Introduction à la compilation

Nicolas Bedon

Cours de L3
Université de Rouen

https://dpt-info-sciences.univ-rouen.fr/~bedonnic/ puis Enseignement/Compilation

Références

- Compilateurs : techniques, principes & outils
  A. Aho, M. Lam, R. Sethi et J. Ullman
- Flex & Bison : text processing tools
  J. Levine
- Manuels de Flex et Bison
- Cours de C
- Cours de théorie des langages
- Des notions de programmation fonctionnelle et de programmation objet sont bienvenues

Modalités

- 20h CM, 12h TD, 18h TP
- Contrôle :
  - Examen terminal E_1
  - Un partiel Pa et un projet Pr
  - Le projet est obligatoire : il compte pour un tiers de la note finale. Sa note Pr est définitive.
  - Note de session 1 = 2/3*Ecrit + Pr/3
    - écrit = $(E_1+Pa)/2$
  - Examen terminal E_2 de seconde session
  - Note de session 2 : 2/3 E_2 + Pr/3
    - Les notes E_1 et Pa ne sont plus prises en compte
    - La note Pr de projet reste (elle ne se rattrape pas!)

Sur la nécessité de traduire

```
.section        .rodata
  .LC0:
        .string "Bonjour !"
        .text
  .globl main
        .type   main, @function
  main:
  .LFB2:
        pushq   %rbp
  .LCFI0:
        movq    %rsp, %rbp
  .LCFI1:
        subq    $16, %rsp
  .LCFI2:
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        ret
```

- Proche du langage du processeur
- Impossible pour un être humain non informaticien
- Difficile même pour un informaticien

Principes élémentaires:

- registres
- adresses
- routines

Sur la nécessité de traduire

```
.section        .rodata
  .LC0:
        .string "Bonjour !"
        .text
  .globl main
        .type   main, @function
  main:
  .LFB2:
        pushq   %rbp
  .LCFI0:
        movq    %rsp, %rbp
  .LCFI1:
        subq    $16, %rsp
  .LCFI2:
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        ret
```

```
int main(int argc, char *argv[]) {
   printf(« Bonjour ! »);
   return EXIT_SUCCESS;
}
```

Notions élémentaires (C):
- types
- variables
- fonctions
- structures de contrôles évoluées

Mais aussi (C++, Java)
- classes
- objets
- méthodes

Ou encore (ML)
- pattern-matching

Ou encore (Prolog)
- résolution de contraintes

...

De la nécessité de traduire

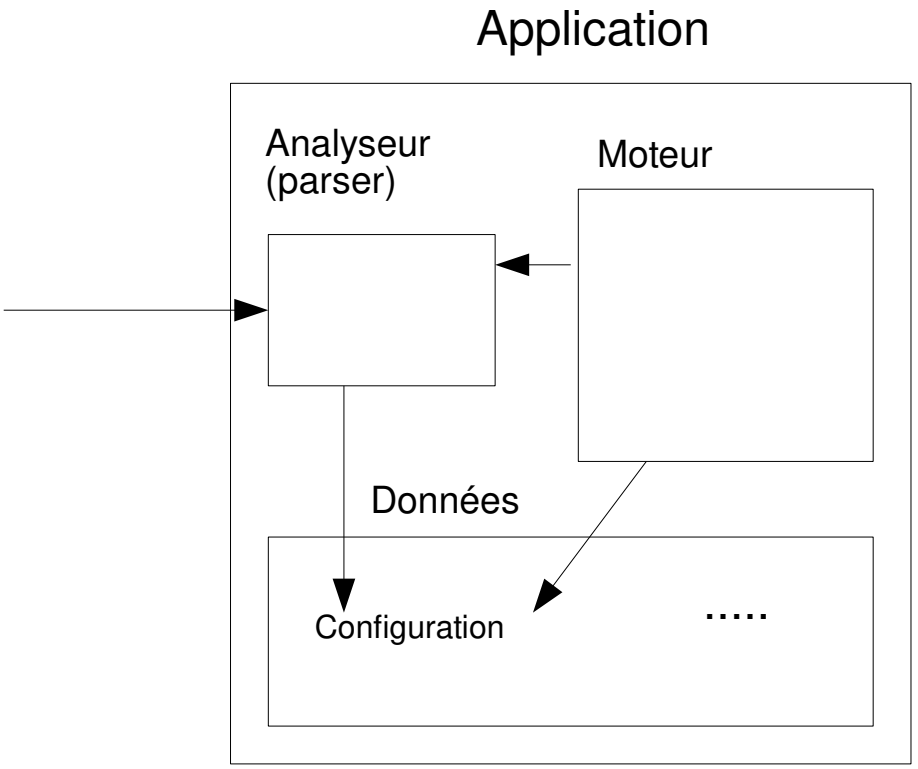
Intérêt des langages de haut niveau:

