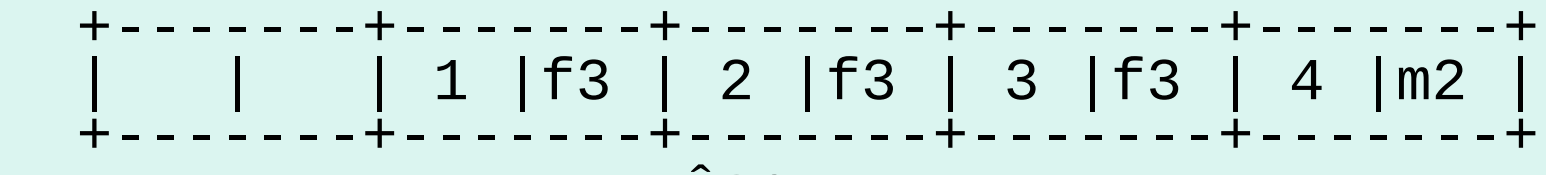
```
+-------+-------+-------+-------+-------+
|   |   | 1 |f3 | 2 |f3 | 3 |f3 | 4 |m2 |
+-------+-------+-------+-------+-------+
          ^sp
 res =
```

Continuations (17g)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```

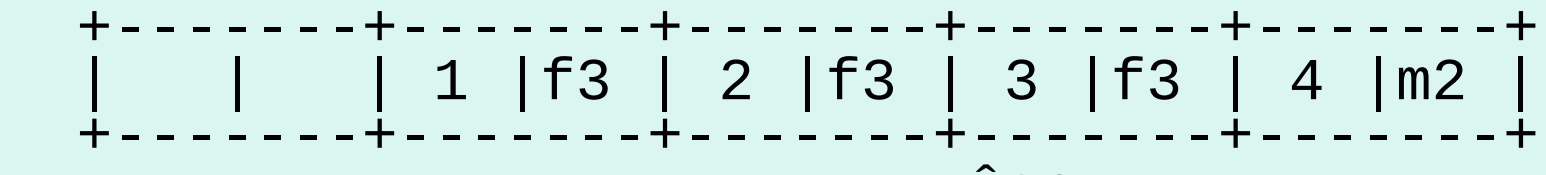


Continuations (17h)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```

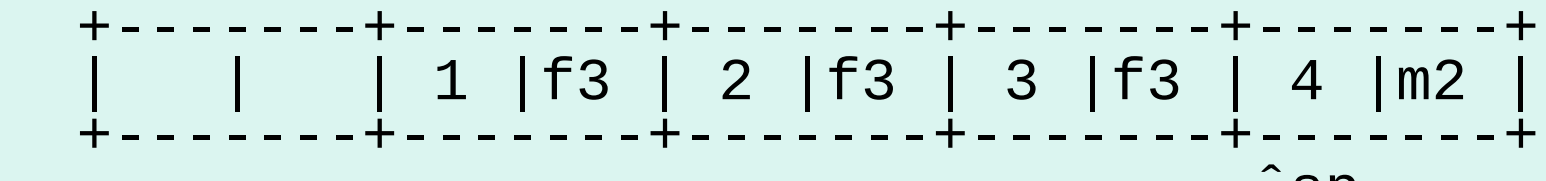


Continuations (17i)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```



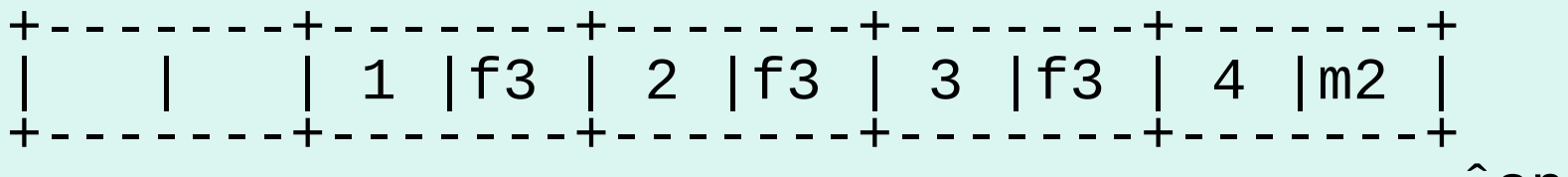
res = 6

Continuations (17j)


```
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))

(write (fact 4))  ; affiche 24
```

```
main:
    sp[-1] = m2
    sp[-2] = 4
    sp -= 2
    pc = fact
m2:
    sp[-1] = m3
    sp[-2] = res
    sp -= 2
    pc = write
m3:
```

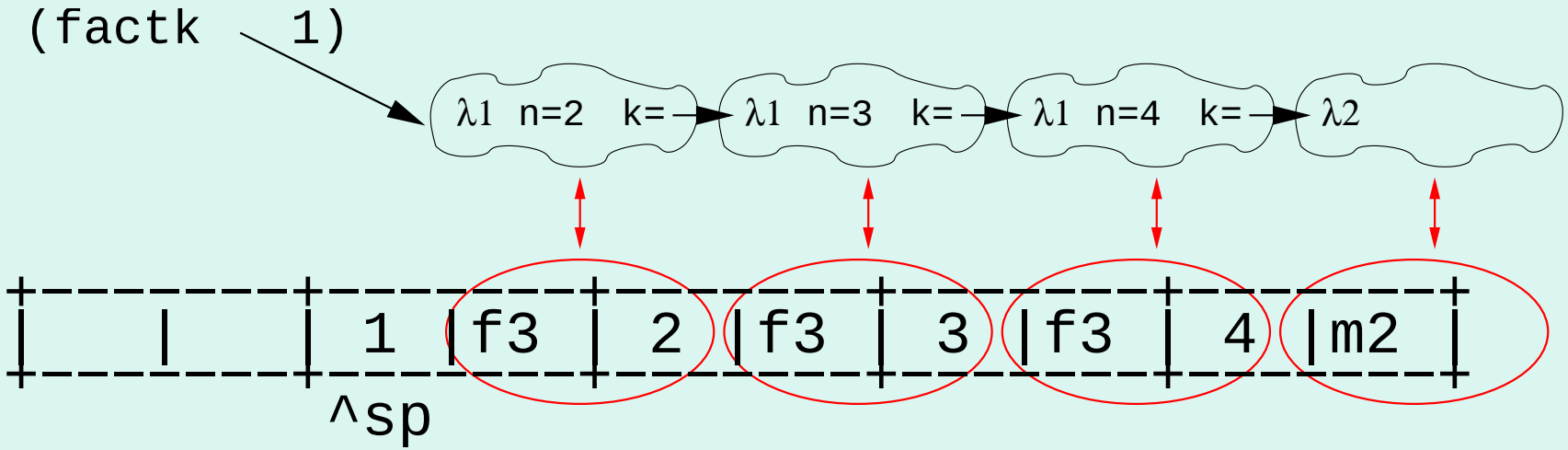
```
fact:
    if sp[0]>1 pc = f2
    res = 1
    sp += 2
    pc = sp[-1]
f2:
    sp[-1] = f3
    sp[-2] = sp[0] - 1
    sp -= 2
    pc = fact
f3:
    res = sp[0] * res
    sp += 2
    pc = sp[-1]
```




```
res = 24
```

Continuations (18)

- Les “frames” sur la pile d’exécution correspondent donc aux fermetures créées pour les continuations



Continuations (19)

- Une différence importante :
  - les continuations sont créées lors de l'appel d'une fonction pour permettre d'**exécuter le point de retour de l'appelant**
  - les frames sont créés lors de l'appel d'une fonction pour permettre d'**exécuter le corps de la fonction** (incluant les paramètres et les variables locales)

- Une autre différence c'est que les fermetures sont chaînées, tandis que les frames sont alloués de façon contigüe

Continuations (20)

- Les frames occupent moins d’espace
- Cependant il faut les désallouer dans l’ordre inverse de leur allocation (avec les fermetures chaînées le GC récupère les fermetures lorsqu’elles ne sont plus nécessaires)
- C’est utile pour les coroutines, le multithreading, le backtracking, etc

Appel terminal (1)

- Traduisons cette fonction en CPS :

```
(define (last lst) ; style direct
  (if (null? (cdr lst))
      (car lst)
      (last (cdr lst))))


(define (lastk k lst) ; CPS
  (if (null? (cdr lst))
      (k (car lst))
      (lastk (lambda (r) (k r))
             (cdr lst))))
```

Appel terminal (2)

- La version CPS fonctionne mais peut être améliorée
- En effet la continuation suivante est suspecte :

  `(lambda (r) (k r))`

  car elle est équivalente à `k`
- Inutile de créer une nouvelle fermeture

Appel terminal (3)

- Version améliorée de `last` :

```
(define (lastk k lst) ; CPS
  (if (null? (cdr lst))
      (k (car lst))
      (lastk k (cdr lst))))
```

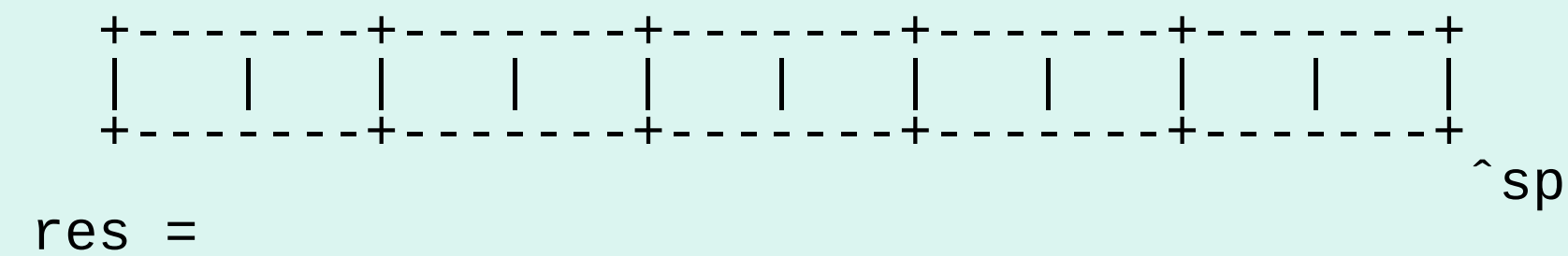
- La continuation **reste la même** d'un appel au suivant
- L'appel récursif à `lastk` est un **appel terminal** car c'est la "dernière opération" à faire dans la fonction qui contient l'appel
- C'est l'**optimisation de l'appel terminal** ("tail-call optimization")

Appel terminal (4a)


```
(define (last lst)
  (if (null? (cdr lst))
      (car lst)
      (last (cdr lst))))

(write (last '(1 2 3)))
```

```
main:
  sp[-1] = m2
  sp[-2] = '(1 2 3)
  sp -= 2
  pc = last
m2:
  sp[-1] = m3
  sp[-2] = res
  sp -= 2
  pc = write
m3:
```

```
last:
  tmp = cdr(sp[0])
  if !null?(tmp) pc = l2
  res = car(sp[0])
  sp += 2
  pc = sp[-1]
l2:
  sp[0] = cdr(sp[0])
  pc = last
```

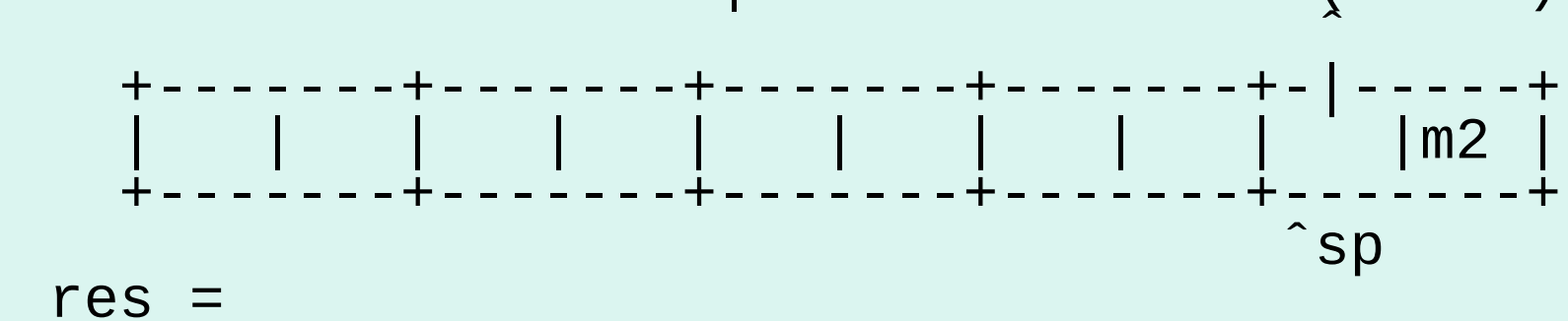


Appel terminal (4b)


```
(define (last lst)
  (if (null? (cdr lst))
      (car lst)
      (last (cdr lst))))

(write (last '(1 2 3)))
```

```
main:
  sp[-1] = m2
  sp[-2] = '(1 2 3)
  sp -= 2
  pc = last
m2:
  sp[-1] = m3
  sp[-2] = res
  sp -= 2
  pc = write
m3:
```

```
last:
  tmp = cdr(sp[0])
  if !null?(tmp) pc = l2
  res = car(sp[0])
  sp += 2
  pc = sp[-1]
l2:
  sp[0] = cdr(sp[0])
  pc = last
```

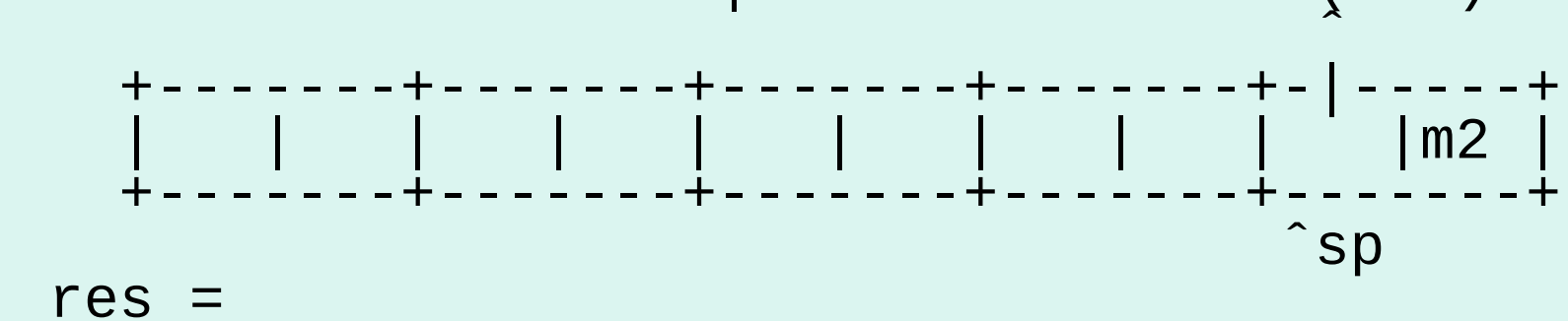


Appel terminal (4c)


```
(define (last lst)
  (if (null? (cdr lst))
      (car lst)
      (last (cdr lst))))

(write (last '(1 2 3)))
```

```
main:
  sp[-1] = m2
  sp[-2] = '(1 2 3)
  sp -= 2
  pc = last
m2:
  sp[-1] = m3
  sp[-2] = res
  sp -= 2
  pc = write
m3:
```

```
last:
  tmp = cdr(sp[0])
  if !null?(tmp) pc = l2
  res = car(sp[0])
  sp += 2
  pc = sp[-1]
l2:
  sp[0] = cdr(sp[0])
  pc = last
```

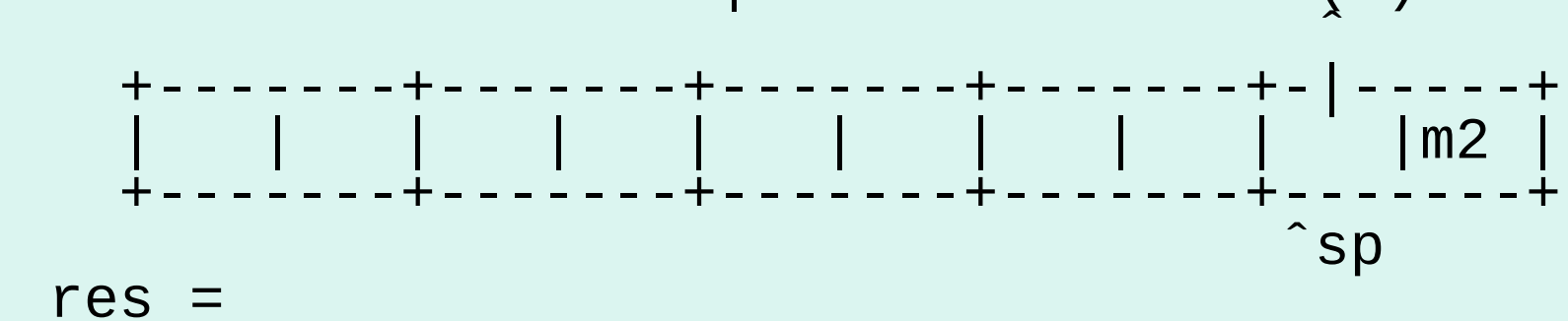


Appel terminal (4d)


```
(define (last lst)
  (if (null? (cdr lst))
      (car lst)
      (last (cdr lst))))

(write (last '(1 2 3)))
```

```
main:
  sp[-1] = m2
  sp[-2] = '(1 2 3)
  sp -= 2
  pc = last
m2:
  sp[-1] = m3
  sp[-2] = res
  sp -= 2
  pc = write
m3:
```

```
last:
  tmp = cdr(sp[0])
  if !null?(tmp) pc = l2
  res = car(sp[0])
  sp += 2
  pc = sp[-1]
l2:
  sp[0] = cdr(sp[0])
  pc = last
```

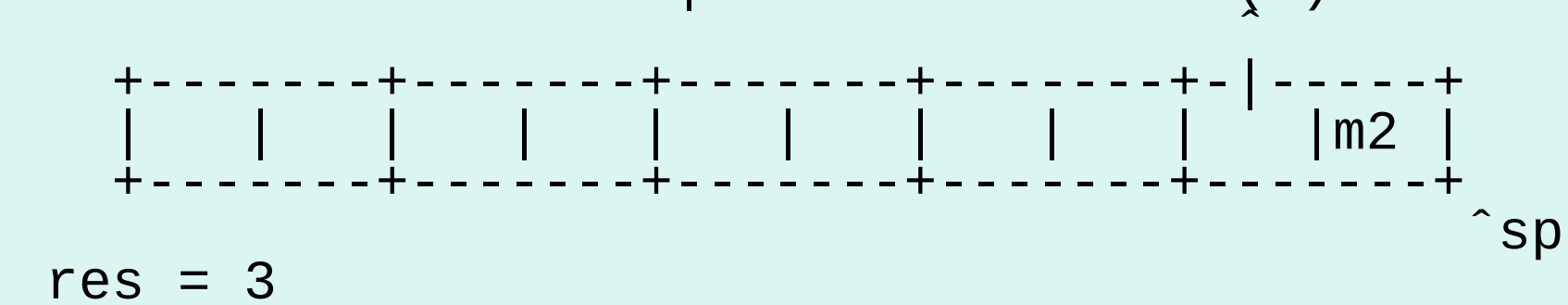


Appel terminal (4e)


```
(define (last lst)
  (if (null? (cdr lst))
      (car lst)
      (last (cdr lst))))

(write (last '(1 2 3)))
```

```
main:
  sp[-1] = m2
  sp[-2] = '(1 2 3)
  sp -= 2
  pc = last
m2:
  sp[-1] = m3
  sp[-2] = res
  sp -= 2
  pc = write
m3:
```

```
last:
  tmp = cdr(sp[0])
  if !null?(tmp) pc = l2
  res = car(sp[0])
  sp += 2
  pc = sp[-1]
l2:
  sp[0] = cdr(sp[0])
  pc = last
```



Appel terminal (5)

- Appel terminal = **“goto” avec paramètre**
- Aucun espace supplémentaire (à long terme) utilisé sur la pile
- Une récursion qui se fait uniquement au moyen d’appels terminaux correspond donc à une **boucle**

Appel terminal (6)

- Traduisons `append-flatten` en CPS :

```
(define (append-flatten t r) ; direct
  (cond ((null? t) r)
        ((pair? t)
         (append-flatten
          (car t)
          (append-flatten (cdr t) r)))
        (else (cons t r))))

(define (append-flattenk k t r) ; CPS
  (cond ((null? t) (k r))
        ((pair? t)
         (append-flattenk
          (lambda (r1)
            (append-flattenk
             (lambda (r2) (k r2))
             (car t)
             r1))
          (cdr t)
          r))
        (else (k (cons t r)))))
```

Appel terminal (7)

- Version optimisée de `append-flattenk` :

```
(define (append-flattenk k t r)
  (cond ((null? t) (k r))
        ((pair? t)
         (append-flattenk
          (lambda (r1)
            (append-flattenk k (car t) r1))
          (cdr t)
          r))
        (else (k (cons t r)))))
```

- On remarque qu'un seul des deux appels récursifs est un appel terminal

- Comment faire pour reconnaitre les appels terminaux?

Appel terminal (8)

- Une expression E est en **position terminale** par rapport à une fonction f qui la contient, ssi la valeur de E est retournée par f et après l'évaluation de E il n'y a pas d'autre opération à faire dans f
- Exemple :

```
(define f
  (lambda (n)
    (* n n)))

(define g
  (lambda (n)
    (let ((x (* n n)))
      x)))
```

- L'expression `(* n n)` est en position terminale dans `f` mais pas dans g

Appel terminal (9)

- L'**analyse de position terminale** consiste à étiquetter les expressions :

$$E^x \qquad x = T \text{ ou } x = N$$

- Règles fondamentales :
  - (`lambda` P E^T)^x
  - (E_0^N E_1^N `...`)^x
  - (`set!` V E_1^N)^x
  - (`if` E_1^N E_2^x E_3^x)^x
  - (`let` ((V E_1^N)`...`) E_0^x)^x

- Appel terminal = appel en position terminale

Appel terminal (10)

- Donc dans :

```
(define append-flatten
  (lambda (t r)
    (cond ((null? t)
           r)
          ((pair? t)
           (append-flatten
            (car t)
            (append-flatten (cdr t) r)))
          (else
           (cons t r)))))
```

seul l'appel `(append-flatten (car t) ...)` est en position terminale (techniquement `(cons t r)` l'est aussi mais nous traiterons `cons`, `+`, etc comme des opérations primitives)

Appel terminal (11)

- Analyse d’appel terminal:

```
> (app-termi
    '(lambda (x)
       (if (f x)
           (g x)
           (let ((y (h x))) (h y))))
    'N)
((g x) (h y))
```

Appel terminal (12)


```
(define (app-termi expr pos)
  (match expr
    ((lambda ,P ,E1)
     (app-termi E1 'T))
    ((set! ,V ,E1)
     (app-termi E1 'N))
    ((if ,E1 ,E2 ,E3)
     (append (app-termi E1 'N)
             (app-termi E2 pos)
             (app-termi E3 pos)))
    ((let ,B ,E0)
     (append (apply append
                    (map (lambda (x) (app-termi x 'N))
                         (map cadr B)))
             (app-termi E0 pos)))
    ((,op . ,Es)
     (let ((t (apply append
                     (map (lambda (x) (app-termi x 'N))
                          (cons op Es)))))
       (if (eq? pos 'T)
           (cons expr t)
           t)))
    (,_
     '())))
```

Conversion CPS (1)

- Est-ce possible de convertir automatiquement du style direct au “Continuation Passing Style”?
- Oui! **“Conversion CPS”**
- La conversion CPS est une **transformation source-à-source**

Conversion CPS (2)

- Dans ce programme :

```
(let ((carre (lambda (x) (* x x))))
  (write (+ (carre 10) 1)))
```

  la continuation de `(carre 10)` est un calcul qui attends une valeur à laquelle 1 sera ajouté et le résultat imprimé

- Cette continuation est représentée par la fonction

```
(lambda (r) (write (+ r 1)))
```

Conversion CPS (3)

- Cette continuation doit être passée à `carre` pour qu'elle puisse y envoyer son résultat

- Donc il faut ajouter un paramètre "continuation" à toutes les `lambda`-expressions, changer les appels pour passer la fonction de continuation, et utiliser cette continuation lorsque la fonction doit retourner un résultat

```
(let ((carre (lambda (k x) (k (* x x)))))
  (carre (lambda (r) (write (+ r 1)))
         10))
```

Conversion CPS (4)

- Lorsque la conversion CPS est effectuée systématiquement sur tout le programme
  - Tous les appels de fonction sont des **appels terminaux**
  - Chaque appel non-terminal **crée une fermeture** pour la continuation de l'appel
- Dans une première version de la conversion CPS, nous allons prendre la position que **chaque sous-expression a une continuation**

```
(+ 5 x)
```

la continuation de 5 c'est une fonction qui reçoit la valeur 5 en paramètre

Conversion CPS : règles (1)

- Nous utiliserons la notation

$$\boxed{E}_{\mathcal{K}}$$

  pour dénoter l'expression Scheme qui est la conversion en CPS de l'expression E où l'expression Scheme $\mathcal{K}$ représente la continuation de E

- L'expression E peut contenir des appels non-terminaux
- $\mathcal{K}$ est une expression en CPS (contenant aucun appel non-terminal) dont la valeur est une fonction à 1 paramètre
- $\mathcal{K}$ est une variable ou une `lambda`-expression

Conversion CPS : règles (2)

- Première règle :

$$\boxed{\textit{programme}} = \boxed{\textit{programme}}\ \texttt{(lambda (}r\texttt{) (halt }r\texttt{))}$$

- Cette règle indique que la **continuation primordiale** d'un programme prends r, le résultat du programme, et appelle l'opération primitive `halt` qui termine l'exécution du programme

Conversion CPS : règles (3)

- $$\boxed{c}_{\mathcal{K}} = \texttt{(}\mathcal{K}\ \ c\texttt{)}$$

- $$\boxed{v}_{\mathcal{K}} = \texttt{(}\mathcal{K}\ \ v\texttt{)}$$

- $$\boxed{\texttt{(set! } v\ \ E_1\texttt{)}}_{\mathcal{K}} = \boxed{E_1}_{\begin{array}{l}\texttt{(lambda (}r_1\texttt{)}\\ \quad\texttt{(}\mathcal{K}\ \texttt{(set! } v\ \ r_1\texttt{)))}\end{array}}$$

- $$\boxed{\texttt{(if } E_1\ \ E_2\ \ E_3\texttt{)}}_{\mathcal{K}} = \boxed{E_1}_{\begin{array}{l}\texttt{(lambda (}r_1\texttt{)}\\ \quad\texttt{(if } r_1\ \ \boxed{E_2}_{\mathcal{K}}\ \ \boxed{E_3}_{\mathcal{K}}\texttt{))}\end{array}}$$

Conversion CPS : règles (4)

- $\underset{\mathcal{K}}{\boxed{\texttt{(begin } E_1 \ E_2\texttt{)}}} = \boxed{E_1}$ `(lambda (`r_1`)` $\underset{\mathcal{K}}{\boxed{E_2}}$ `)`

- $\underset{\mathcal{K}}{\boxed{\texttt{(+ } E_1 \ E_2\texttt{)}}} =$ (pour toute opération primitive)

  $\boxed{E_1}$

  `(lambda (`r_1`)` $\boxed{E_2}$ `)`

  `(lambda (`r_2`) (`$\mathcal{K}$ `(+` r_1 r_2`)))`

- $\underset{\mathcal{K}}{\boxed{\texttt{(lambda (}P_1 \ldots P_n\texttt{) } E_0\texttt{)}}} =$

  `(`$\mathcal{K}$ `(lambda (`k $P_1 \ldots P_n$`)` $\underset{k}{\boxed{E_0}}$ `))`

Conversion CPS : règles (5)

- $\underset{\mathcal{K}}{\boxed{(E_0)}} =$ $\boxed{E_0}$ `(lambda (`r_0`) (`r_0 $\mathcal{K}$`))`

- $\underset{\mathcal{K}}{\boxed{(E_0\ E_1)}} =$ $\boxed{E_0}$ `(lambda (`r_0`)` $\boxed{E_1}$ `(lambda (`r_1`) (`r_0 $\mathcal{K}$ r_1`)))`

- $\underset{\mathcal{K}}{\boxed{(E_0\ E_1\ E_2)}} =$

$\boxed{E_0}$ `(lambda (`r_0`)` $\boxed{E_1}$ `(lambda (`r_1`)` $\boxed{E_2}$ `(lambda (`r_2`) (`r_0 $\mathcal{K}$ r_1 r_2`)))`

- etc.

- Exemple 1 :

(+ 5 x) =
(lambda (r) (halt r))

5 =
(lambda (r_1) x)
(lambda (r_2) ((lambda (r) (halt r))
(+ r_1 r_2)))

5 =
(lambda (r_1) ((lambda (r_2) ((lambda (r) (halt r))
(+ r_1 r_2)))
x))

((lambda (r_1) ((lambda (r_2) ((lambda (r) (halt r))
(+ r_1 r_2)))
x))
5)

Conversion CPS : règles (7)

- Exemple 2 : $\boxed{\texttt{((lambda (x) (+ 5 x)) 9)}}$ =

$$\texttt{(lambda (r) (halt r))}$$

$$\boxed{\texttt{(lambda (x) (+ 5 x))}} \quad =$$

$$\texttt{(lambda (}r_0\texttt{)} \quad \boxed{\texttt{9}} \quad \texttt{)}$$

$$\texttt{(lambda (}r_1\texttt{) (}r_0\texttt{ (lambda (r) (halt r)) }r_1\texttt{))}$$

$$\boxed{\texttt{(lambda (x) (+ 5 x))}} \quad =$$

$$\texttt{(lambda (}r_0\texttt{) ((lambda (}r_1\texttt{) (}r_0\texttt{ (lambda (r) (halt r)) }r_1\texttt{))}$$

$$\texttt{9))}$$

$$\texttt{((lambda (}r_0\texttt{) ((lambda (}r_1\texttt{) (}r_0\texttt{ (lambda (r) (halt r)) }r_1\texttt{))}$$

$$\texttt{9))}$$

$$\texttt{(lambda (}k\texttt{ x) }\boxed{\texttt{(+ 5 x)}}\texttt{))}$$

$$k$$

=

Conversion CPS : règles (8)


```
((lambda (r0)
   ((lambda (r1)
      (r0 (lambda (r) (halt r)) r1))
    9))
 (lambda (k x)
   ((lambda (r2)
      ((lambda (r3)
         (k (+ r2 r3)))
       x))
    5)))
```

Conversion CPS : implantation (1)

- L'algorithme de conversion CPS s'exprime comme une fonction `cps` à 2 paramètres

$$\boxed{\frac{E}{\mathcal{K}}} \texttt{ = (cps 'E '}\mathcal{K}\texttt{)}$$

$$\texttt{cps} :: exp\ \ contcps \rightarrow\ \ expcps$$

exp = expression Scheme
$expcps$ = exp contenant aucun appel non-terminal
$contcps$ = $expcps$ dénotant une fonction à 1 paramètre

Conversion CPS : implantation (2)

- Exemple :

```
(cps '(+ 5 x)
     '(lambda (r) (halt r))) =>

((lambda (#:g1)
   ((lambda (#:g2)
      ((lambda (r) (halt r))
       (+ #:g1 #:g2)))
    x))
 5)
```

Conversion CPS : implantation (3)


```
(define (cps E K)
  (match E

    (,const when (constant? const)
     `(,K ,const))

    (,var when (variable? var)
     `(,K ,var))

    ((set! ,var ,E1)
     (let ((r1 (gensym)))
       (cps E1 `(lambda (,r1) (,K (set! ,var ,r1))))))

    ((if ,E1 ,E2 ,E3)
     (let ((r1 (gensym)))
       (cps E1 `(lambda (,r1)
                  (if ,r1 ,(cps E2 K) ,(cps E3 K))))))

    ((begin ,E1 ,E2)
     (let ((r1 (gensym)))
       (cps E1 `(lambda (,r1) ,(cps E2 K)))))

    ((,op ,E1 ,E2) when (memq op '(+ - * / < <= =))
     (let ((r1 (gensym)) (r2 (gensym)))
       (cps E1 `(lambda (,r1)
                  ,(cps E2 `(lambda (,r2)
                              (,K (,op ,r1 ,r2)))))))))
```

Conversion CPS : implantation (4)


```
    ((lambda ,params ,E0)
     (let ((k (gensym)))
       `(,K (lambda (,k ,@params) ,(cps E0 k)))))

    ((,E0)
     (let ((r0 (gensym)))
       (cps E0 `(lambda (,r0) (,r0 ,K)))))

    ((,E0 ,E1)
     (let ((r0 (gensym)) (r1 (gensym)))
       (cps E0 `(lambda (,r0)
                  ,(cps E1 `(lambda (,r1)
                              (,r0 ,K ,r1))))))))

    ((,E0 ,E1 ,E2)
     (let ((r0 (gensym)) (r1 (gensym)) (r2 (gensym)))
       (cps E0 `(lambda (,r0)
                  ,(cps E1 `(lambda (,r1)
                              ,(cps E2 `(lambda (,r2)
                                          (,r0 ,K ,r1 ,r2)))))))))
    (,_
     (error "unknown expression"))))

(define (constant? c)
  (match c
    ((quote ,x)
     #t)
    (,x
     (or (number? x) (string? x) (boolean? x) (char? x)))))

(define variable? symbol?)
```

Conversion CPS : exemple (1)

- Exemple :

```
> (cps '(lambda (x y) (f x (g y))) 'halt)
(halt (lambda (#:g0 x y)
        ((lambda (#:g1)
           ((lambda (#:g2)
              ((lambda (#:g4)
                 ((lambda (#:g5)
                    (#:g4 (lambda (#:g3)
                            (#:g1 #:g0 #:g2 #:g3))
                          #:g5))
                  y))
               g))
            x))
         f)))
```

- Reconstruction des `lets` :

```
(halt (lambda (#:g0 x y)
        (let* ((#:g1 f) (#:g2 x) (#:g4 g) (#:g5 y))
          (#:g4 (lambda (#:g3)
                  (#:g1 #:g0 #:g2 #:g3))
                #:g5)))))
```

Conversion CPS : exemple (2)

- Exemple avec reconstruction des `lets` :

```
> (cps '(set! fact
             (lambda (n)
               (if (<= n 1)
                   1
                   (* n (fact (- n 1))))))
       'halt)
(let ((#:g1
       (lambda (#:g2 n)
         (let* ((#:g10 n)
                (#:g11 1)
                (#:g3 (<= #:g10 #:g11)))
           (if #:g3
               (#:g2 1)
               (let* ((#:g4 n)
                      (#:g6 fact)
                      (#:g8 n)
                      (#:g9 1)
                      (#:g7 (- #:g8 #:g9)))
                 (#:g6 (lambda (#:g5)
                         (#:g2 (* #:g4 #:g5)))
                       #:g7)))))))
  (halt (set! fact #:g1)))
```

Conversion CPS : améliorations (1)

- Cet algorithme de conversion CPS est correct, mais souffre de quelques problèmes :
  - Complexité du code généré : il est inutile de créer des continuations pour les expressions simples qui ne font pas d'appel de fonction
  - Explosion exponentielle de la taille du code qui contient des `if`

Conversion CPS : améliorations (2)

- Une expression est **simple** s'il n'est pas nécessaire de créer une continuation pour son évaluation

- Une définition syntaxique :

$\langle$*simple*$\rangle$ ::= C
| V
| (`lambda` (P_1 `...` P_n) $\langle$*expr*$\rangle$)
| (`set!` V $\langle$*simple*$\rangle$)
| (`+` $\langle$*simple*$\rangle$ $\langle$*simple*$\rangle$)

Conversion CPS : améliorations (3)


```
(define (simple? expr)
  (match expr
    (,const when (constant? const)
     #t)
    (,var when (variable? var)
     #t)
    ((lambda ,params ,E0)
     #t)
    ((set! ,var ,E1)
     (simple? E1))
    ((,op . ,Es) when (primitive? op)
     (all simple? Es))
    (,_
     #f)))

(define (primitive? op) (memq op primitives))

(define primitives
  '(+ - * / eq? = < > <= >=
    null? pair? cons car cdr write))

(define (all pred lst)
  (or (null? lst)
      (and (pred (car lst)) (all pred (cdr lst)))))
```

Conversion CPS : améliorations (4)

- Besoin de 2 fonctions auxiliaires :

`simple-cps` :: $expsimple \rightarrow expcps$

```
> (simple-cps '5)
5
> (simple-cps '(+ x 1))
(+ x 1)
> (simple-cps '(lambda (x) (+ x 1)))
(lambda (#:g0 x) (#:g0 (+ x 1)))
```

`multi-cps` :: $listeexp\ (listeexpsimple \rightarrow expcps) \rightarrow expcps$

```
(multi-cps '(x 5 (+ x 1)) body)
  = (body '(x 5 (+ x 1)))

(multi-cps '(x (f 5) (+ x 1)) body)
  = `(f (lambda (#:g1)
          ,(body '(x #:g1 (+ x 1))))
        5)
```

Conversion CPS : améliorations (5)


```
(define (simple-cps expr)
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     var)

    ((set! ,var ,E1)
     `(set! ,var ,(simple-cps E1)))

    ((,op . ,Es) when (primitive? op)
     `(,op ,@(map simple-cps Es)))

    ((lambda ,params ,E0)
     (let ((k (gensym)))
       `(lambda (,k ,@params) ,(cps E0 k))))))
```

Conversion CPS : améliorations (6)


```
(define (multi-cps exprs body)

  (define (inner sexpr)
    (multi-cps (cdr exprs)
               (lambda (sexprs)
                 (body (cons sexpr sexprs)))))

  (if (null? exprs)
      (body '())
      (let ((expr (car exprs)))
        (if (simple? expr)
            (inner (simple-cps expr))
            (let ((r (gensym)))
              (cps expr
                   `(lambda (,r)
                      ,(inner r))))))))
```

Conversion CPS : améliorations (7)


```
(define (cps E K)
  (match E

    (,s when (simple? s)
     `(,K ,(simple-cps s)))

    ((set! ,v ,E1)
     (multi-cps (list E1)
                (lambda (sexprs)
                  `(,K (set! ,v ,@sexprs)))))

    ((if ,E1 ,E2 ,E3)
     (multi-cps (list E1)
                (lambda (sexprs)
                  `(if ,@sexprs
                       ,(cps E2 K)
                       ,(cps E3 K)))))

    ((begin ,E1 . ,Es)
     (if (null? Es)
         (cps E1 K)
         (let ((r1 (gensym)))
           (cps E1 `(lambda (,r1)
                      ,(cps `(begin ,@Es) K)))))))
```

Conversion CPS : améliorations (8)


```
((,op . ,Es) when (primitive? op)
 (multi-cps Es
            (lambda (sexprs)
              `(,K (,op ,@sexprs)))))

((,E0 . ,Es)
 (multi-cps (cons E0 Es)
            (lambda (sexprs)
              `(,(car sexprs)
                ,K
                ,@(cdr sexprs)))))

(,_
 (error "unknown expression"))))
```

Conversion CPS : améliorations (9)

- Exemple :

```
> (cps '(set! fact
             (lambda (n)
               (if (<= n 1)
                   1
                   (* n (fact (- n 1))))))
       'halt)
(halt (set! fact
            (lambda (#:g0 n)
              (if (<= n 1)
                  (#:g0 1)
                  (fact (lambda (#:g1)
                          (#:g0 (* n #:g1)))
                        (- n 1))))))
```

Conversion CPS : améliorations (10)

- Exemple :

```
> (cps '(+ (if a 1 2)
           (if b 3 4)
           (if c 5 6))
       'halt)
(if a
    (let ((#:g1 1))
      (if b
          (let ((#:g2 3))
            (if c
                (let ((#:g3 5)) (halt (+ #:g1 #:g2 #:g3)))
                (let ((#:g3 6)) (halt (+ #:g1 #:g2 #:g3)))))
          (let ((#:g2 4))
            (if c
                (let ((#:g3 5)) (halt (+ #:g1 #:g2 #:g3)))
                (let ((#:g3 6)) (halt (+ #:g1 #:g2 #:g3)))))))
    (let ((#:g1 2))
      (if b
          (let ((#:g2 3))
            (if c
                (let ((#:g3 5)) (halt (+ #:g1 #:g2 #:g3)))
                (let ((#:g3 6)) (halt (+ #:g1 #:g2 #:g3)))))
          (let ((#:g2 4))
            (if c
                (let ((#:g3 5)) (halt (+ #:g1 #:g2 #:g3)))
                (let ((#:g3 6)) (halt (+ #:g1 #:g2 #:g3)))))))))
```

Conversion CPS : améliorations (11)


```
(define (cps E K)
  (match E

    (,t when (simple? t)
     `(,K ,(simple-cps t)))

    ((set! ,v ,E1)
     (multi-cps (list E1)
                (lambda (sexprs)
                  `(,K (set! ,v ,@sexprs)))))

    ((if ,E1 ,E2 ,E3)
     (if (not (variable? K))
         (let ((k (gensym)))
           `((lambda (,k) ,(cps E k)) ,K))
         (multi-cps (list E1)
                    (lambda (sexprs)
                      `(if ,@sexprs
                           ,(cps E2 K)
                           ,(cps E3 K))))))

    ...
```

Conversion CPS : améliorations (12)

- Exemple :

```
> (cps '(+ (if a 1 2)
           (if b 3 4)
           (if c 5 6))
        'halt)
(let ((#:g6
       (lambda (#:g1)
         (let ((#:g5
                (lambda (#:g2)
                  (let ((#:g4
                         (lambda (#:g3)
                           (halt
                            (+ #:g1 #:g2 #:g3)))))
                    (if c (#:g4 5) (#:g4 6))))))
           (if b (#:g5 3) (#:g5 4))))))
  (if a (#:g6 1) (#:g6 2)))
```

Compiler Scheme à JavaScript (1)

- La conversion CPS est l'algorithme fondamental d'un compilateur de Scheme à Scheme-CPS (le langage Scheme sans l'appel non-terminal)
- La conversion CPS peut également servir de base pour un compilateur de Scheme à JavaScript car **JavaScript possède les fermetures**
- Malheureusement, **JavaScript ne possède pas l'appel terminal** (tout les appels consomment un espace sur la pile)
- La conversion CPS ne peut donc pas être utilisée sans modification

Compiler Scheme à JavaScript (2)

- Scheme et JavaScript ont :
  - **typage dynamique**
  - **gestion automatique de la mémoire**
  - **fermetures**
- JavaScript :
  - **a un système objet de type prototype**
  - **a une syntaxe infixe**
  - **le type “liste” n’est pas prédéfini**
  - **n’a pas l’appel terminal**

Compiler Scheme à JavaScript (3)

- Comparaison de Scheme et JavaScript :

Scheme

```
(define pi 3.1416)

(define c (cons 1 '()))

(define longueur
  (lambda (x)
    (if (pair? x)
      (+ 1 (longueur (cdr x)))
      0)))

(define adder
  (lambda (x)
    (lambda (y)
      (+ x y))))

(define inc (adder 1))

(display (inc (* 2 pi)))
```

JavaScript

```
function Pair(car,cdr)
{ this.car = car
  this.cdr = cdr
}

pi = 3.1416

c = new Pair(1,null)

longueur =
  function(x)
    { return (x instanceof Pair)
             ? 1 + longueur(x.cdr)
             : 0
    }

adder =
  function(x)
    { return function(y)
               { return x+y }
    }

inc = adder(1)

document.write(inc(2*pi))
```

Compilation de l'appel terminal (1)

- La conversion CPS de

```
(define fact
  (lambda (n)
    (if (<= n 1) 1 (* n (fact (- n 1))))))
```

donne

```
(set! fact
      (lambda (g0 n)
        (if (<= n 1)
            (g0 1)
            (fact (lambda (g1)
                    (g0 (* n g1)))
                  (- n 1)))))
```

dont la traduction en JavaScript donne

```
fact = function(g0,n)
         { return (n<=1)
                  ? g0(1)
                  : fact(function(g1)
                           { return g0(n*g1) },
                         n-1)
         }
```

Compilation de l'appel terminal (2)

- Dans un langage qui n'a pas d'appel terminal (comme JavaScript, C, ...) on peut obtenir le même effet avec une **trampoline**

- Voici un exemple simple pour expliquer :

```
(define f (lambda (n) (f (+ n 1))))
```

  dont la traduction en JavaScript donne

```
f = function(n) { return f(n+1) }
```

- Chaque appel à `f` va consommer de l'espace sur la pile d'exécution

Compilation de l'appel terminal (3)

- Pour éviter d'accumuler des blocs d'activation sur la pile, il faut **sortir de la fonction appelante** avant d'exécuter la fonction appelée
- Cela peut se faire en **retournant une fonction qui effectue l'appel** (une sorte de continuation sans paramètre)
- Donc

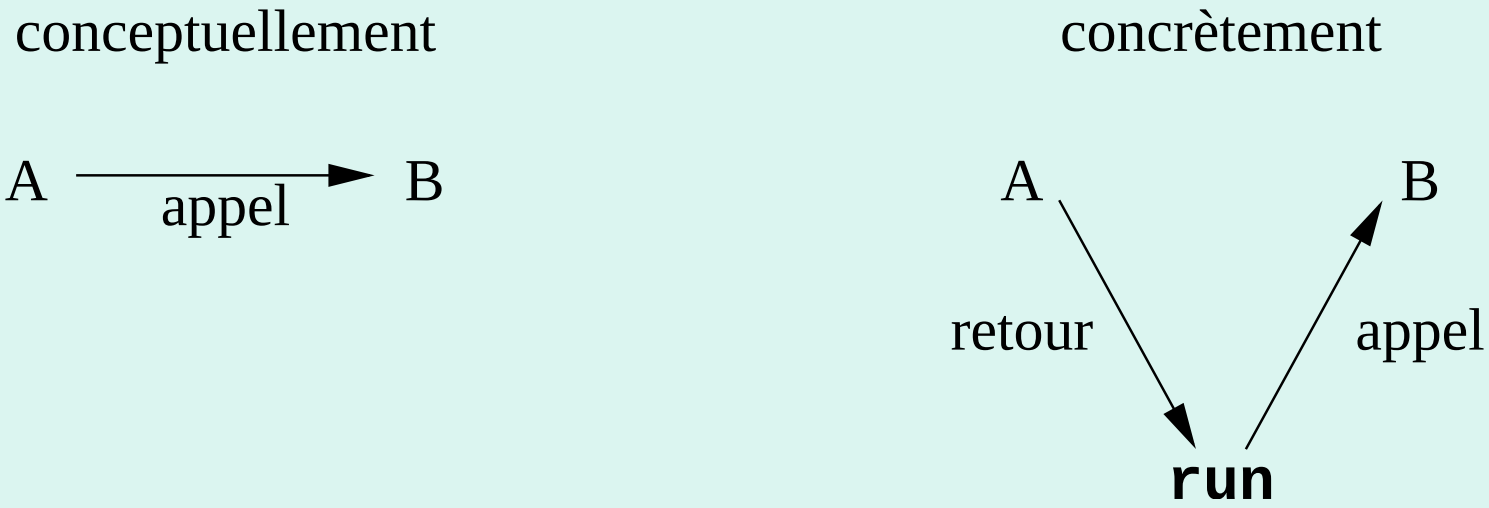
```
f = function(n) { return f(n+1) }
```

devient

```
f = function(n)
      { return function() { return f(n+1) } }

function run(pc) { while (pc) pc = pc() }

run( function() { return f(0) } )
```

Compilation de l'appel terminal (4)

- La fonction `run` c'est la trampoline (on peut sauter d'une fonction A à une fonction B seulement en "rebondissant" par la trampoline)



- Pour terminer il suffit de retourner `false`

```
halt = function(r) { return false }
```

Compilation de l'appel terminal (5)

- Pour compiler de Scheme à JavaScript, il faut employer les règles de traduction suivantes après la conversion CPS

$T[\,c\,]$	=	c
$T[\,v\,]$	=	v
$T[\,$`(set!` v x`)`$\,]$	=	v `=` $T[x]$
$T[\,$`(lambda (`p_1`...)` x`)`$\,]$	=	`function(`p_1`,...)` `{ return` $T[x]$ `}`
$T[\,$`(if` x y z`)`$\,]$	=	$T[x]$ `?` $T[y]$ `:` $T[z]$
$T[\,$`(+` x y`)`$\,]$	=	$T[x]$`+`$T[y]$
$T[\,$`(`x y`...)`$\,]$	=	`function()` `{ return` $T[x]$`(`$T[y]$`,...)` `}`

Compilation de l'appel terminal (6)

- Exemple de traduction à JavaScript :

```
(set! fact          ; après la conversion CPS
      (lambda (g0 n)
        (if (<= n 1)
            (g0 1)
            (fact (lambda (g1) (g0 (* n g1)))
                  (- n 1)))))

fact =              ; résultat de la traduction
  function(g0,n)
    { return
        (n<=1)
        ? function()
            { return g0(1) }
        : function()
            { return
                fact(function(g1)
                       { return
                           function()
                             { return g0(n*g1) }
                       },
                     n-1)
            }
    }
```

`call/cc` (1)

- La compilation d'un programme Scheme peut reposer sur la conversion CPS pour transformer le programme de sa **forme directe** (avec continuations implicites) en un programme en **forme CPS** (avec continuations explicites)
- Tout langage de programmation se modélise avec des continuations, mais en Scheme les continuations sont aussi des **données de première classe**
- La fonction prédéfinie `call/cc` permet à un programme en forme directe d'avoir accès à la continuation implicite (**capture de continuation**/**réification de continuation**)

call/cc (2)

- La fonction `call/cc` recoit une continuation (implicite) et un seul paramètre, une fonction qui sera appelée avec cette continuation (implicite) et la **continuation réifiée** comme unique paramètre

- Exemple :

```
> (+ 1 (call/cc (lambda (c) 3)))
4
> (+ 1 (call/cc (lambda (c) (c 3))))
4
> (+ 1 (call/cc (lambda (c) (+ 2 (c 3)))))
4
> (define f #f)
> (+ 1 (call/cc (lambda (c) (set! f c) 3)))
4
> (+ 2 (f 7))
8
```

call/cc (3)

- Un appel à une **continuation réifiée** abandonne la continuation présente et continue le calcul dénoté par la continuation

- Implantation (dans le “monde CPS”) :

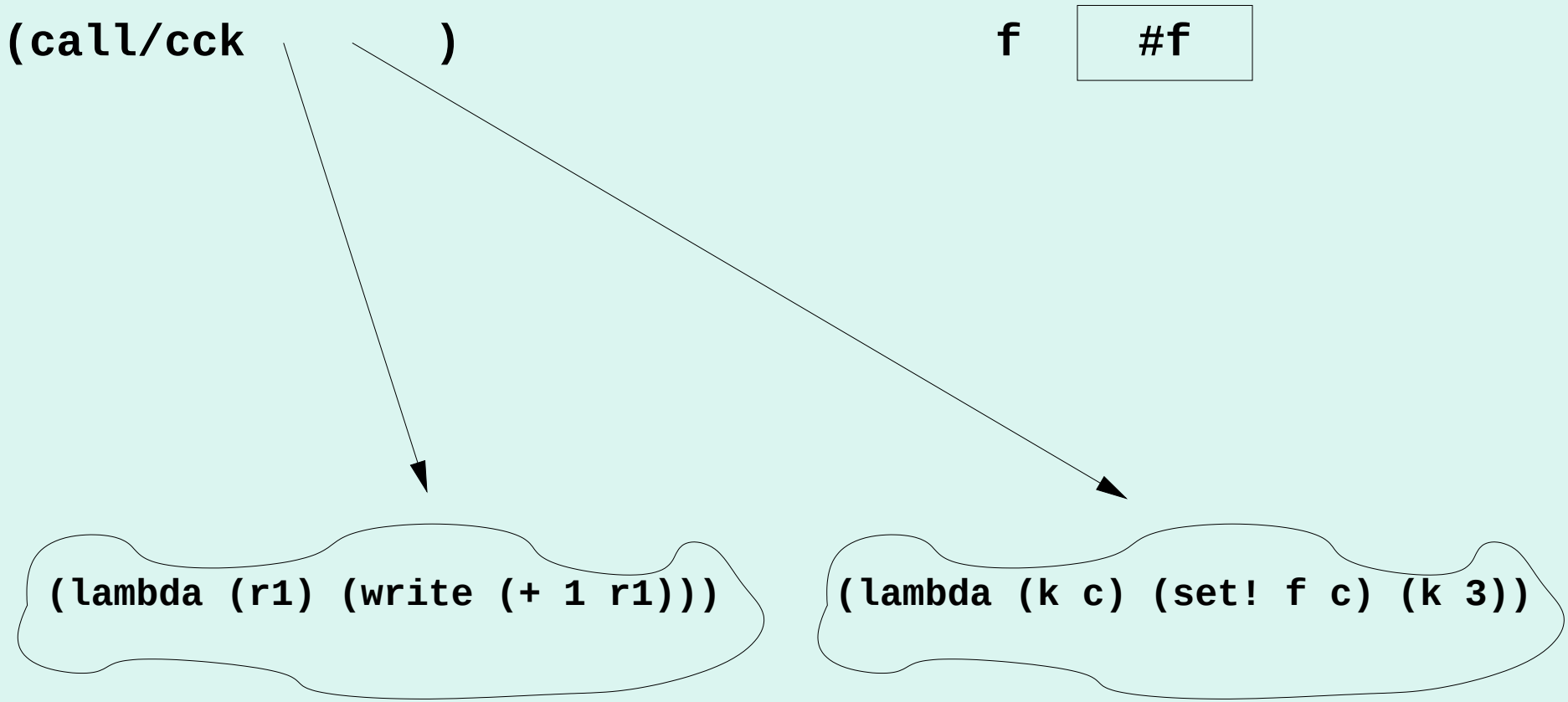
```
(define (call/cck k1 receveur)
  (receveur k1
            (lambda (k2 r) (k1 r))))
```

- Exemple, après conversion CPS :

```
(call/cck (lambda (r1) (write (+ 1 r1)))
          (lambda (k c) (set! f c) (k 3)))
  => imprime 4

(f (lambda (r2) (write (+ 2 r2))) 7)
  => imprime 8
```

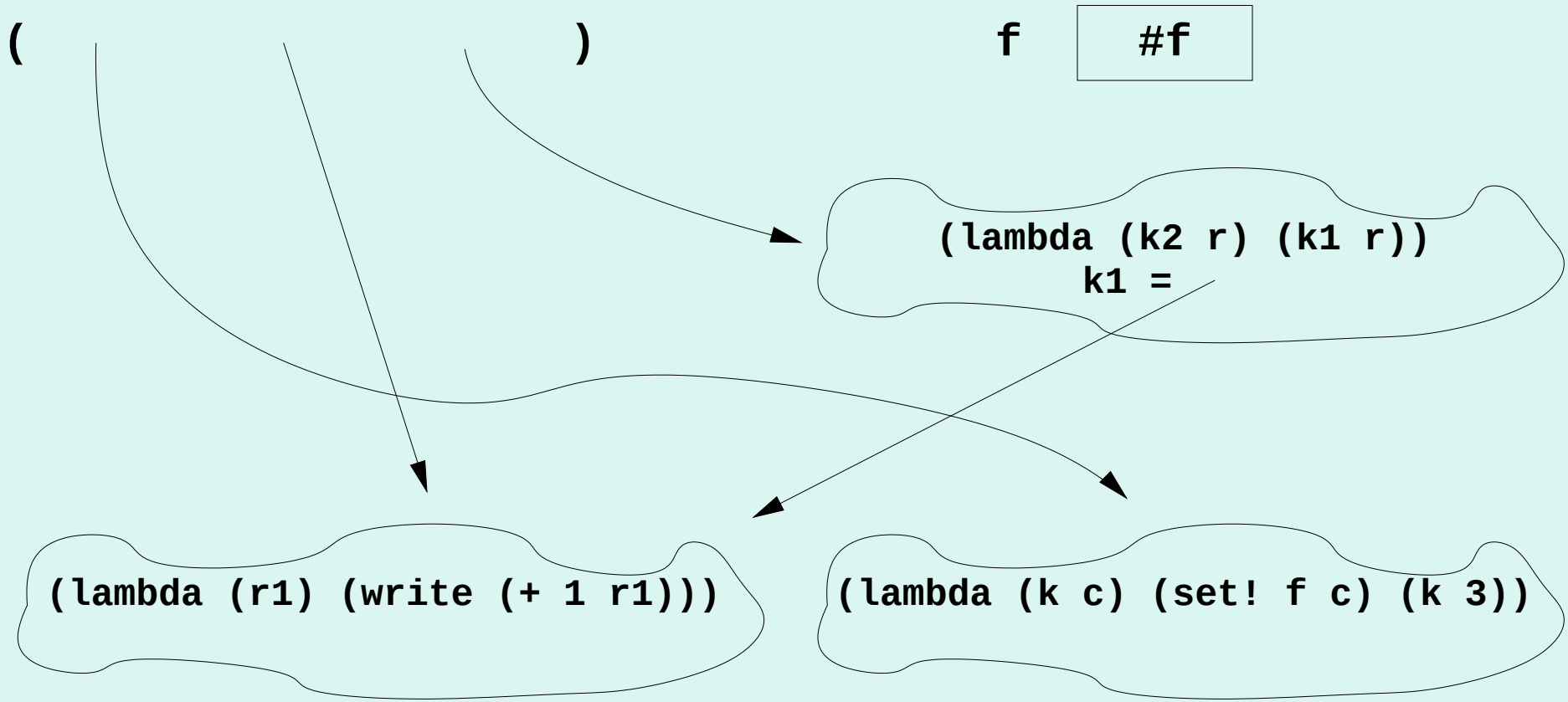
call/cc (4)

- Trace de `(call/cck (lambda (r1) ...) (lambda (k c) ...))`



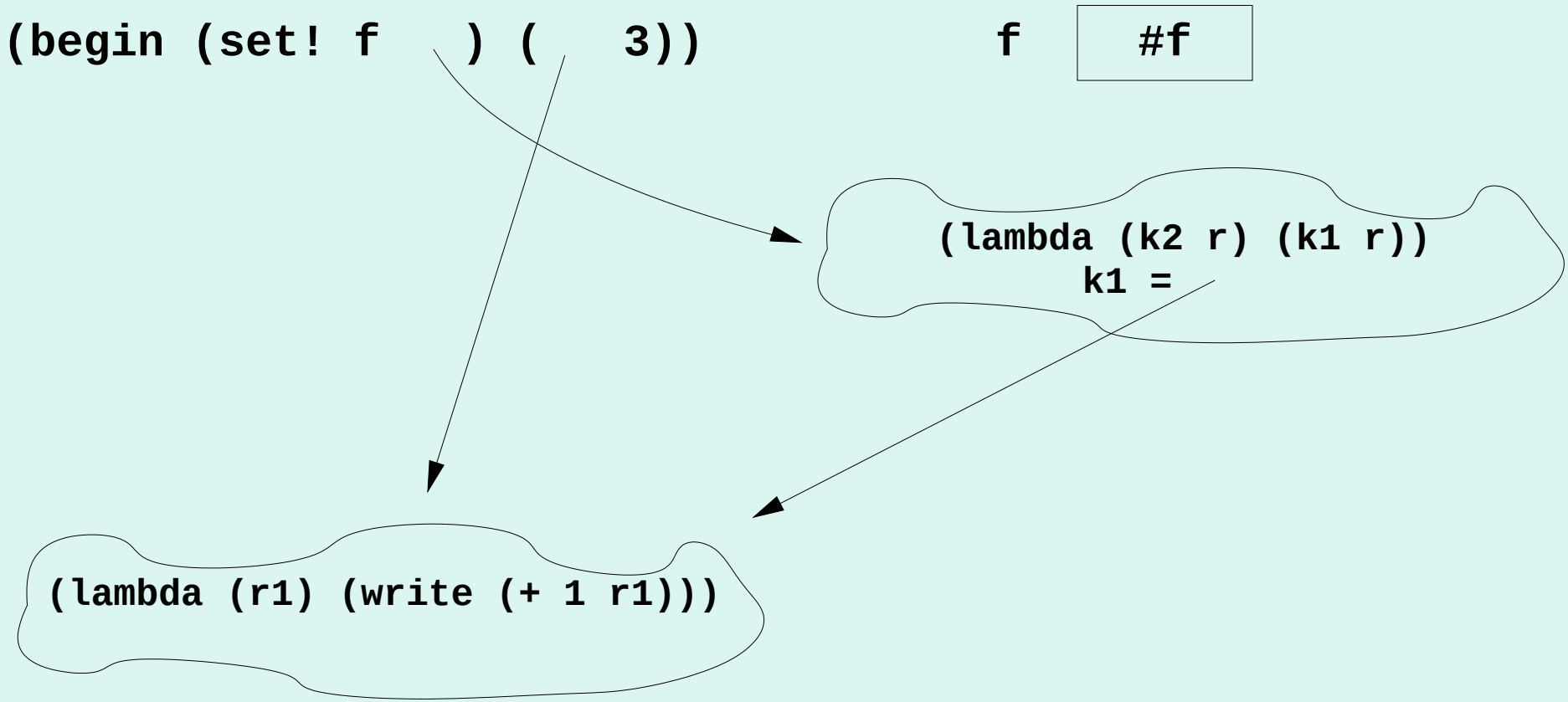
call/cc (5)

- Trace de `(call/cck (lambda (r1) ...) (lambda (k c) ...))`



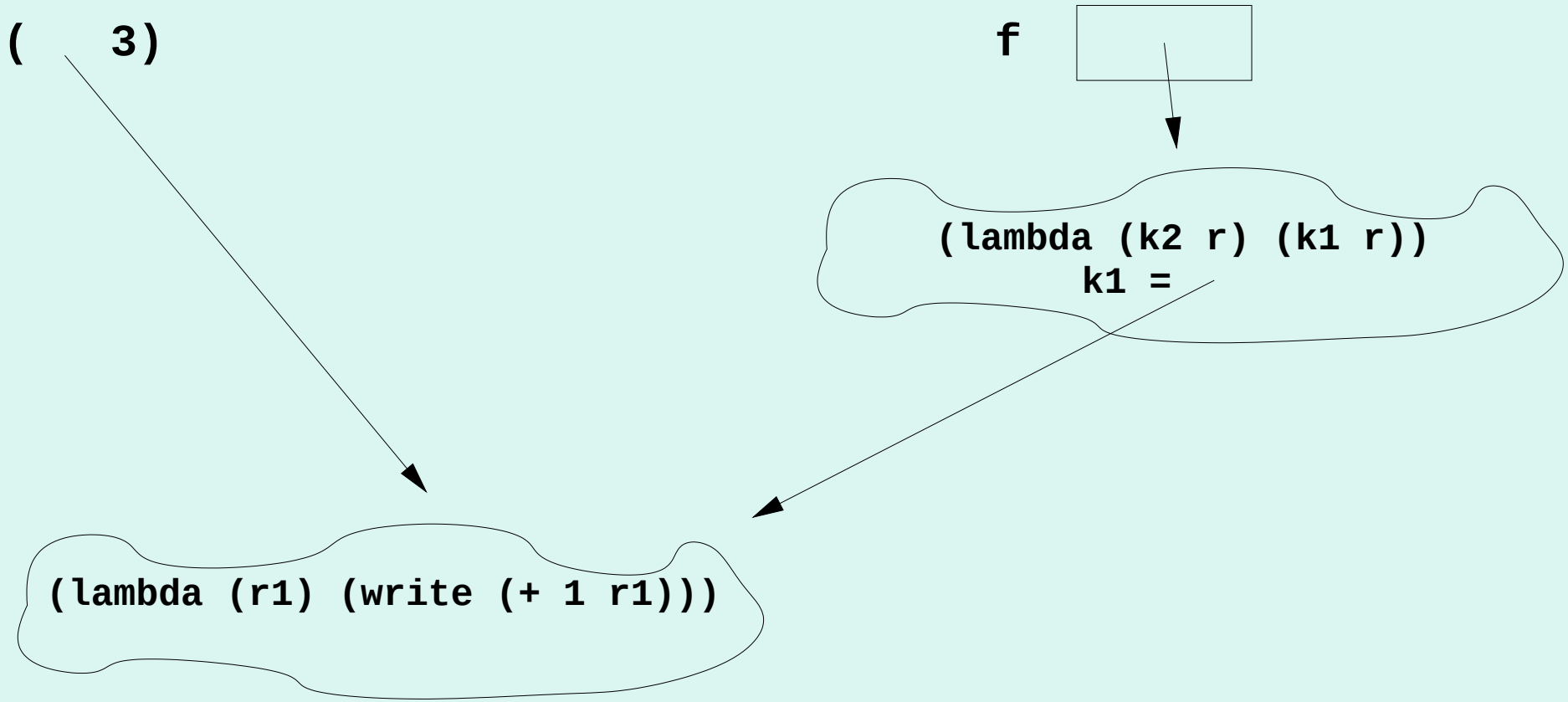
call/cc (6)

- Trace de
  `(call/cck (lambda (r1) ...) (lambda (k c) ...))`



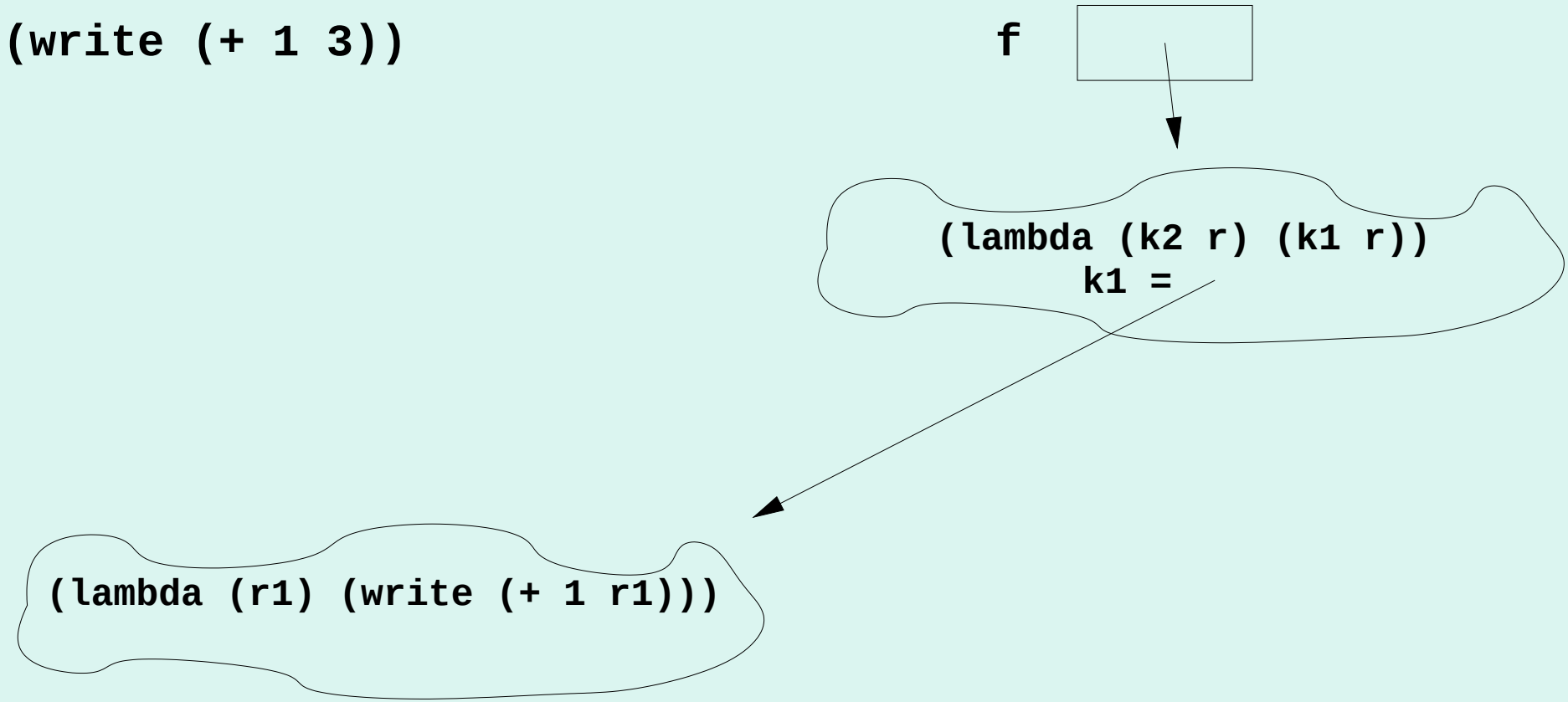
call/cc (7)

- Trace de `(call/cck (lambda (r1) ...) (lambda (k c) ...))`



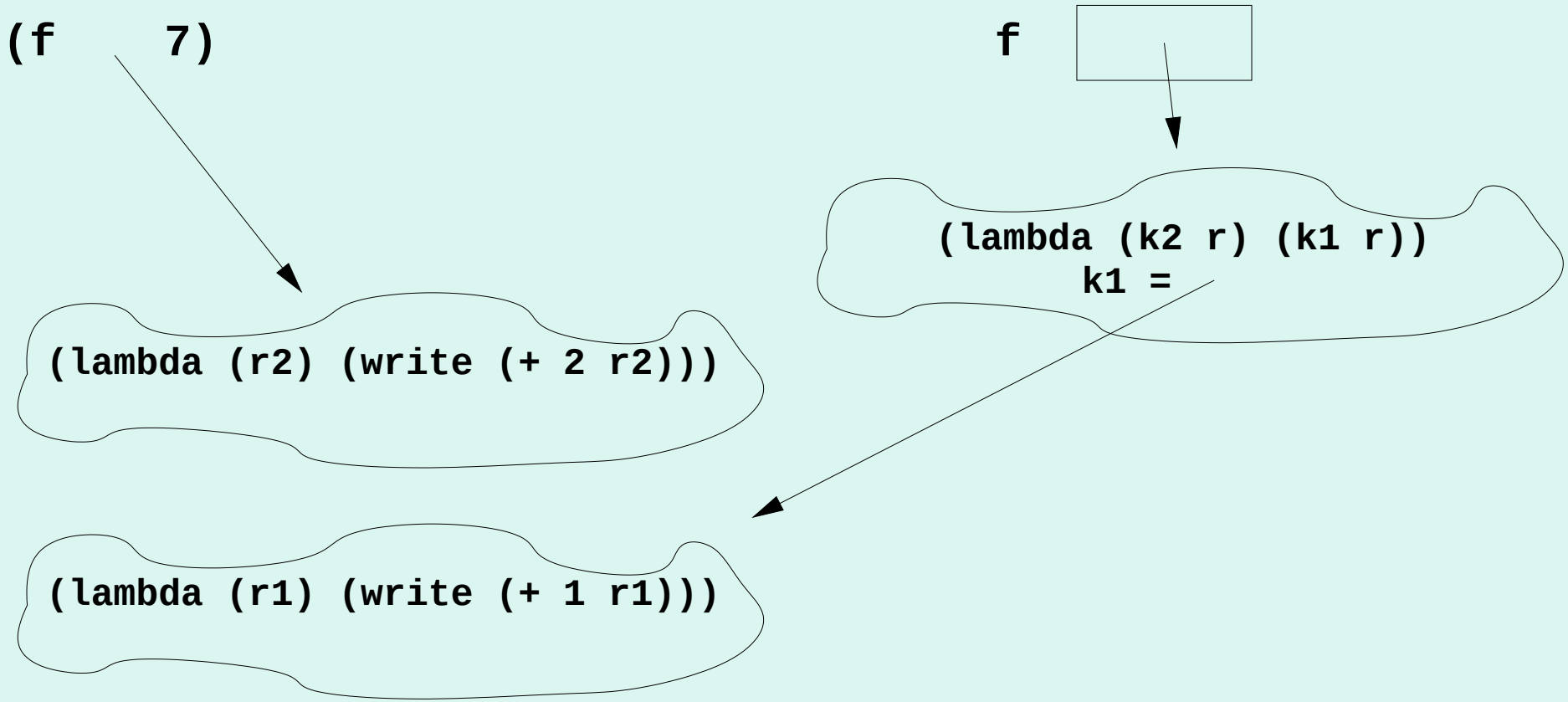
call/cc (8)

- Trace de `(call/cck (lambda (r1) ...) (lambda (k c) ...))`



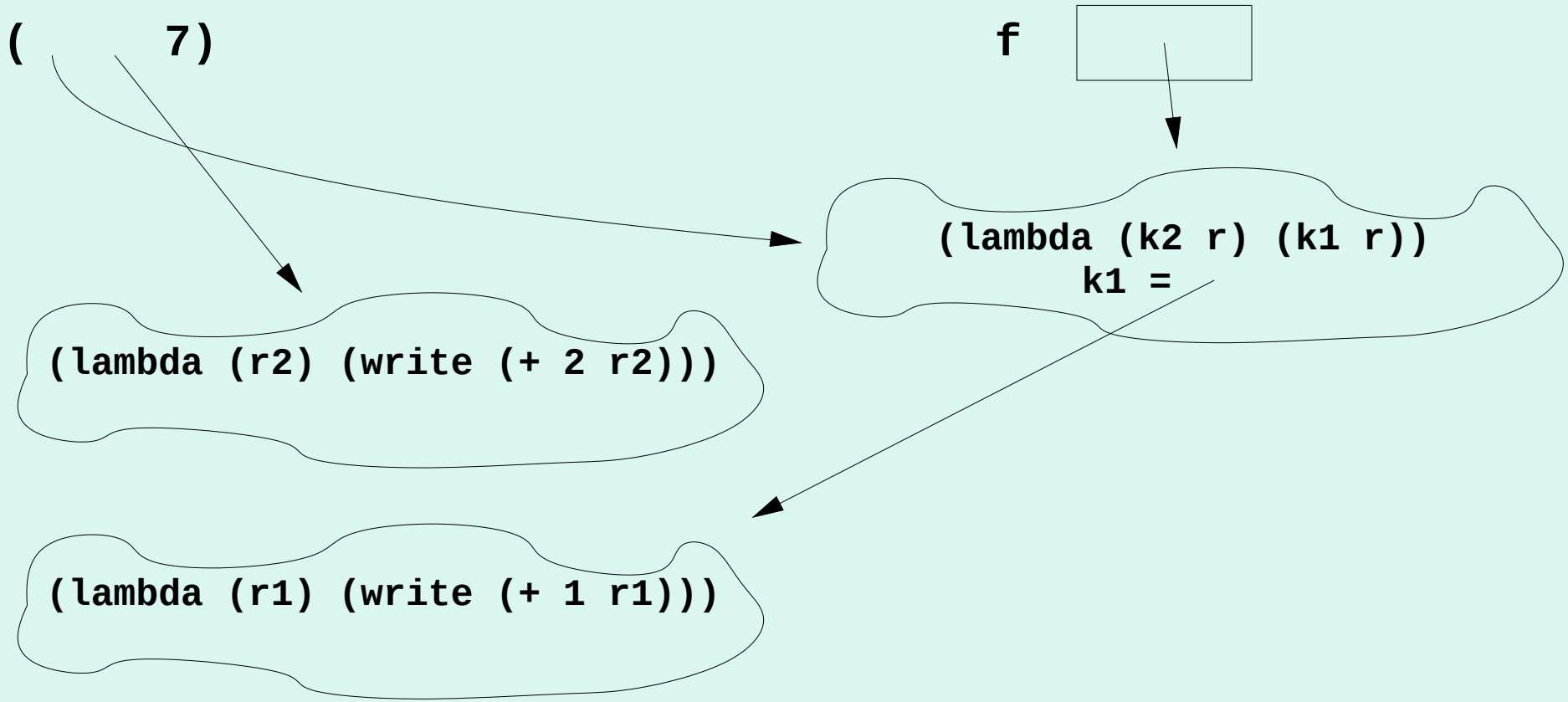
call/cc (9)

- Trace de
  `(f (lambda (r2) (write (+ 2 r2))) 7)`



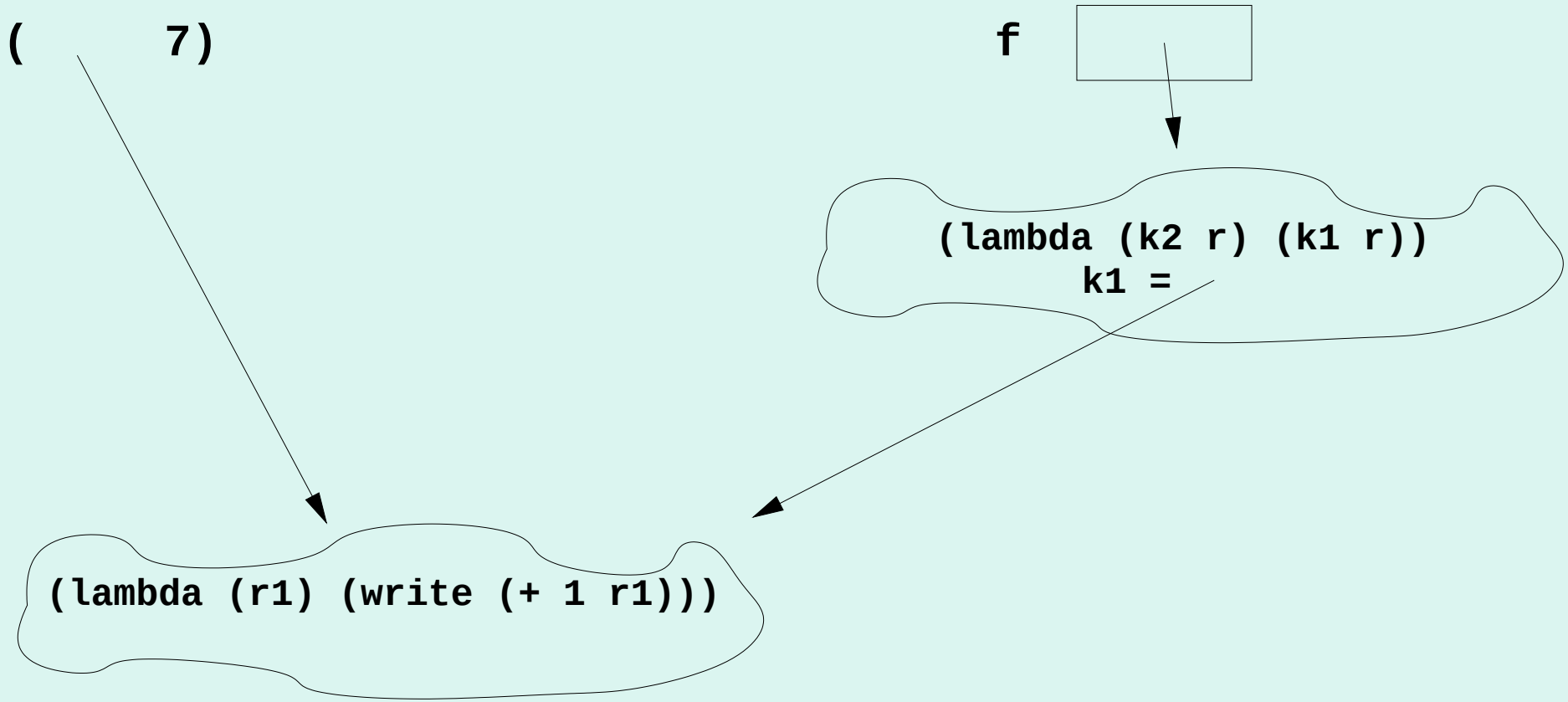
call/cc (10)

- Trace de
  `(f (lambda (r2) (write (+ 2 r2))) 7)`



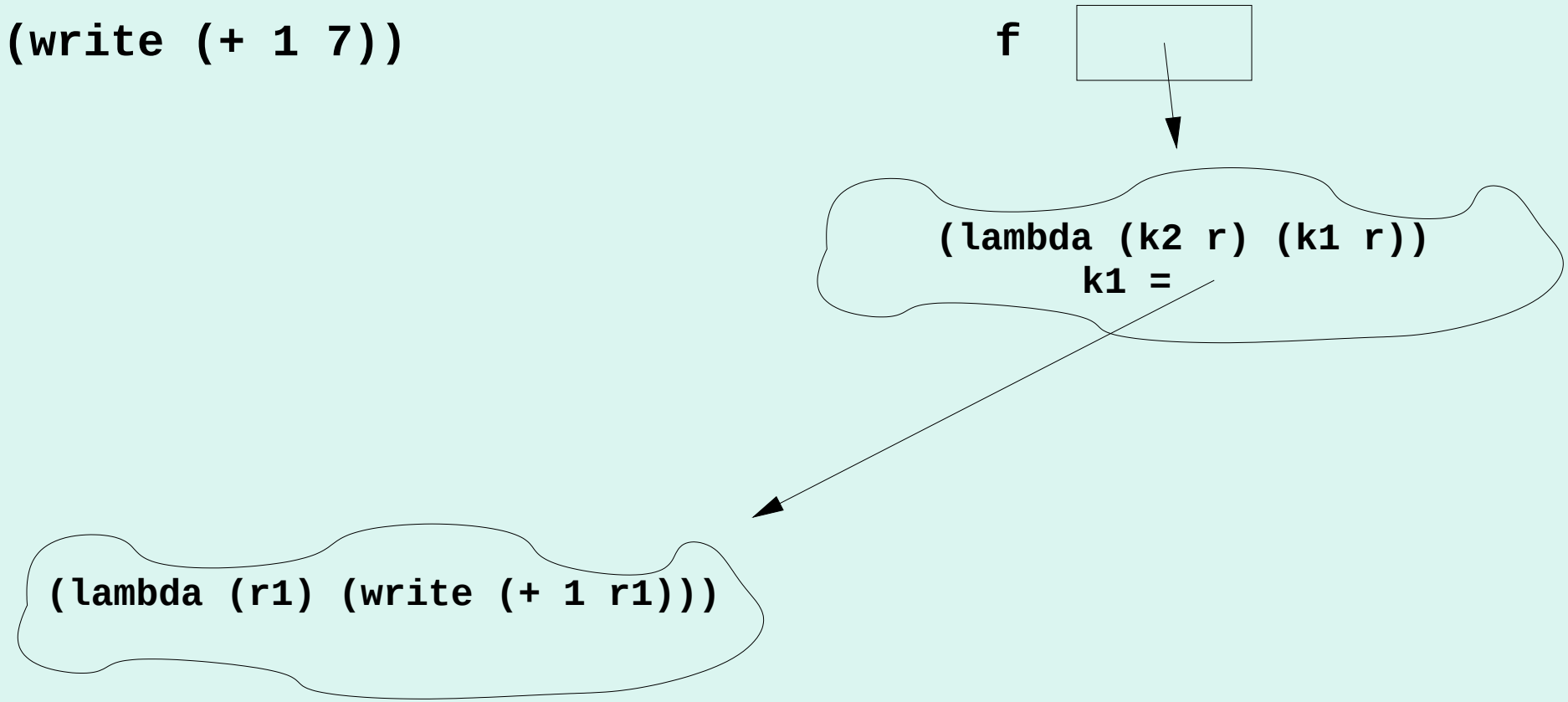
call/cc (11)

- Trace de
  `(f (lambda (r2) (write (+ 2 r2))) 7)`



call/cc (12)

- Trace de
  `(f (lambda (r2) (write (+ 2 r2))) 7)`



Continuations : applications

- Les continuations permettent d'implanter plusieurs formes de contrôle :
  - **Traitement d'exception**
  - **Retour arrière** ("backtracking")
  - **Coroutines**, **processus**, **threads**

Traitement d'exception

- Une continuation représente un point du calcul; l'appel de la continuation permet de poursuivre immédiatement ce calcul (en **échappant la continuation du calcul présent**)
- Exemple :

```
> (define (map-inv-echap lst)
    (call/cc (lambda (return)
               (map (lambda (x)
                      (if (= x 0)
                          (return "erreur")
                          (/ 1 x)))
                    lst))))

> (map-inv-echap '(1 2 3))
(1 1/2 1/3)

> (map-inv-echap '(1 0 3))
"erreur"
```

Retour arrière (1)

- Dans le retour arrière on désire abandonner une branche de calcul pour revenir à un **point antérieur du calcul** et essayer une **nouvelle branche de calcul**

- On se sert d'une continuation pour représenter le point antérieur du calcul où une nouvelle branche est explorée

- Donc l'appel de cette continuation abandonnera la branche de calcul présente pour en essayer une nouvelle

Retour arrière (2)

- Pour que l'approche soit facile à utiliser, on utilisera une variable `fail` qui contient une fonction sans paramètre qui permet d'abandonner la branche de calcul présente

```
(define fail
  (lambda ()
    (error "can't backtrack")))
```

Retour arrière (3)

- Voici un exemple de calcul sans retour arrière pour trouver un triplet x, y, z tel que $1 \leq x, y, z \leq 9$ et $x^2 = y^2 + z^2$:

```
(let loop1 ((x 1))
  (if (<= x 9)
      (let loop2 ((y 1))
        (if (<= y 9)
            (let loop3 ((z 1))
              (if (<= z 9)
                  (if (= (* x x)
                         (+ (* y y) (* z z)))
                      (write (list x y z))
                      (loop3 (+ z 1)))
                  (loop2 (+ y 1))))
            (loop1 (+ x 1))))
      (error "impossible")))
```

- Nous allons essayer de faire le même calcul par retour arrière

Retour arrière (4)


```
(define (in-range a b)
  (call/cc
   (lambda (cont)
     (enumerate a b cont))))

(define (enumerate a b cont)
  (if (> a b)
      (fail)
      (let ((save fail))
        (set! fail
          (lambda ()
            (set! fail save)
            (enumerate (+ a 1) b cont)))
        (cont a))))

(let ((x (in-range 1 9))
      (y (in-range 1 9))
      (z (in-range 1 9)))
  (if (= (* x x)
         (+ (* y y) (* z z)))
      (write (list x y z))
      (fail)))
```

Processus et coroutines (1)

- Un **processus** c'est un **processeur virtuel** en charge d'un certain calcul
- Sur un monoprocesseur il y a au maximum un processus en exécution (le processeur est **multiplexé** parmis les **processus exécutables** par l'ordonnanceur du système d'exploitation)
- Une **coroutine** c'est un processus qui décide à quel moment et à quelle autre coroutine céder le processeur
- Un processus est soit **en exécution** ou **est suspendu dans un certain état**

Processus et coroutines (2)

- On peut représenter l'**état d'un processus suspendu** par une **continuation stockée dans le descripteur de processus**
- Pour **continuer l'exécution du processus** on appèle sa continuation
- Pour **suspendre un processus** on capture sa continuation et on la sauve dans le descripteur de processus

Coroutines (1)

- Opérations pour une librairie de coroutines :

`(corout F)` Crée et retourne une coroutine dont l'exécution est un appel à la fonction sans-paramètre *F*. La coroutine n'exécute pas tout de suite, c'est seulement en exécutant un `(yield X)` que l'exécution de la coroutine *X* se fera.

`(start C)` Démarre l'exécution de la coroutine primordiale *C*. La coroutine primordiale, c'est la première coroutine à exécuter. Un programme ne peut faire qu'un seul appel à `start`.

`(yield C)` Suspend la coroutine présente (pour qu'on puisse continuer son exécution plus tard), et continue l'exécution de la coroutine *C*.

Coroutines (2)

- Exemple d'utilisation :

```
(define a
  (corout (lambda ()
            (let loop ()
              (display "A")
              (yield b)
              (loop)))))

(define b
  (corout (lambda ()
            (let loop ()
              (display "B")
              (yield a)
              (loop)))))

(start a)
```

ABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABAB

Coroutines (3)

- Implantation des opérations

```
(define current #f)

(define (corout thunk)  ; coroutine = paire
  (cons (lambda (dummy) (thunk))
        '()))

(define (start coroutine)
  (set! current coroutine)
  ((car current) #f))

(define (yield coroutine)
  (call/cc
    (lambda (k)
      (set-car! current k)
      (start coroutine))))
```

Processus (1)


```
(define (fib n)
  (yield)
  (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2)))))

(spawn (let loop ((i 0))
         (if (< i 100)
           (begin
             (display "A")
             (fib 17)
             (loop (+ i 1))))))

(spawn (let loop ((i 0))
         (if (< i 100)
           (begin
             (display "B")
             (fib 22)
             (loop (+ i 1))))))

(boot)
```

```
ABAAAAAAAAAAABAAAAAAAAAAABAAAAAAAAAAABAAAAAAAAA
AABAAAAAAAAAABAAAAAAAAAAABAAAAAAAAAAABAAAAAAAAA
AAABAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBdone
```

Processus (2)


```
;; Queues.

(define (next q) (vector-ref q 0))
(define (prev q) (vector-ref q 1))
(define (next-set! q x) (vector-set! q 0 x))
(define (prev-set! q x) (vector-set! q 1 x))

(define (empty? q) (eq? q (next q)))

(define (queue) (init (vector #f #f)))

(define (init q)
  (next-set! q q)
  (prev-set! q q)
  q)
```

Processus (3)


```
;; Queues.

(define (deq x)
  (let ((n (next x)) (p (prev x)))
    (next-set! p n)
    (prev-set! n p)
    (init x)))

(define (enq q x)
  (let ((p (prev q)))
    (next-set! p x)
    (next-set! x q)
    (prev-set! q x)
    (prev-set! x p)
    x))
```

Processus (4)


```
;; Process scheduler.

(define (boot)
  ((call/cc (lambda (k)
              (set! graft k)
              (schedule)))))

(define graft #f)
(define current #f)
(define readyq (queue))

(define (process cont)
  (init (vector #f #f cont)))

(define (cont p) (vector-ref p 2))
(define (cont-set! p x) (vector-set! p 2 x))

(define (make-process thunk)
  (enq readyq
       (process (lambda (r)
                  (graft (lambda ()
                           (end (thunk))))))))

(define-macro (spawn expr)
  `(make-process (lambda () ,expr)))
```

Processus (5)


```
(define (schedule)
  (if (empty? readyq)
      (graft (lambda () (pp 'done)))
      (let ((p (deq (next readyq))))
        (set! current p)
        ((cont p) #f))))

(define (end result) (schedule))

(define (yield)
  (call/cc (lambda (k)
             (cont-set! current k)
             (enq readyq current)
             (schedule))))
```

Processus (6)


```
;; Preemptive version.

(define (boot)
  (##heartbeat-interval-set! 0.1)
  (##interrupt-vector-set! 1 yield)
  ((call/cc (lambda (k)
              (set! graft k)
              (schedule))))
  (##interrupt-vector-set! 1 list))
```

Environnement dynamique (1)

- Les **objets paramètres** (“paramètres” dans ce qui suit) sont une extension à Scheme permettant la liaison dynamique

- Opérations :
  - `(make-parameter V)` : crée un paramètre
  - `(P)` : retourne la valeur du paramètre *P*
  - `(P V)` : change la valeur du paramètre *P*
  - `(parameterize ((P V)) E)` : évalue *E* dans un contexte où le paramètre *P* a la valeur *V*

- `parameterize` est l’analogue de `let` pour la liaison dynamique

Environnement dynamique (2)

- Exemple pratique : entrées/sorties redirigées
- En Scheme, (`display` *OBJ PORT*) imprime l'objet *OBJ* sur le port de sortie *PORT*
- Si *PORT* n'est pas spécifié on utilise (`current-output-port`), c'est-à-dire

```
(define (display obj
                 #!optional
                 (port (current-output-port)))
  ...)
```

Environnement dynamique (3)

- Par défaut (`current-output-port`) retourne le port pour la sortie standard du programme (“stdout”)

- On peut rediriger le port de sortie temporairement à l’aide de (`with-output-to-file` *filename thunk*) :

```
(define (rapport)
  (display "NOTES DE L'EXAMEN\n")
  (display "-----------------\n")
  ...)

(rapport) ; affiche sur stdout

(with-output-to-file "notes.txt" rapport)

(rapport) ; affiche sur stdout
```

Environnement dynamique (4)

- Implantation possible de `display` :

```
(define current-output-port (make-parameter stdout))

(define current-radix (make-parameter 10))

(define (display obj
                 #!optional
                 (port (current-output-port)))
  (cond ((string? obj)
         (for-each (lambda (c) (write-char c port))
                   (string->list obj)))
        ((number? obj)
         (display (number->string obj
                                  (current-radix))
                  port))
        ...))

(define (with-output-to-file filename thunk)
  (parameterize ((current-output-port
                  (open-output-file filename)))
    (let ((result (thunk)))
      (close-output-port (current-output-port))
      result)))
```

Environnement dynamique (5)

- Implantation des paramètres :

```
(define (make-parameter val)
  (let ((key (list 'key))) ; clé unique
    (lambda (#!optional (newval key))
      (if (eq? newval key)
          val
          (set! val newval)))))

(define-macro (parameterize . rest)
  (match (cons 'parameterize rest)
    ((parameterize ((,p ,v)) . ,body)
     (let ((psym (gensym))
           (vsym (gensym))
           (save (gensym)))
       `(let ((,psym ,p)
              (,vsym ,v))
          (let ((,save (,psym)))
            (,psym ,vsym)
            (let ((result (let () ,@body)))
              (,psym ,save)
              result)))))))
```

Environnement dynamique (6)

- Cette implantation ne fonctionne pas dans un contexte multithread car **les mutations du paramètre par différents threads interfèrent**

```
(thread-start!
 (make-thread (lambda ()
                (with-output-to-file
                 "notes1.txt"
                 rapport))))

(thread-start!
 (make-thread (lambda ()
                (with-output-to-file
                 "notes2.txt"
                 rapport))))
```

- Il est possible que les rapports soient envoyés dans les fichiers **et sur stdout**, et que les fichiers n'aient **pas le même contenu**

Environnement dynamique (7)

- Il faut que chaque thread possède son propre **environnement dynamique**

- Lorsque le contrôle passe d'un thread à un autre, il faut sauver l'environnement dynamique courant dans l'état du thread et installer l'environnement dynamique du prochain thread

- Idée : utiliser notre propre version de `call/cc` pour sauvegarder l'état des threads

```
(define dyn-env '())

(define (call/cc-capturing-dyn-env receiver)
  (let ((save dyn-env))
    (let ((result (call/cc receiver)))
      (set! dyn-env save)
      result)))
```

Environnement dynamique (8)

- Nouvelle implantation des paramètres :

```
(define get-key (list 'get-key))

(define (make-parameter val)
  (let ((key (list 'key)))
    (lambda (#!optional (newval key))
      (if (eq? newval get-key)
          key
          (let ((x (assq key dyn-env)))
            (if (eq? newval key)
                (if x
                    (cdr x)
                    val)
                (if x
                    (set-cdr! x newval)
                    (set! val newval)))))))))
```

Environnement dynamique (9)


```
(define-macro (parameterize . rest)
  (match (cons 'parameterize rest)
    ((parameterize ((,p ,v)) . ,body)
     (let ((psym (gensym))
           (vsym (gensym))
           (save (gensym)))
       `(let ((,psym ,p)
              (,vsym ,v))
          (call/cc-capturing-dyn-env
            (lambda (_)
              (set! dyn-env
                    (cons (cons
                           (,psym get-key)
                           ,vsym)
                          dyn-env))
              ,@body))))))))
```

Graphes de Flux

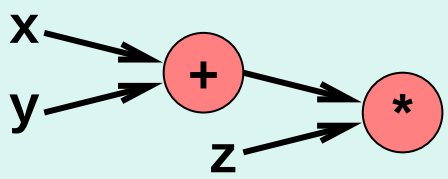
- Le **flux de donnée** c'est le trajet que suit une donnée pendant l'exécution du code

- Le **flux de contrôle** c'est le trajet que suit le point d'exécution pendant l'exécution du code

- Ces informations se représentent normalement à l'aide de **graphes**
  - le **Data Flow Graph** (DFG)
  - le **Control Flow Graph** (CFG)

Graphe de Flux de Donnée (DFG)

- Le DFG indique
  - les **dépendences de données** entre opérations, ce qui restreint l'ordre d'exécution des opérations, p.ex. dans `(x+y)*z` il faut faire + avant `*`



  - l'**origine des données** contenues dans les variables, structures, etc

```
(define (f x i)
  (let ((v (vector (* x 2) (sqrt x))))
    (+ 1 (vector-ref v i))))
```

Graphes de Flux de Contrôle (CFG)

- Le **flux de contrôle** c'est le trajet que suit le point d'exécution pendant l'exécution du code

- Le flux est guidé par les **instructions de contrôle** (branchements inconditionnels/conditionnels, appels et retours de fonctions, `goto`, `if`, `switch`, `while`, ...)

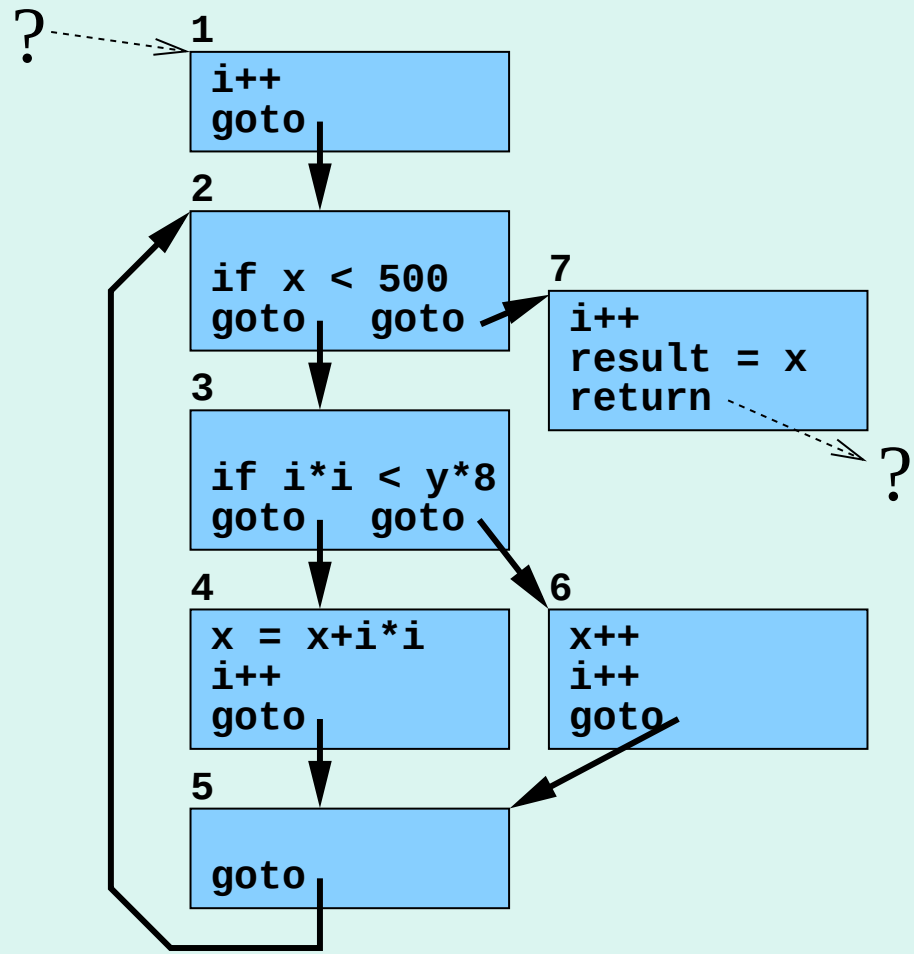
- Un CFG est une représentation intermédiaire qui
  - représente de façon explicite les contraintes sur le flux de contrôle du programme
  - facilite les analyses reliées au flux de contrôle (p.e. est ce que l'instruction X est atteignable à partir de l'instruction Y)

Structure d'un CFG

- Un CFG est un **graphe dirigé**
  - chaque noeud ("basic block") représente une section de code **sans branchements**
  - une arête représente un branchement possible (statique ou dynamique)
  - un basic block se termine toujours par une **instruction de branchement**
- Concept : le bloc A **domine** le bloc B ssi le flux passe toujours par A avant de passer au bloc B
- On dit qu'une analyse/optimisation est
  - **locale** : au niveau d'un B.B.
  - **globale** ou **intraprocédurale** : niveau fonction
  - **interprocédurale** : niveau module ou programme

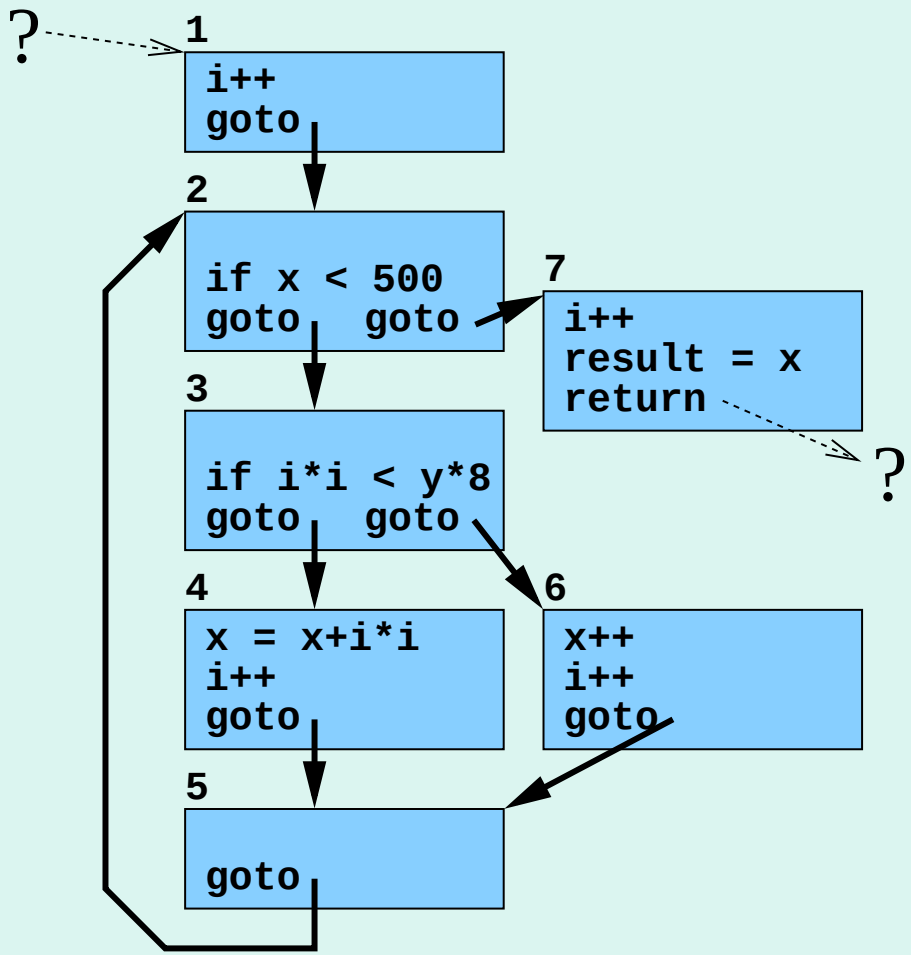
CFG : exemple


```
int f( int x, int y, int i ) {
  i++;
  while (x < 500) {
    if (i*i < y*8) {
      x = x+i*i;
      i++;
    } else {
      x++;
      i++;
    }
  }
  i++;
  return x;
}
```



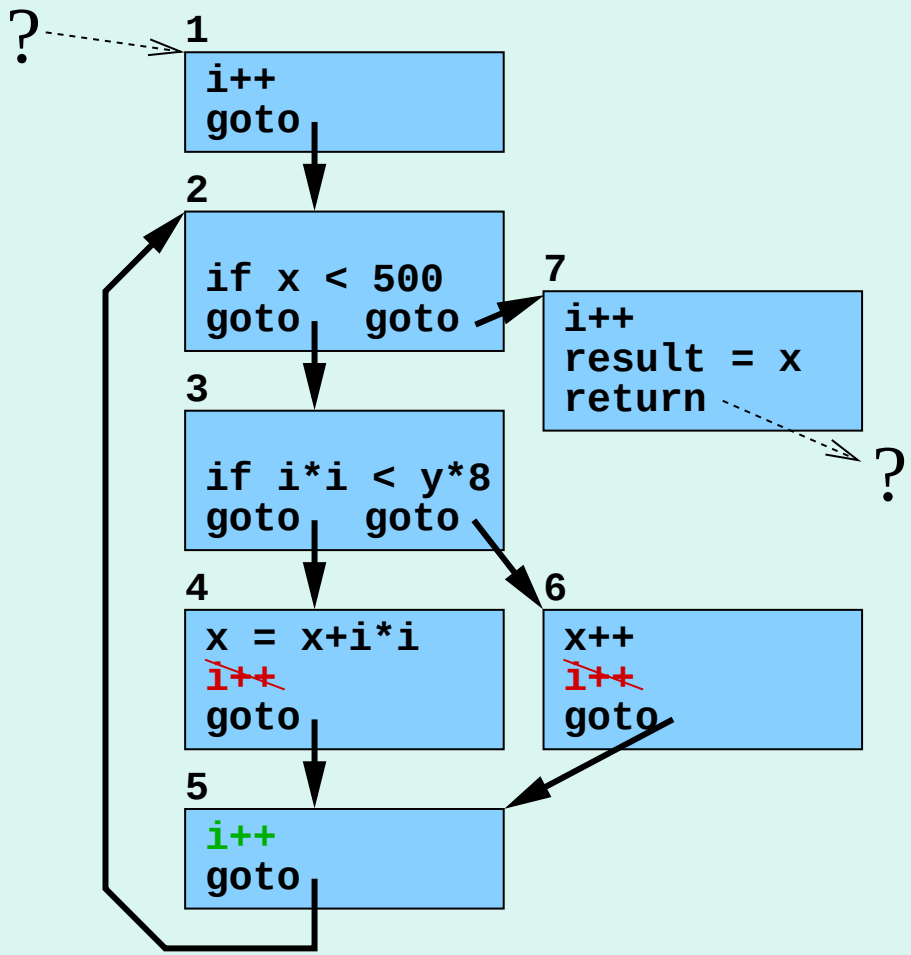
CFG : optimizations (1)

- CFG :



CFG : optimizations (2)

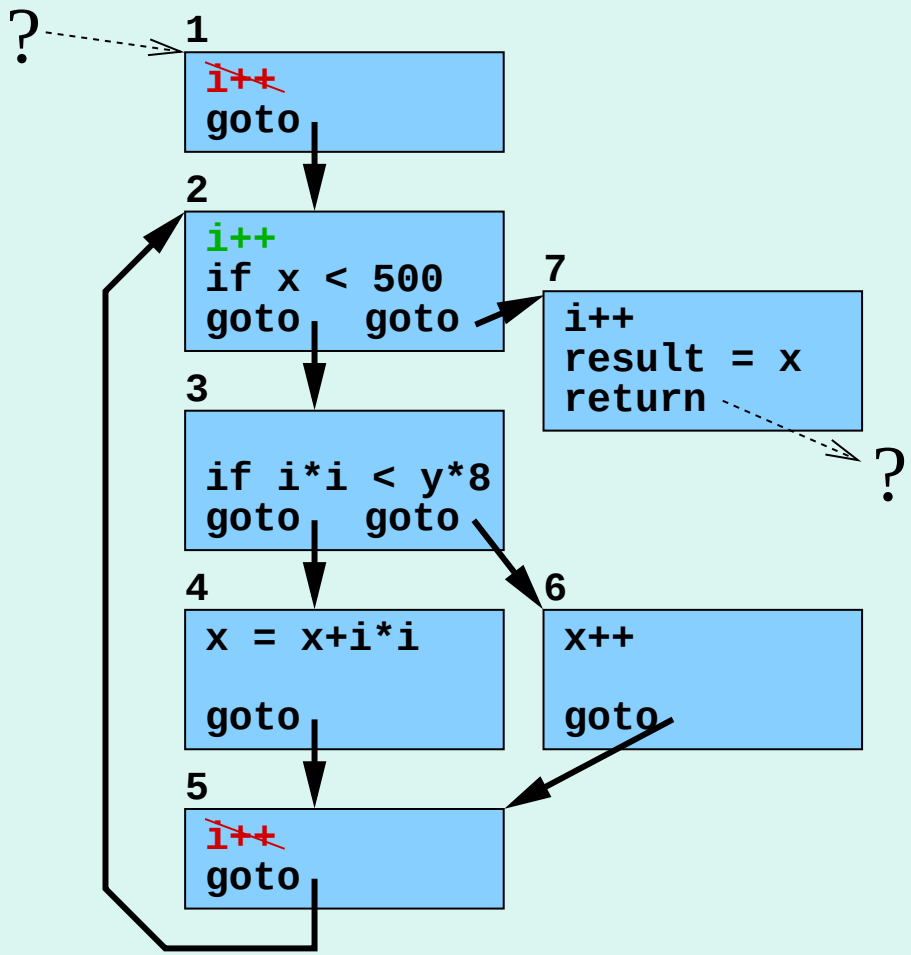
- CFG :



- “code motion” : réduit la taille du code

CFG : optimizations (3)

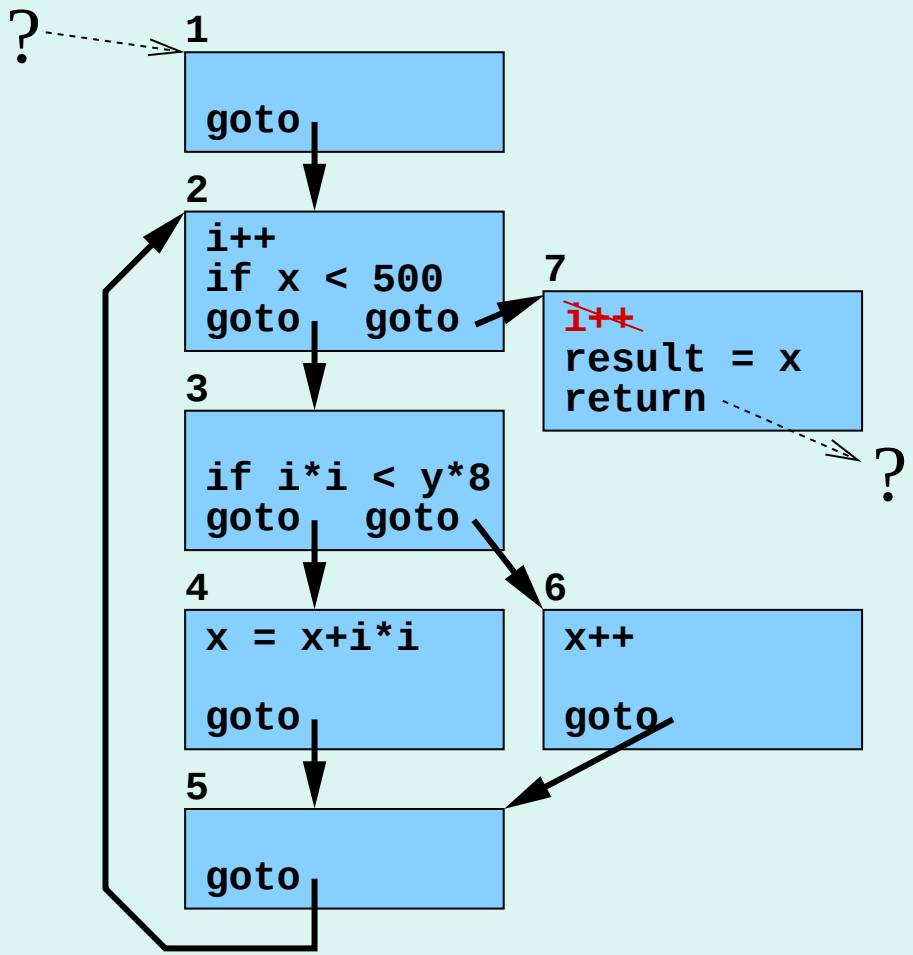
- CFG :



- “code motion” : réduit la taille du code

CFG : optimizations (4)

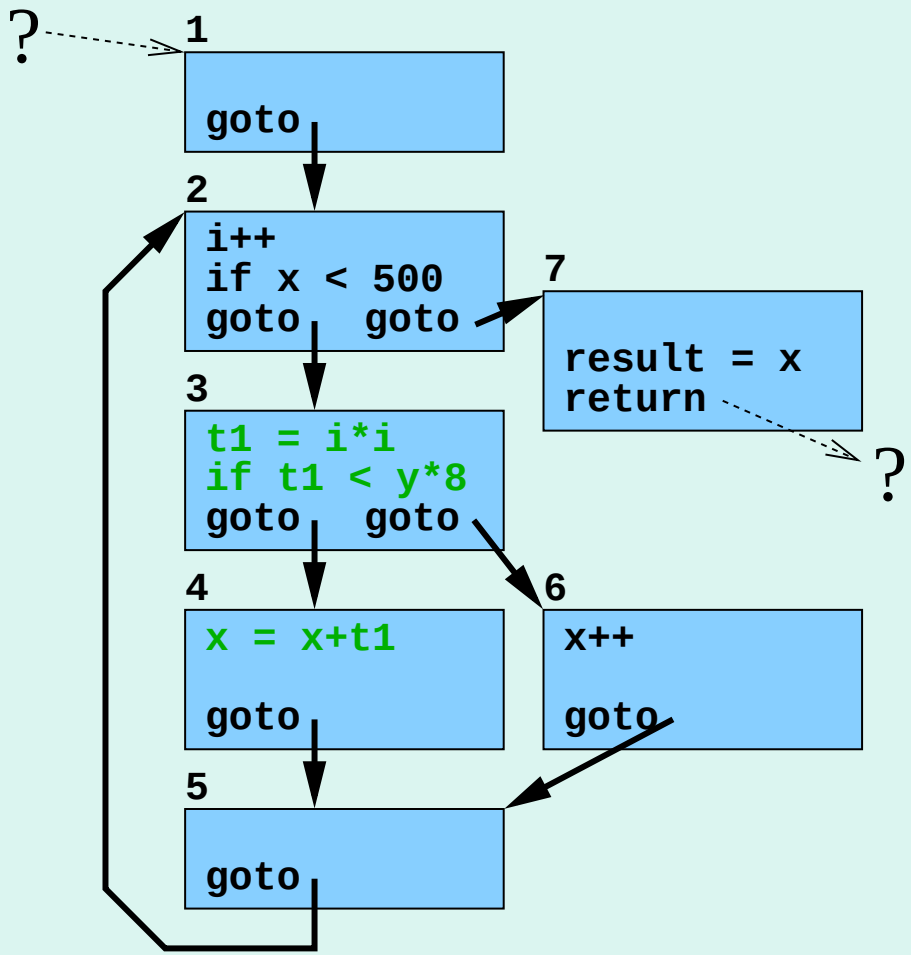
- CFG :



- “dead code elimination” : élimine code inutile

CFG : optimizations (5)

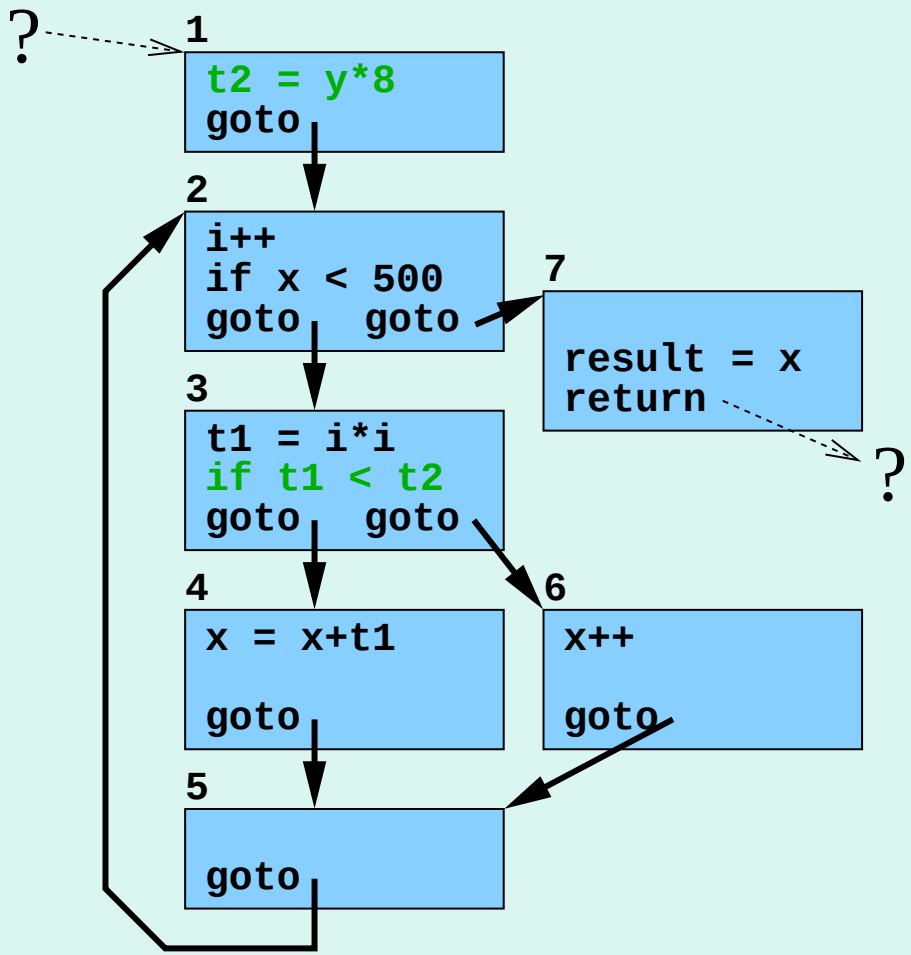
- CFG :



- “common subexpression elimination” : élimine calculs redondants

CFG : optimizations (6)

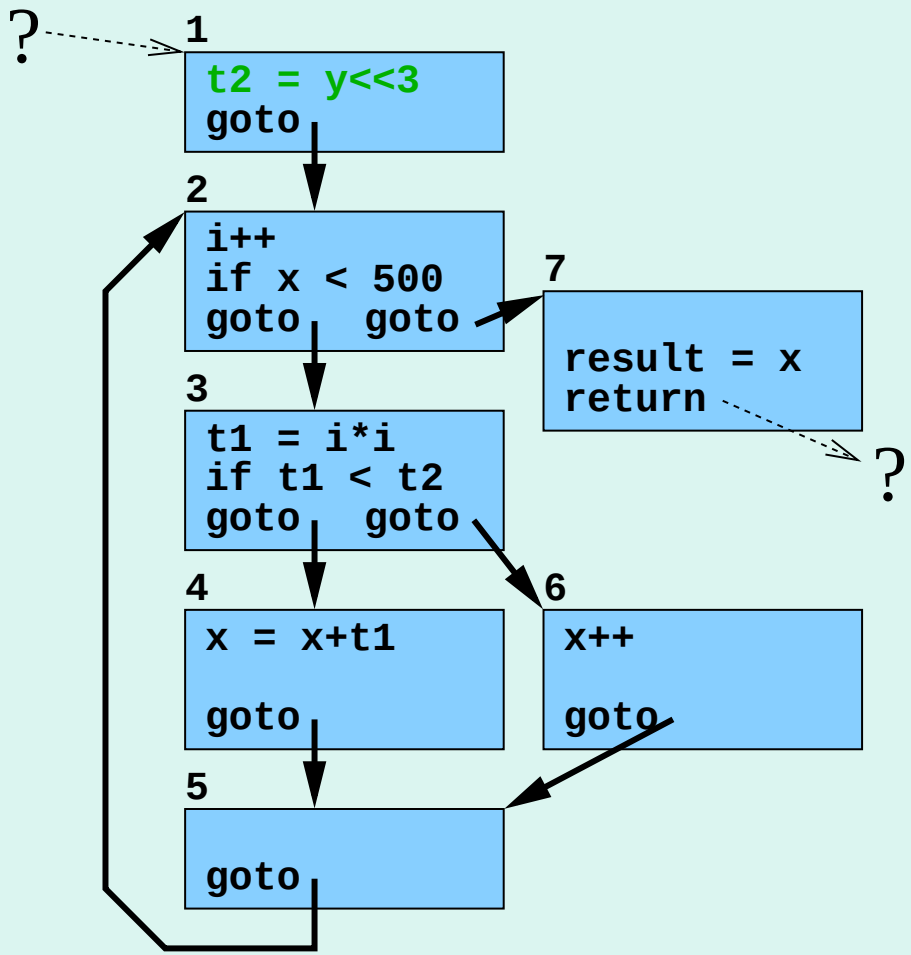
- CFG :



- “loop invariant code motion” : sort les calculs invariants des boucles (normalement spéculatif et doit vérifier que cela ne change pas la sémantique, e.g. exceptions)

CFG : optimizations (7)

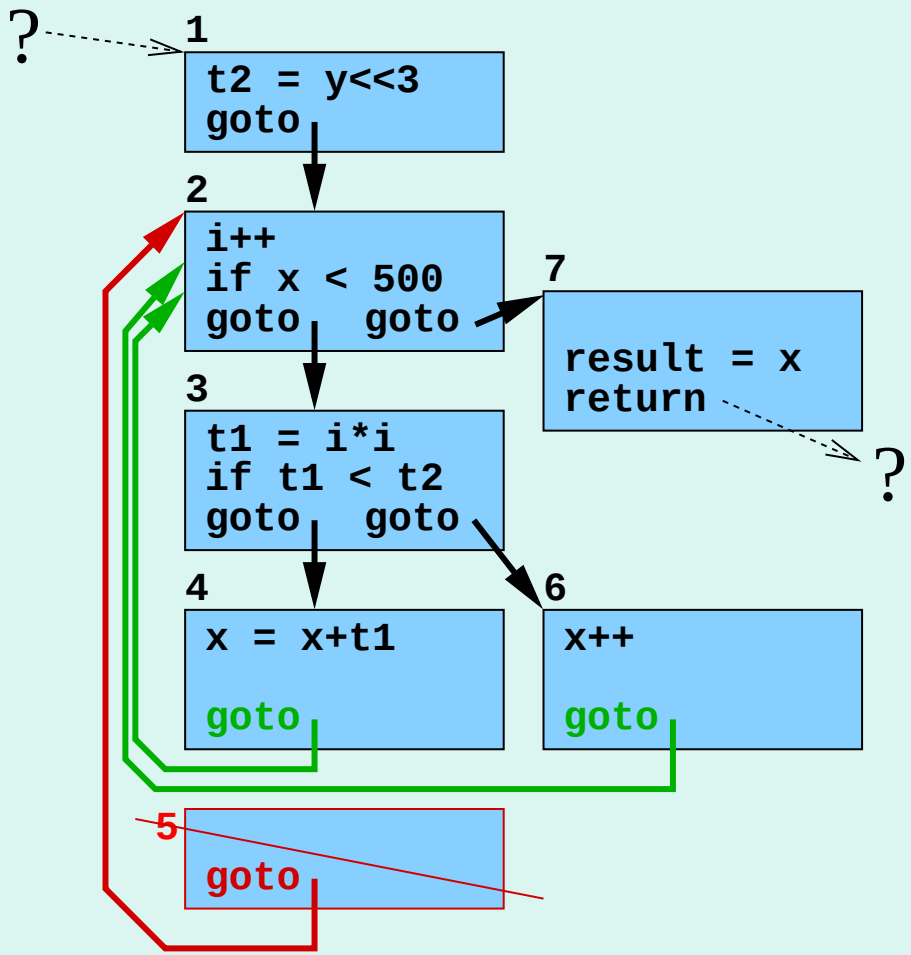
- CFG :



- “strength reduction” : utiliser des calculs équivalents moins chers

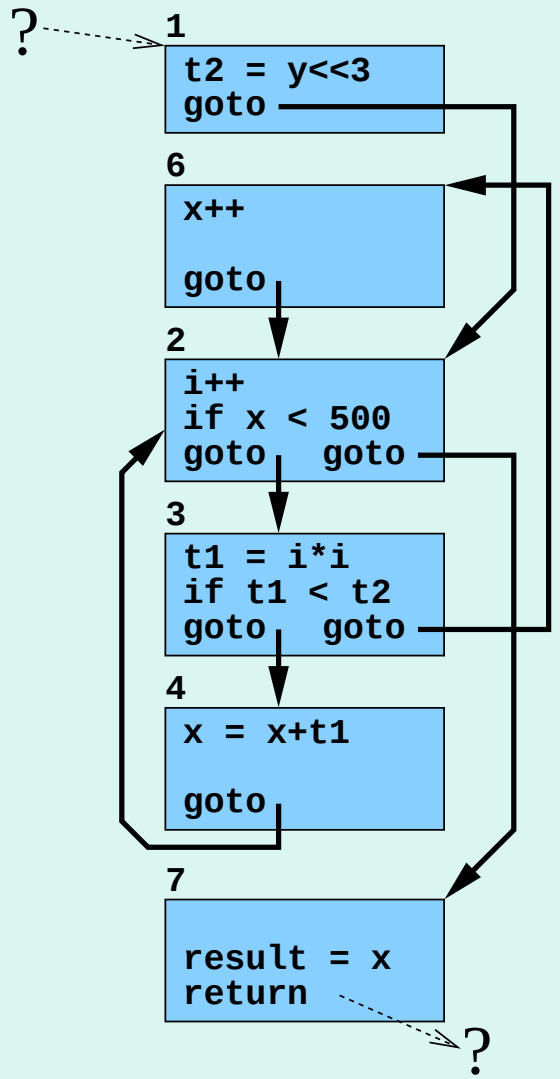
CFG : optimizations (8)

- CFG :



- “branch tensioning” : élimine les branchements vers des branchements

CFG : optimizations (9)



code "assembleur"

```
1:
   t2 = y<<3
   goto 2

6:
   x++

2:
   i++
   if x >= 500 goto 7

   t1 = i*i
   if t1 >= t2 goto 6

   x = x+t1
   goto 2

7:
   result = x
   return
```

- linéarisation du code (“trace scheduling”) : minimise le nombre des branchements

Optimizations (1)

- “Constant propagation”

```
{
  int n = 10;
  x = y*n;
}
```

$\Rightarrow$

```
x = y*10;
```

- Remplacer référence à variable de valeur constante par cette constante

Optimizations (2)

- “Copy propagation”

```
int f( int x ) {
  int y = x;
  return 2*y;
}
```

⇒

```
int f( int x ) {
  return 2*x;
}
```

```
{
  int x = 10;
  int y = x;
  z = n*y;
}
```

⇒

```
z = n*10;
```

- Remplacer référence à variable dont la valeur est une autre variable, par cette autre variable

Optimizations (3)

- “Constant folding”

```
x = x*(2+3);    ⇒    x = x*5;

if (2 < 3) x = 1; else x = 2;    ⇒    x = 1;
```

- Faire à la compilation les opérations sur des opérandes dont la valeur est connue à la compilation

- C’est un cas spécial de l’**évaluation partielle**

Optimizations (4)

- “Function inlining”

```
void f( int x ) {
  printf( "valeur = %d", x );
}

void g( int n ) {
  f( n );
  f( n*n );
}
```

⇒

```
void f( int x ) {
  printf( "valeur = %d", x );
}

void g( int n ) {
  { int x = n; printf( "valeur = %d", x )
  { int x = n*n; printf( "valeur = %d", x
}
```

- Toutes des **variantes de la béta-réduction**...

Optimizations (5)

- “Jump threading”

```
   if (x < y) goto A;
   ...
A: if (x == y) goto C;
B:
```

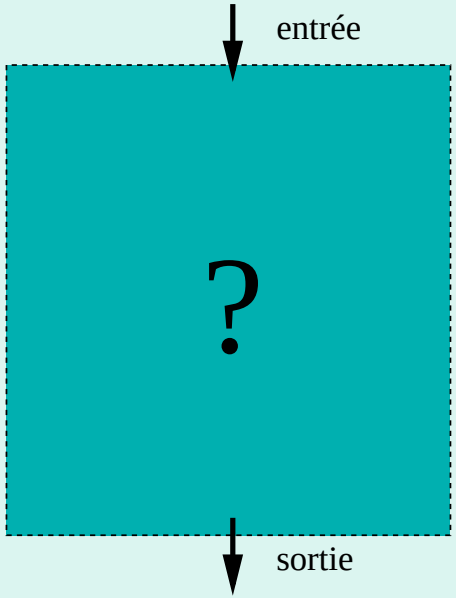
⇒

```
   if (x < y) goto B;
   ...
A: if (x == y) goto C;
B:
```

- Simplifier branchements conditionnels à des branchements conditionnels sur des conditions dépendantes

CFG structurés (1)

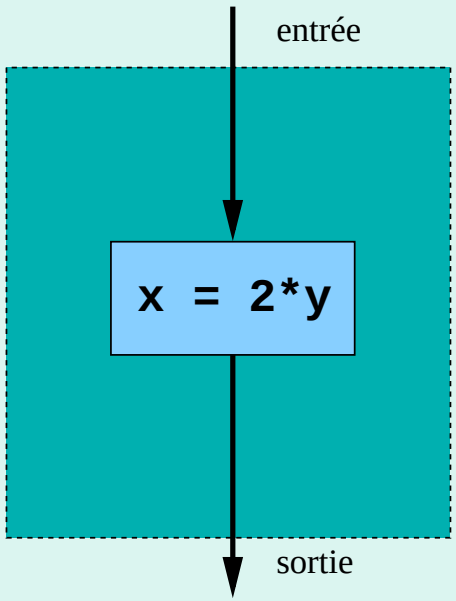
- Un **CFG structuré** est un CFG qui a une entrée et une sortie

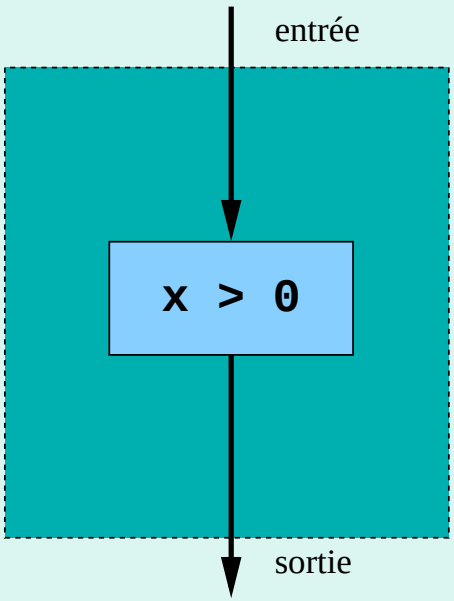


- De plus sa forme suit une grammaire, de sorte qu'il est composé de sous graphes qui sont des CFG structurés

CFG structurés (2)

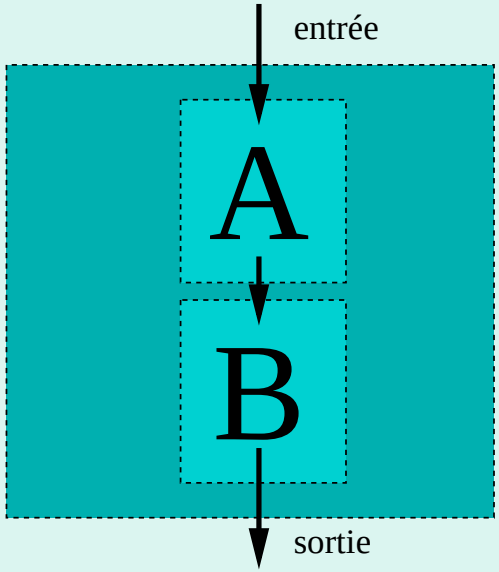
- Les CFG structurés fondamentaux sont des énoncés et expressions simples (sans branchements ni appels de fonction)





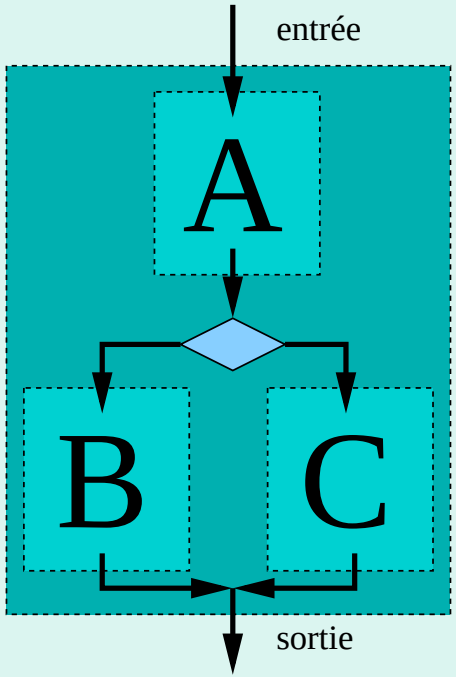
CFG structurés (3)

- A; B;



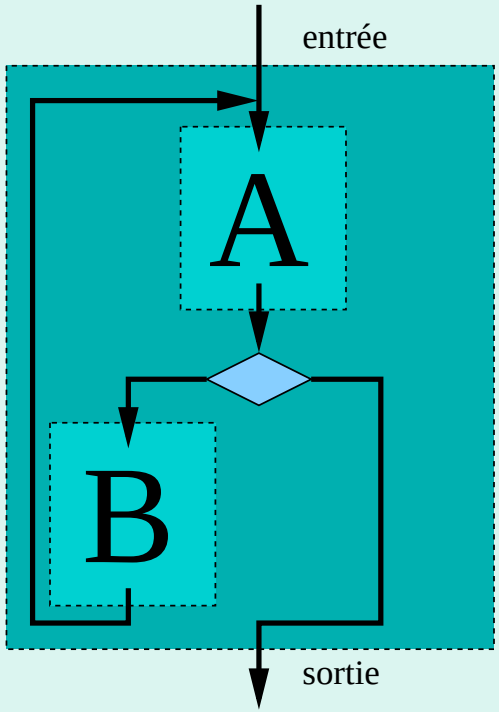
CFG structurés (4)

- if (A) B; else C;



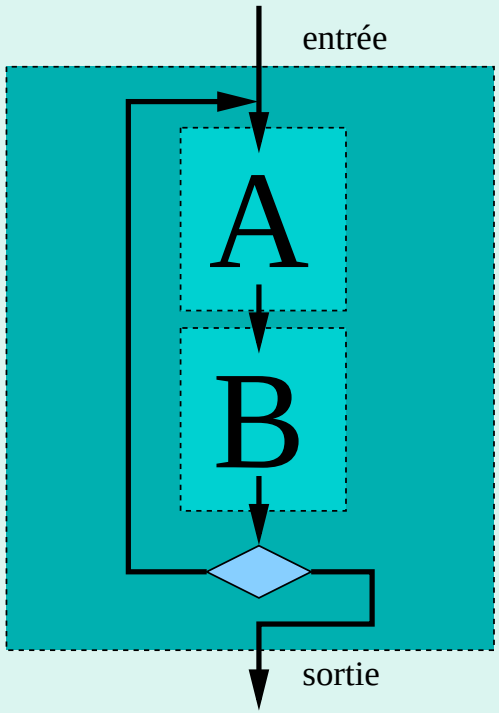
CFG structurés (5)

- while (A) B;



CFG structurés (6)

- do { A; } while(B);

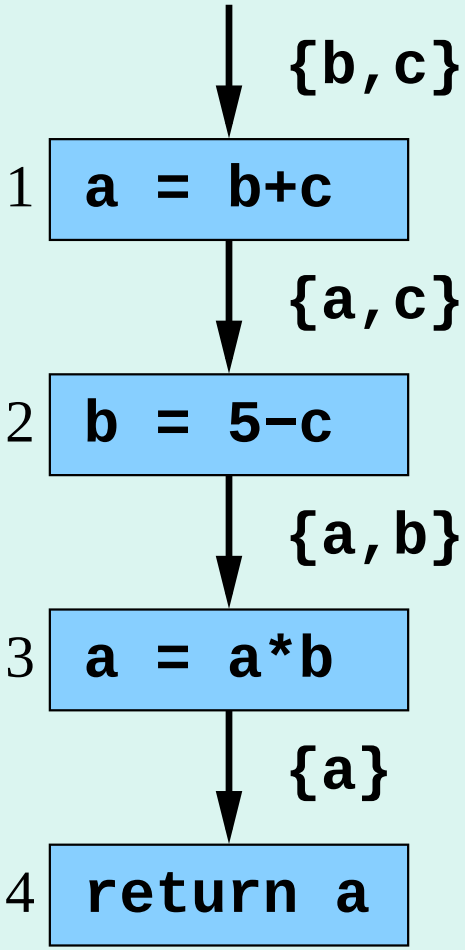


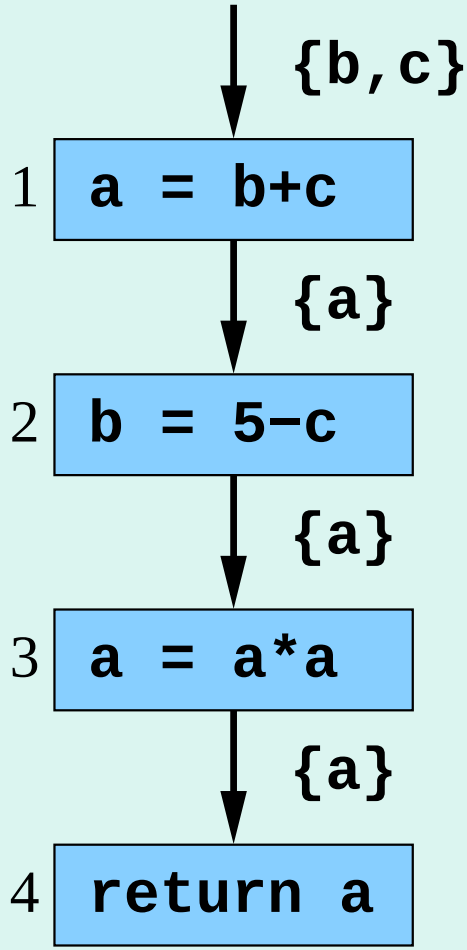
Analyses de Flux de Données

- Ces analyses déterminent des **propriétés** reliées à l'exécution du programme pour chaque instruction ou basic block (**informations de flux**)
  - variables vivantes ("live variables")
  - définitions visibles ("reaching definitions")
  - expressions disponibles ("available expressions")
- Ces propriétés sont souvent représentés par des **ensembles**
- Conceptuellement les propriétés sont attachées aux **arcs du CFG** (i.e. une propriété est satisfaite **entre** deux basic block ou instruction)
- Puisque la propriété dépend du flux de contrôle, on se sert du CFG pour faire ces analyses

Variables vivantes (1)

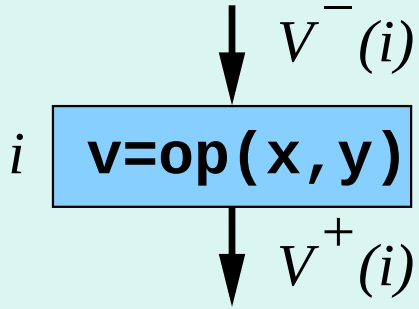
- Une variable est vivante sur un arc du CFG si son contenu présent est **utile pour le reste du programme**



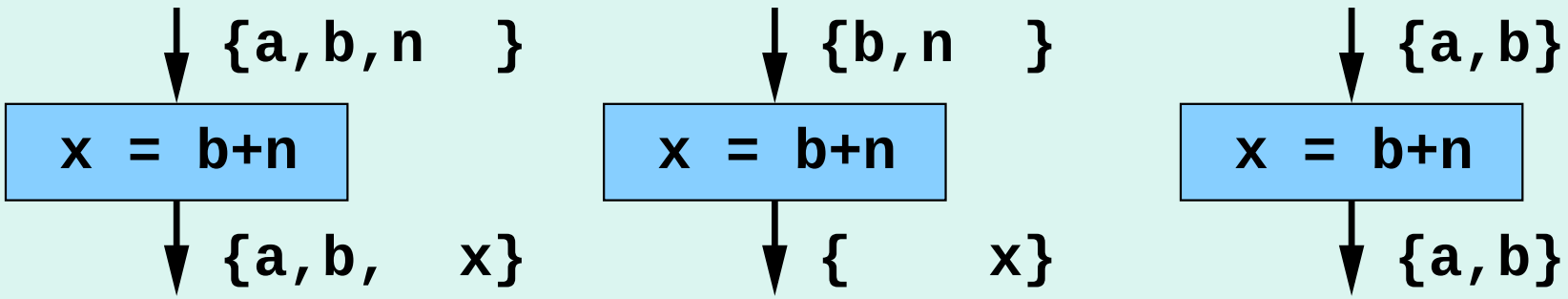


Variables vivantes (2)

- Qu'est-ce qui est contribué par chaque instruction?
- Cas général :



- Autres exemples :



Variables vivantes (3)

- Constatation #1 : $V^-(i)$ dépend de $V^+(i)$
  $\Rightarrow$ **backward flow analysis**

- Constatation #2 : une instruction i:`v=op(x,y)` ne contribue rien lorsque v $\notin V^+$
  $\Rightarrow V^-(i) = V^+(i)$

- Constatation #3 : sinon, $V^-(i) = (V^+(i) - \{\texttt{v}\}) \cup \{\texttt{x},\texttt{y}\}$

- Note : pour bien traiter les branchements de flux

$$V^+(i) = \bigcup_{s \in SUCC(i)} V^-(s)$$

Variables vivantes (4)

- Un CFG donne un système d'équations qui exprime les relations entre les $V^-(i)$ et $V^+(i)$, pour tout i
- La résolution de ce système d'équations donne les variables vivantes à chaque point du programme
- Une façon de résoudre le système d'équations c'est par un **algorithme de point fixe**

Algorithme de point fixe itératif (1)

- Tableau $V_0[i]$ contenant l'ensemble vide, pour tout i
- Tableau $V_x[i]$, $x \geq 1$ est calculé à partir de V_{x-1}
- On remplace dans le système d'équation les $V^-(i)$ par $V_x[i]$ (du côté gauche) et $V_{x-1}[i]$ (du côté droit)
- On itère sur $x \geq 1$ jusqu'à ce que $V_x = V_{x-1}$

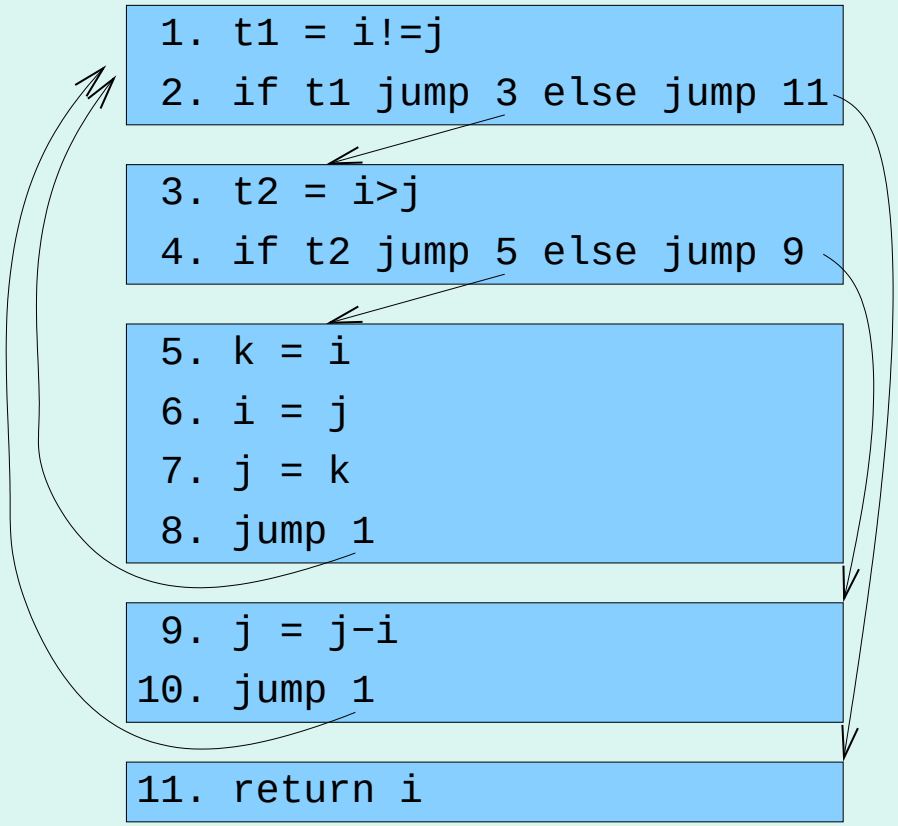
Algorithme de point fixe itératif (2)

- Terminaison?
- Oui car à chaque itération qui n’est pas la dernière, on ajoute au moins un élément aux ensembles
- Il y a un **nombre maximal d’éléments par ensemble**

Exemple : GCD (1)

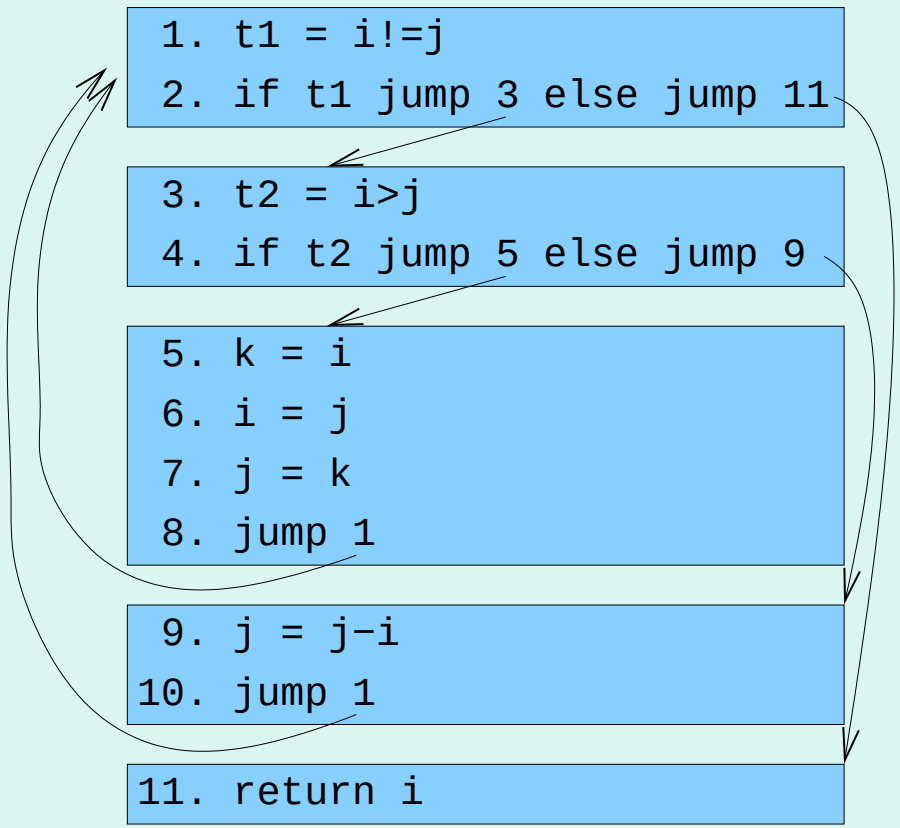
- L'algorithme de GCD en C et son CFG :

```
int gcd(int i, int j)
{
  int k;
  while (i != j)
    if (i > j)
      { k=i; i=j; j=k; }
    else
      j=j-i;
  return i;
}
```



Exemple : GCD (2)

- Trace de l'algorithme de point fixe itératif :



	V0	V1	V2	V3	V4
1. t1 = i!=j	{ }	{i,j }	{i,j }	{i,j }	{i,j }
2. if t1 jump 3 else jump 11	{ }	{ t1}	{i,j,t1}	{i,j,t1}	{i,j,t1}
3. t2 = i>j	{ }	{i,j }	{i,j }	{i,j }	{i,j }
4. if t2 jump 5 else jump 9	{ }	{ t2}	{i,j,t2}	{i,j,t2}	{i,j,t2}
5. k = i	{ }	{i }	{i,j }	{i,j }	{i,j }
6. i = j	{ }	{ j }	{ j,k }	{ j,k }	{ j,k }
7. j = k	{ }	{ k }	{ k }	{i, k }	{i, k }
8. jump 1	{ }	{ }	{i,j }	{i,j }	{i,j }
9. j = j-i	{ }	{i,j }	{i,j }	{i,j }	{i,j }
10. jump 1	{ }	{ }	{i,j }	{i,j }	{i,j }
11. return i	{ }	{i }	{i }	{i }	{i }

- On s'arrête lorsque $x = 4$, car $V_3 = V_4$

Variables vivantes : equation (1)

- Équation de flux pour l'analyse de variables vivantes (sans élimination de code mort)

$$V(i) = (\bigcup_{s \in SUCC(i)} V(s) - KILL(i)) \cup GEN(i)$$

$$KILL(i : x = op(y, z)) = \{x\}$$

$$GEN(i : x = op(y, z)) = \{y, z\}$$

- Pour accélérer l'analyse, on traite chaque basic block comme une seule instruction composée (les ensembles $GEN(i)$ et $KILL(i)$ représentent l'effet cumulé des instructions le composant)

Variables vivantes : equation (2)

- Par ex., si un basic block B contient les 2 instructions :

$$i : x = op_1(y, z)$$
$$j : a = op_2(b, c)$$

alors

$$V(i) = (V(j) - KILL(i)) \cup GEN(i)$$
$$V(j) = (\bigcup_{s \in SUCC(j)} V(s) - KILL(j)) \cup GEN(j)$$

et

$$V(B) = (((\bigcup_{s \in SUCC(B)} V(s) - KILL(j)) \cup GEN(j)) - KILL(i)) \cup GEN(i)$$

- Donc

$$V(B) = (\bigcup_{s \in SUCC(B)} V(s) - KILL(B)) \cup GEN(B)$$

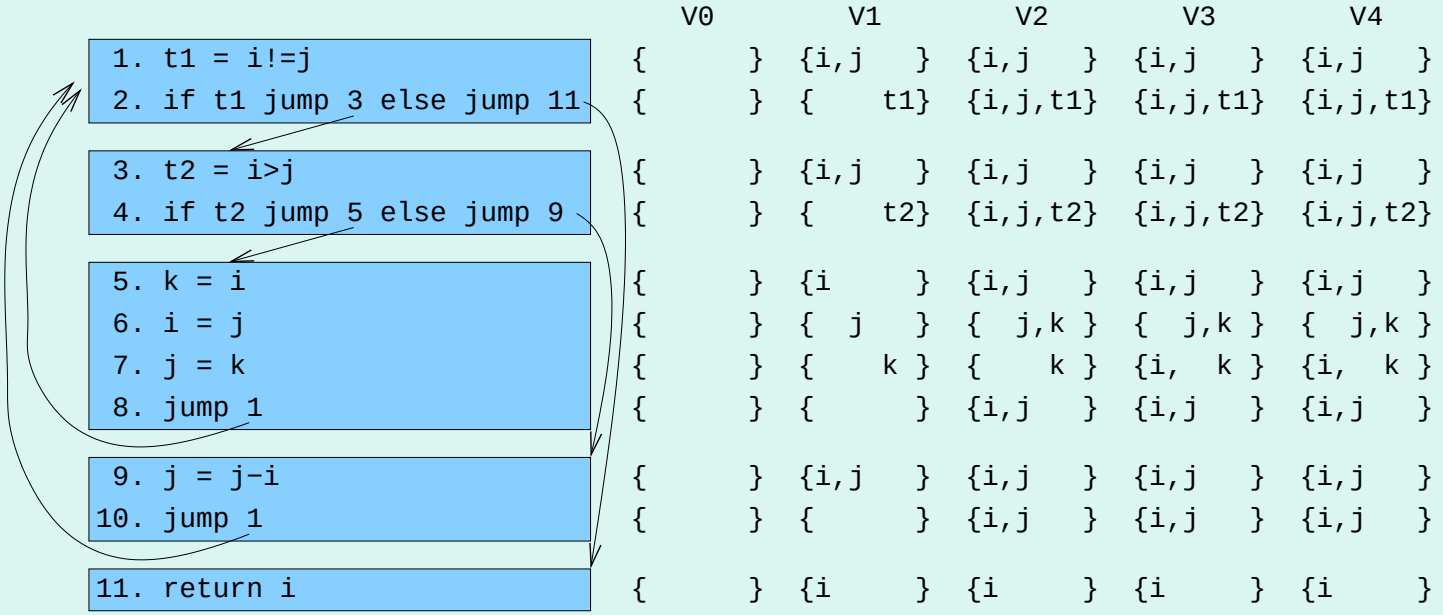
$$KILL(B) = \{a, x\}$$
$$GEN(B) = (\{b, c\} - \{x\}) \cup \{y, z\}$$

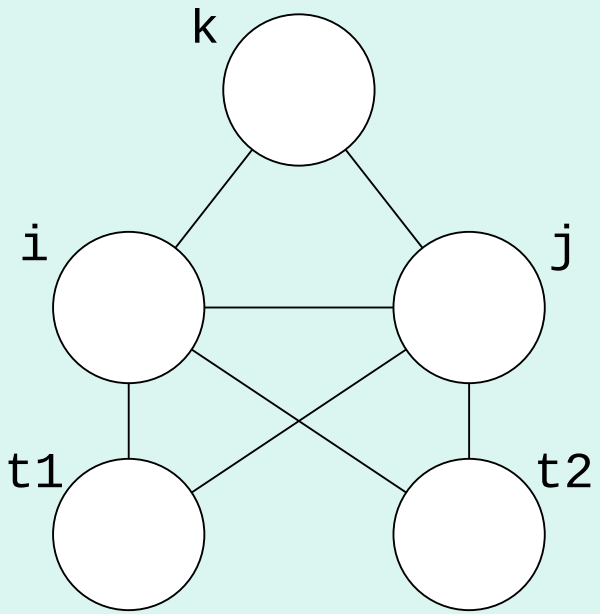
Allocation de registre (1)

- Le résultat de l'analyse des variables vivantes est utile pour l'**allocation de registre**
- L'objectif est d'assigner à chaque variable un "registre" (ou emplacement mémoire) contenant sa valeur, en utilisant le **minimum** de registres
- Deux variables peuvent être assignées au même registre seulement si ces variables ne sont **pas vivantes en même temps**
- Deux variables **interfèrent** si il existe un point d'exécution où elles sont simultanément vivantes
- On construit le **graphe d'interférence** où chaque noeud est une variable et deux variables sont reliées si elles interfèrent

Allocation de registre (2)

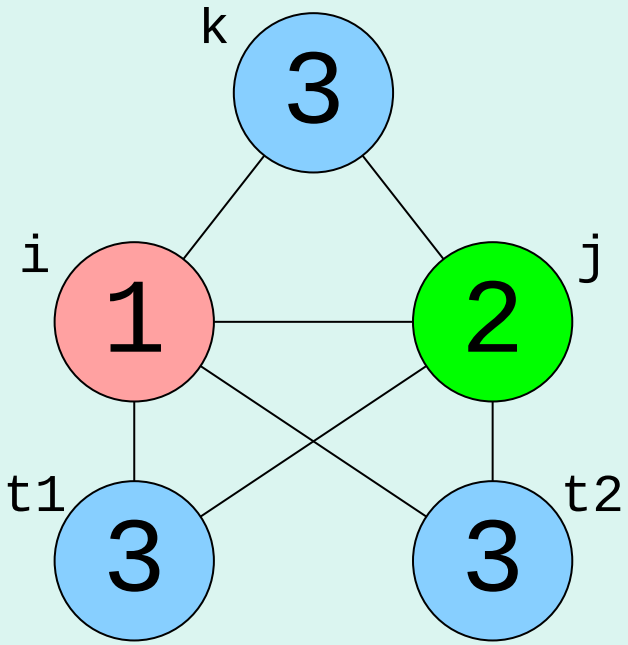
- Graphe d'interférence des variables pour le programme de GCD :





Allocation de registre (3)

- Le **coloriage du graphe d'interférence** donne une solution au problème d'allocation de registre
  - Deux noeuds adjacents ne doivent pas avoir la même couleur
  - Utiliser le nombre minimal de couleurs



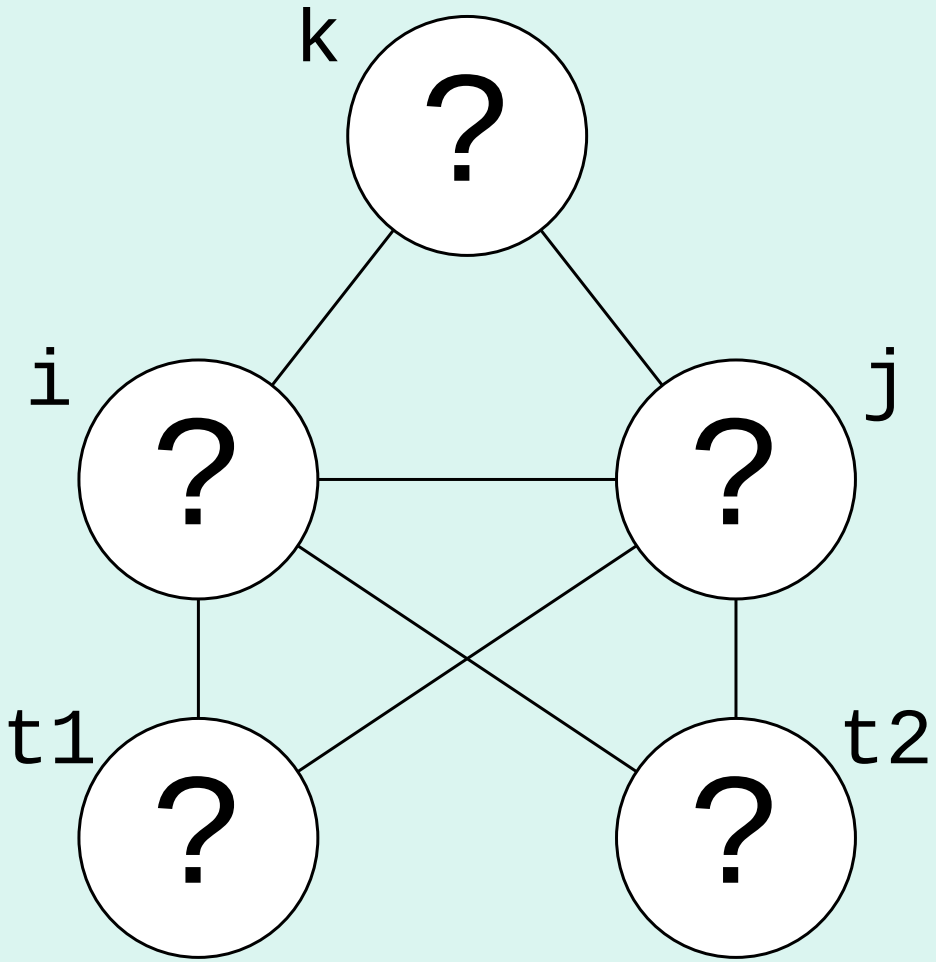
Allocation de registre (4)

- L'algorithme exact de coloriage d'un graphe avec N noeuds est de complexité $O(N^N)$

- L'algorithme polynomial suivant donne des assignations de bonne qualité en pratique (mais non-exact)
  - Si le graphe est non-vide, retirer du graphe le noeud avec le **degré minimum**, et répéter
  - Reconstruire le graphe en insérant tour à tour les noeuds retirés dans l'ordre inverse, en assignant la **plus petite couleur différente des noeuds adjacents**

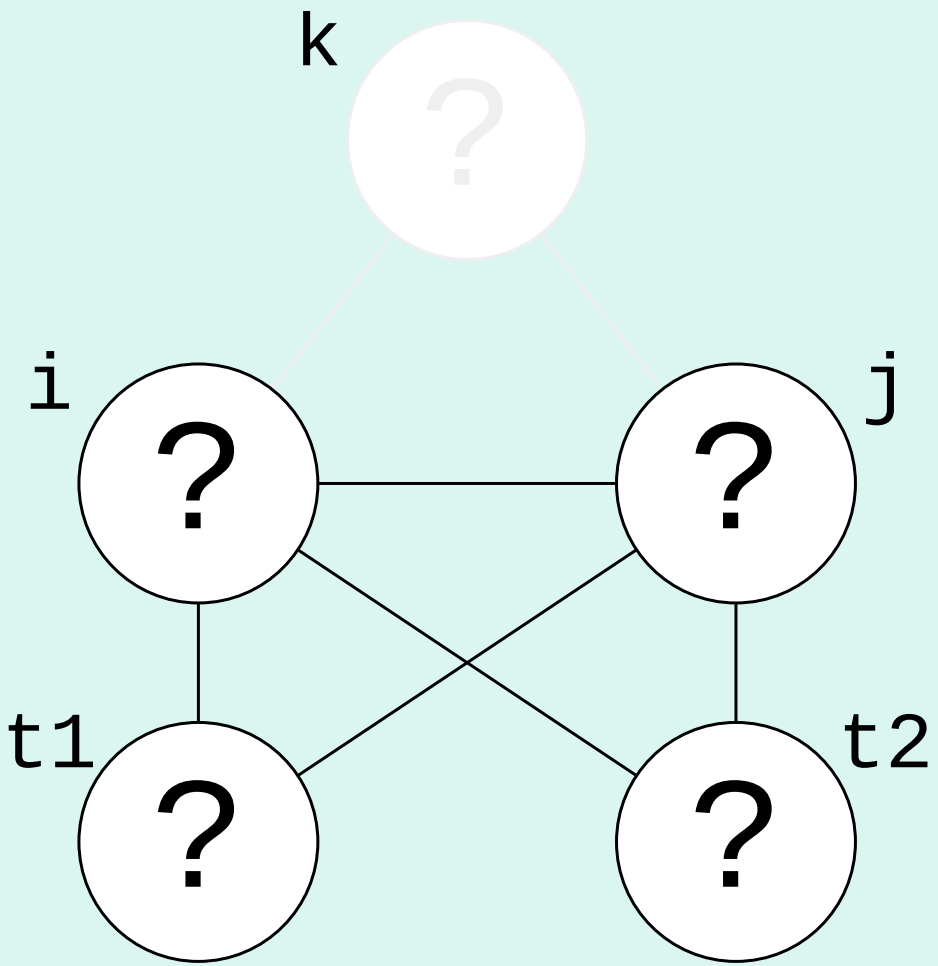
Algorithme de coloriage polynomial (1)

- Graphe d'interférence de départ



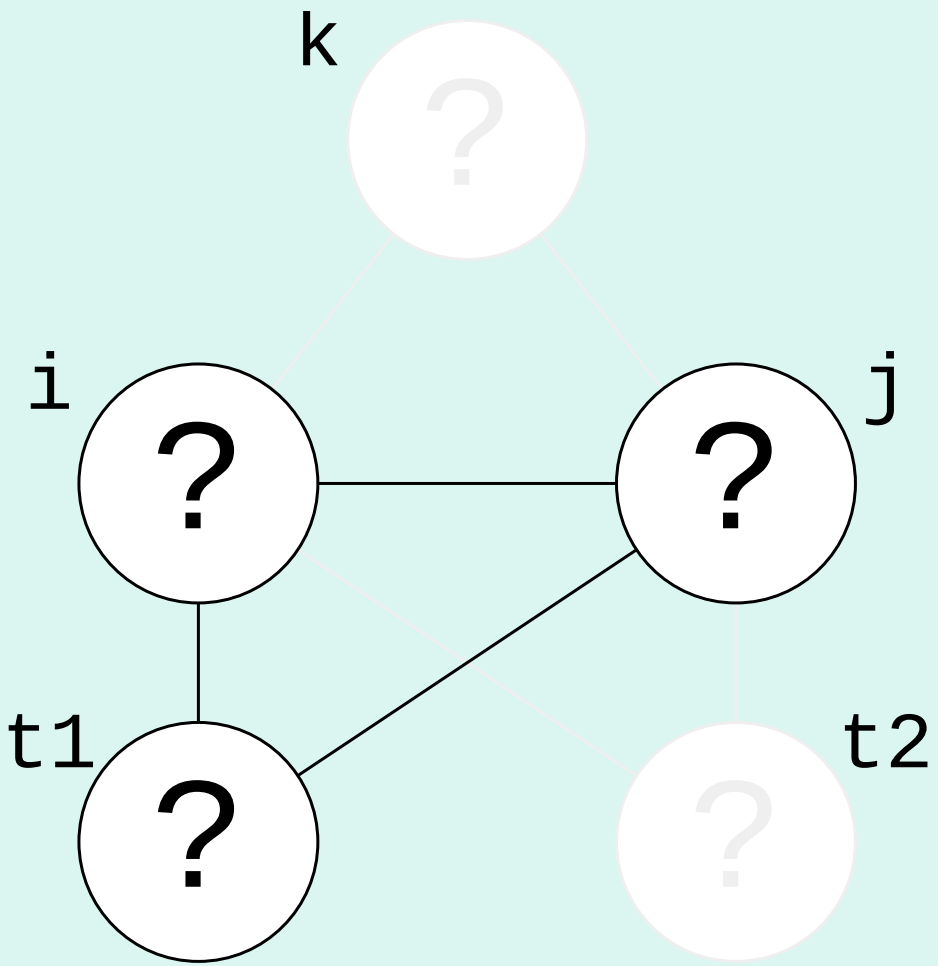
Algorithme de coloriage polynomial (2)

- Graphe d'interférence après avoir retiré k



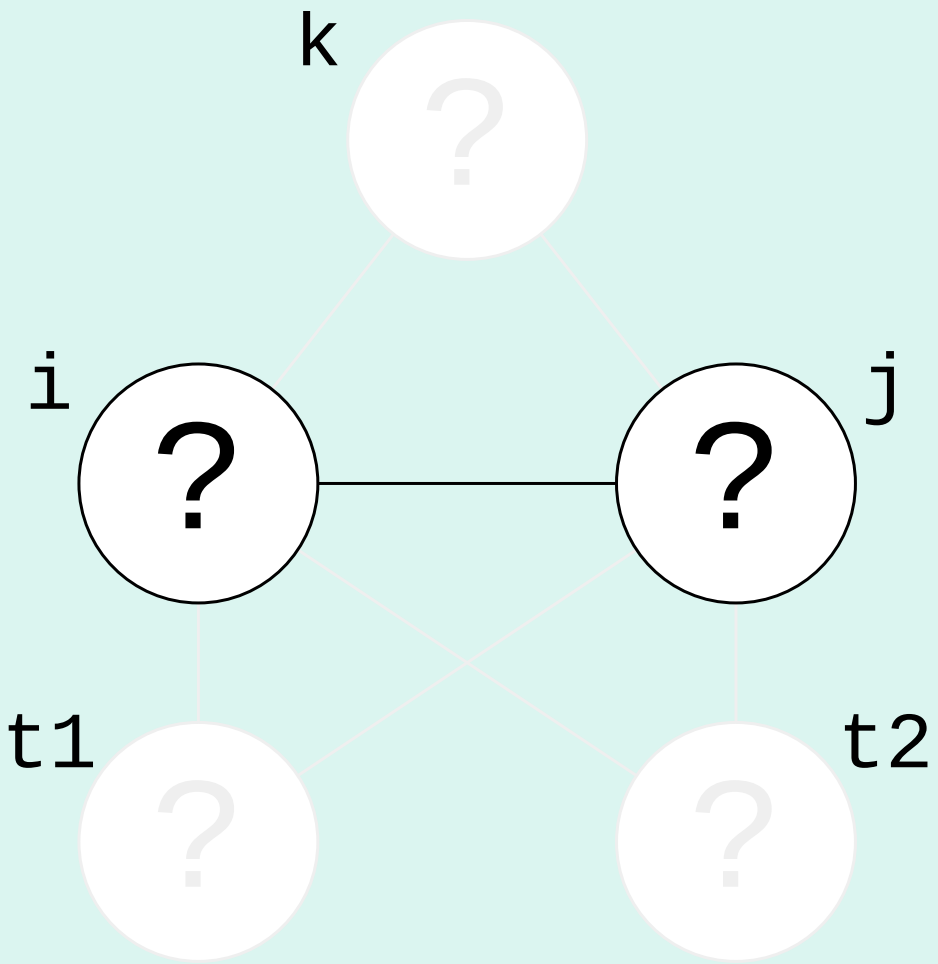
Algorithme de coloriage polynomial (3)

- Graphe d'interférence après avoir retiré `k`, `t2`



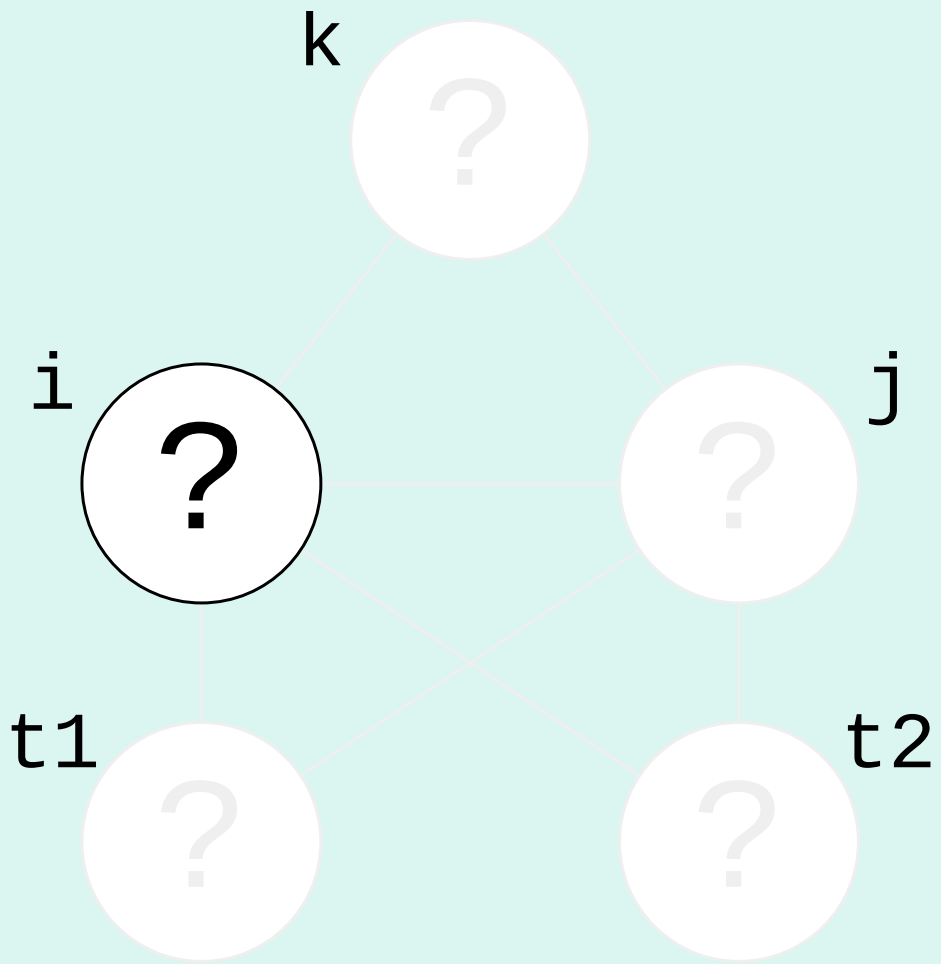
Algorithme de coloriage polynomial (4)

- Graphe d'interférence après avoir retiré `k`, `t2`, `t1`



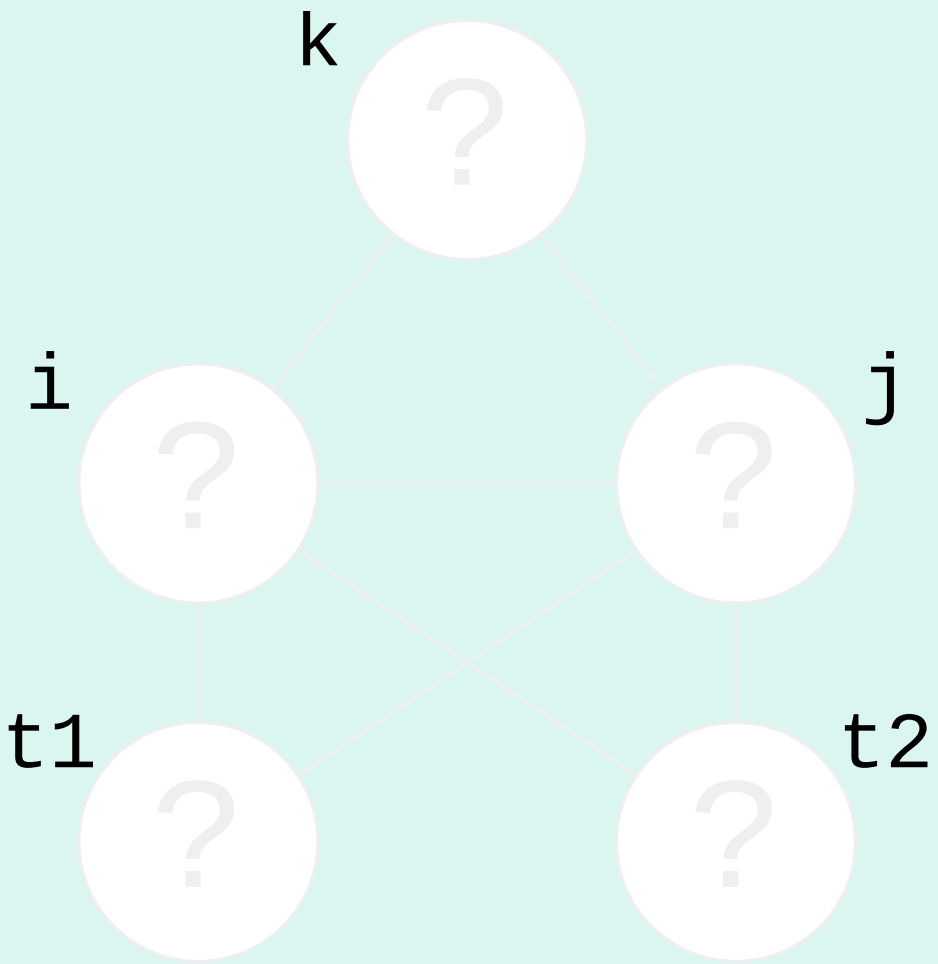
Algorithme de coloriage polynomial (5)

- Graphe d'interférence après avoir retiré `k`, `t2`, `t1`, `j`



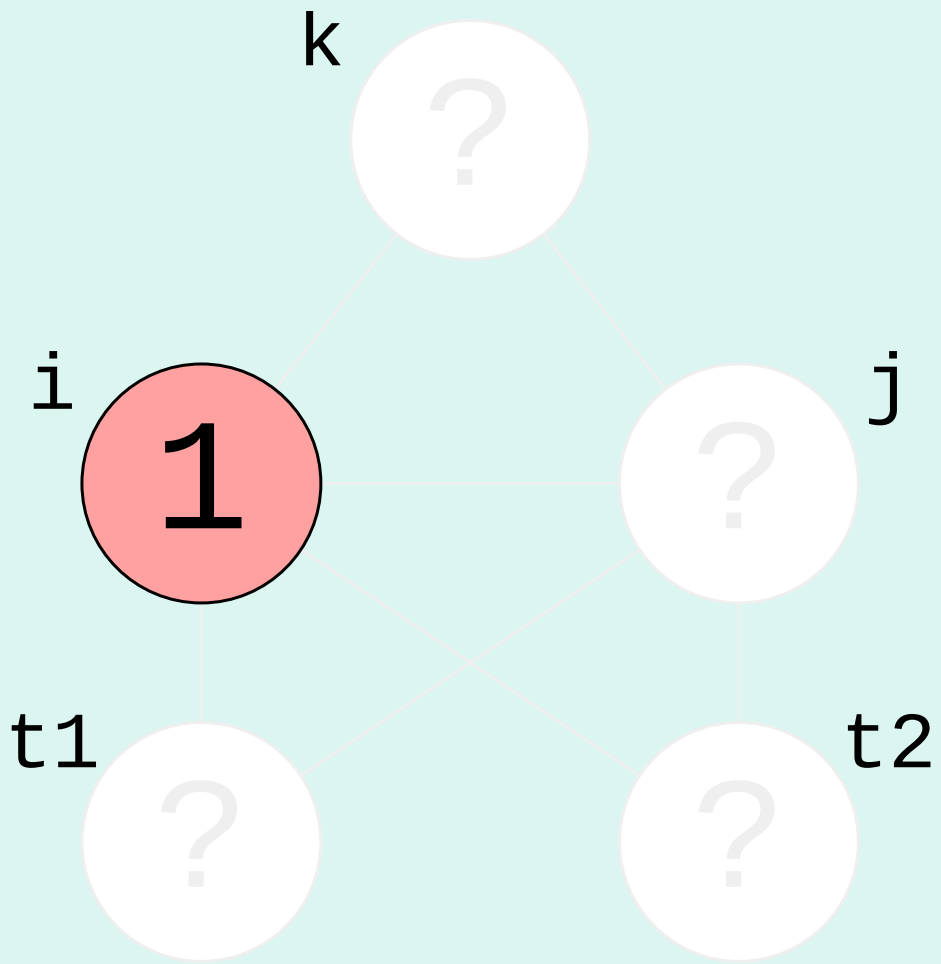
Algorithme de coloriage polynomial (6)

- Graphe d'interférence après avoir retiré `k`, `t2`, `t1`, `j`, `i`



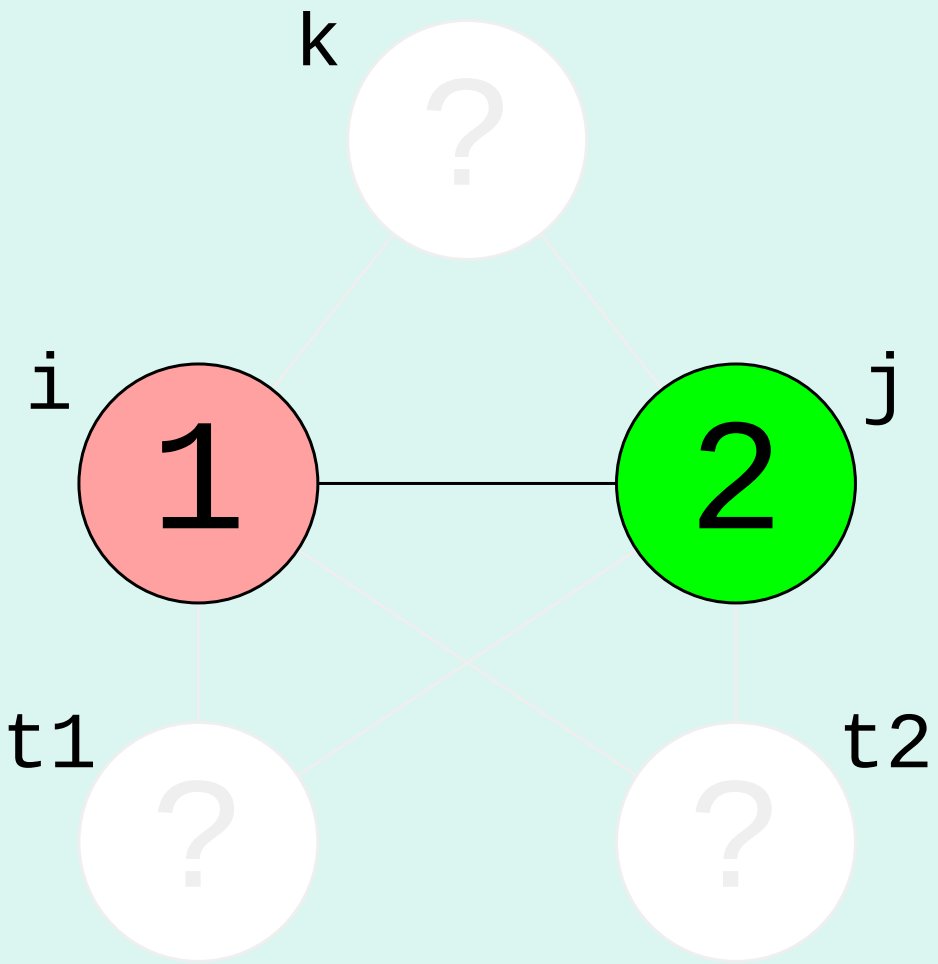
Algorithme de coloriage polynomial (7)

- Graphe d'interférence après avoir remis `i`



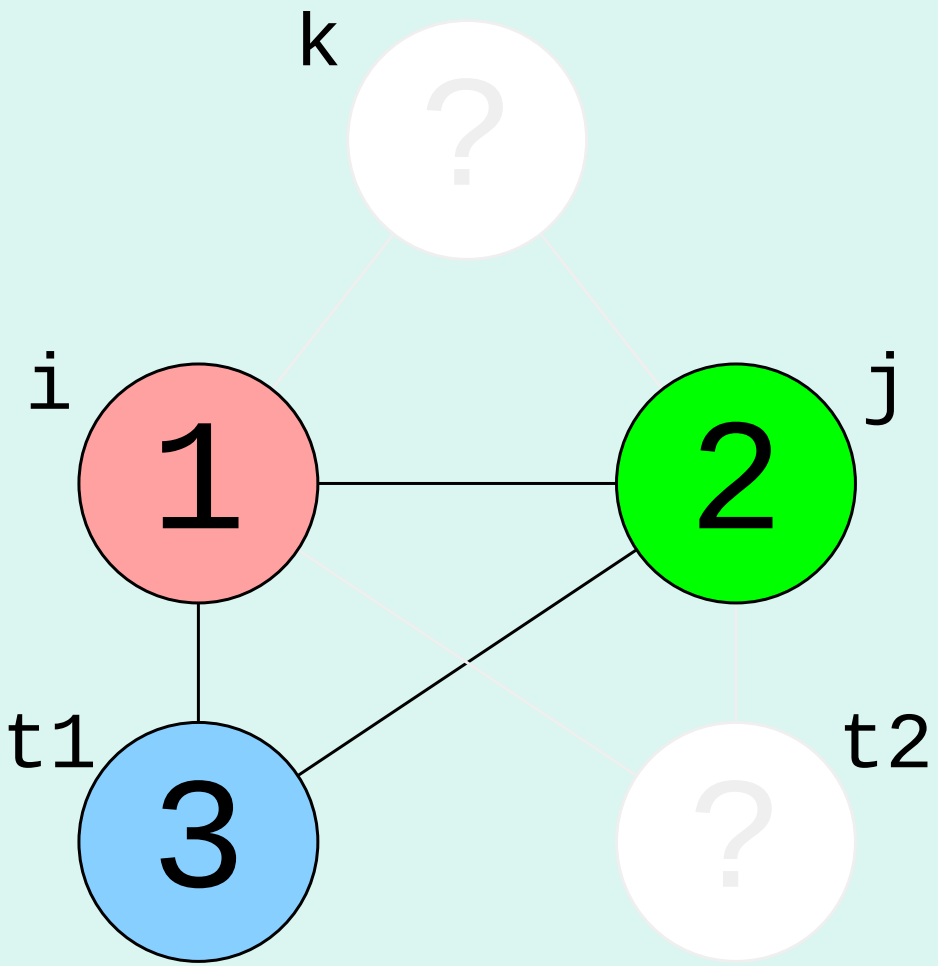
Algorithme de coloriage polynomial (8)

- Graphe d'interférence après avoir remis `i`, `j`



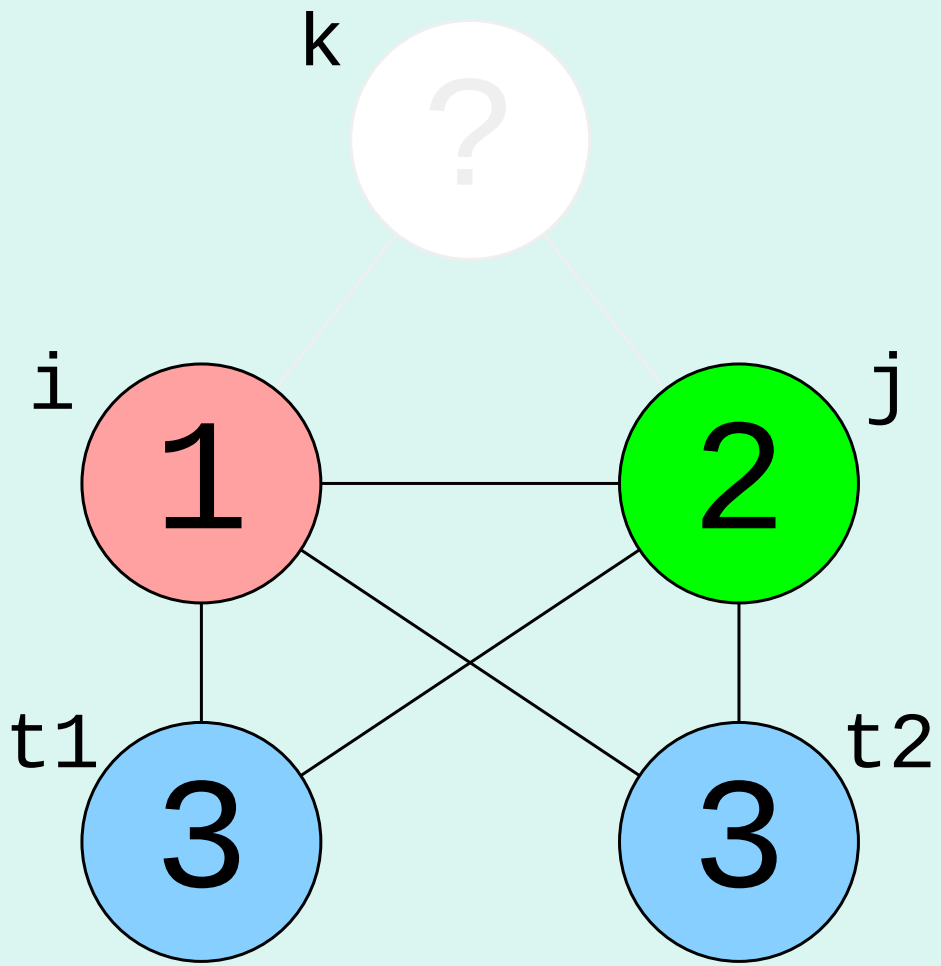
Algorithme de coloriage polynomial (9)

- Graphe d'interférence après avoir remis `i`, `j`, `t1`



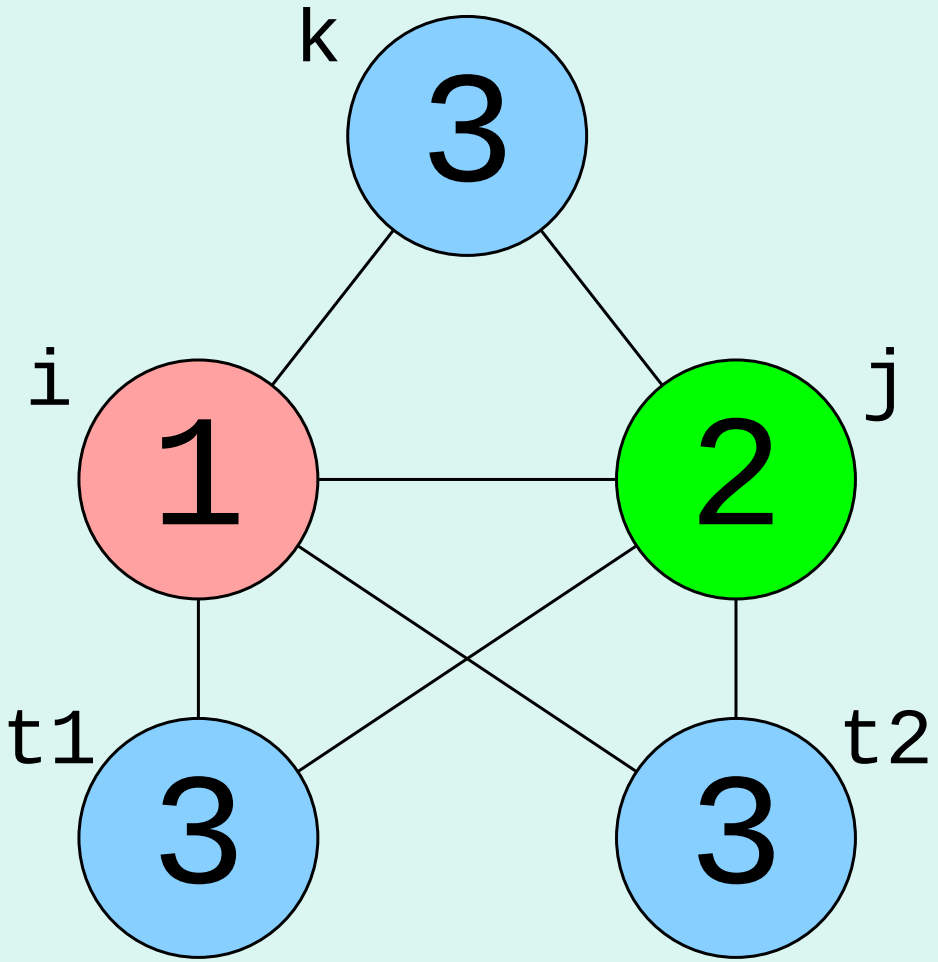
Algorithme de coloriage polynomial (10)

- Graphe d'interférence après avoir remis `i`, `j`, `t1`, `t2`



Algorithme de coloriage polynomial (11)

- Graphe d'interférence après avoir remis `i`, `j`, `t1`, `t2`, `k`



Analyse des expressions disponibles (1)

- Utilile pour effectuer CSE ("**common subexpression elimination**")

- Déf.: une expression $op(x, y)$ est disponible à une instruction i du programme ssi tous les chemins se rendant à i à partir du début du programme ont calculé $op(x, y)$ apres les dernières affectations à x et y

Analyse des expressions disponibles (2)

- Exemple :

```
1. if x=0 jump 2 else jump 4

2. x := b+c
3. jump 6

4. y := b+c
5. jump 6

6. z := x*b          <-- b+c disponible ici
7. b := b+c          <-- b+c disponible ici
8. a := b+c          <-- b+c PAS disponible ici
9. return b+c        <-- b+c disponible ici
```

Analyse des expressions disponibles (3)

- Code transformé après CSE :

```
1. if x=0 jump 2 else jump 4

2. t := b+c;  x := t
3. jump 6

4. t := b+c;  y := t
5. jump 6

6. z := x*b           <-- b+c disponible ici
7. b := t             <-- b+c disponible ici
8. t := b+c; a := t   <-- b+c PAS disponible ici
9. return t           <-- b+c disponible ici
```

Analyse des expressions disponibles (4)

- ⇒ **forward flow analysis**

```
Déf.:  E = ensemble de toutes les expressions du programme
           (ex.: E = {b+c,x*b})

         v
       E = sous-ensemble de E qui lit la variable v
                    x          b
           (ex.: E = {x*b}, E = E)

         -
       Di = ensemble des expressions disponibles juste avant i

         +
       Di = ensemble des expressions disponibles juste après i

Équations de flux:

     +      -                 x
    Di = ( Di U {expr} ) - E      note:  i : x = expr

          -          _          +
    ou   Di =     | |          Dp
             p dans PREC(i)

         -
    => D1 = {}
```

Analyse des expressions disponibles (5)


```
                                  +
1. x := y-z                  D1 = ({}          U {y-z}) - {x+y}

                                  +     +
2. jump 3                    D2 = (D1          U {}   ) - {}

                                  +     +   _   +
3. x := y-z                  D3 = ((D2 |   | D5) U {y-z}) - {x+y}

                                  +     +
4. z := x+y                  D4 = (D3          U {x+y}) - {y-z}

                                  +     +
5. if z=0 jump 6 else jump 3 D5 = (D4          U {}   ) - {}

                                  +     +
6. x := x+y                  D6 = (D5          U {x+y}) - {x+y}

                                  +     +
7. z := y-z;                 D7 = (D6          U {y-z}) - {y-z}

                                  +     +
8. return z                  D8 = (D7          U {}   ) - {}
```

Analyse des expressions disponibles (6)

x:	0	1	2	3
+ D1(x):	{}	{y-z}	{y-z}	{y-z}
+ D2(x):	{}	{}	{y-z}	{y-z}
+ D3(x):	{}	{y-z}	{y-z}	{y-z}
+ D4(x):	{}	{x+y}	{x+y}	{x+y}
+ D5(x):	{}	{}	{x+y}	{x+y}
+ D6(x):	{}	{}	{}	{}
+ D7(x):	{}	{}	{}	{}
+ D8(x):	{}	{}	{}	{}

Analyse des expressions disponibles (7)


```
                                    +
1. x := y-z                    D1 = ({}          U {y-z}) - {x+y}

                                    +     +
2. goto 3                      D2 = (D1          U {}   ) - {}

                                    +     +   _   +
3. x := y-z                    D3 = ((D2 |   | D5) U {y-z}) - {x+y}

                                    +     +
4. y := x+y                    D4 = (D3          U {x+y}) - {y-z,x+y}

                                    +     +
5. if z=0 goto 6 else goto 3   D5 = (D4          U {}   ) - {}

                                    +     +
6. x := x+y                    D6 = (D5          U {x+y}) - {x+y}

                                    +     +
7. z := y-z;                   D7 = (D6          U {y-z}) - {y-z}

                                    +     +
8. return z                    D8 = (D7          U {}   ) - {}
```

Analyse des expressions disponibles (8)

x:	0	1	2	3
+ D1(x):	{}	{y-z}	{y-z}	{y-z}
+ D2(x):	{}	{}	{y-z}	{y-z}
+ D3(x):	{}	{y-z}	{y-z}	{y-z}
+ D4(x):	{}	{}	{}	{}
+ D5(x):	{}	{}	{}	{}
+ D6(x):	{}	{}	{}	{}
+ D7(x):	{}	{}	{}	{}
+ D8(x):	{}	{}	{}	{}