- meilleure facilité d'expression
- séparation des concepts
  - un concept -> un cadre d'utilisation
  - un cadre d'utilisation -> un moyen formel de contrôler les erreurs
  - plus de contrôles = moins d'erreurs
- meilleure lisibilité
  - développement en équipe facilité
  - maintenance facilitée

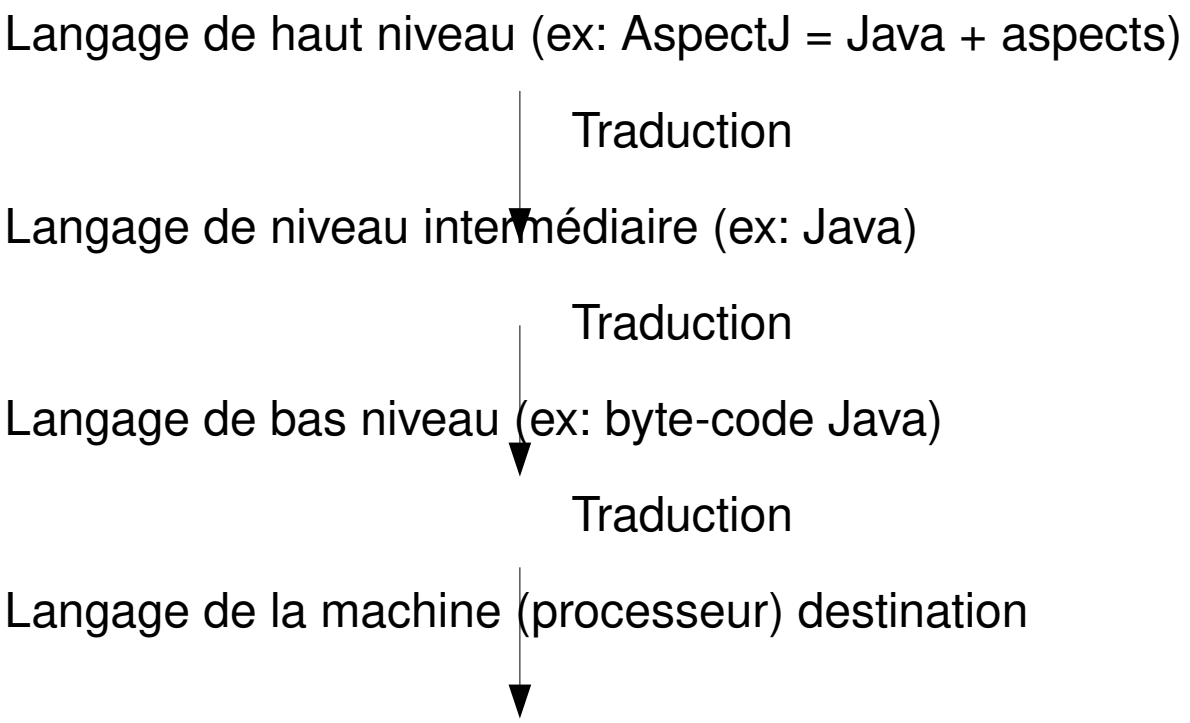
Développement facilité = coûts réduits

De la nécessité de traduire

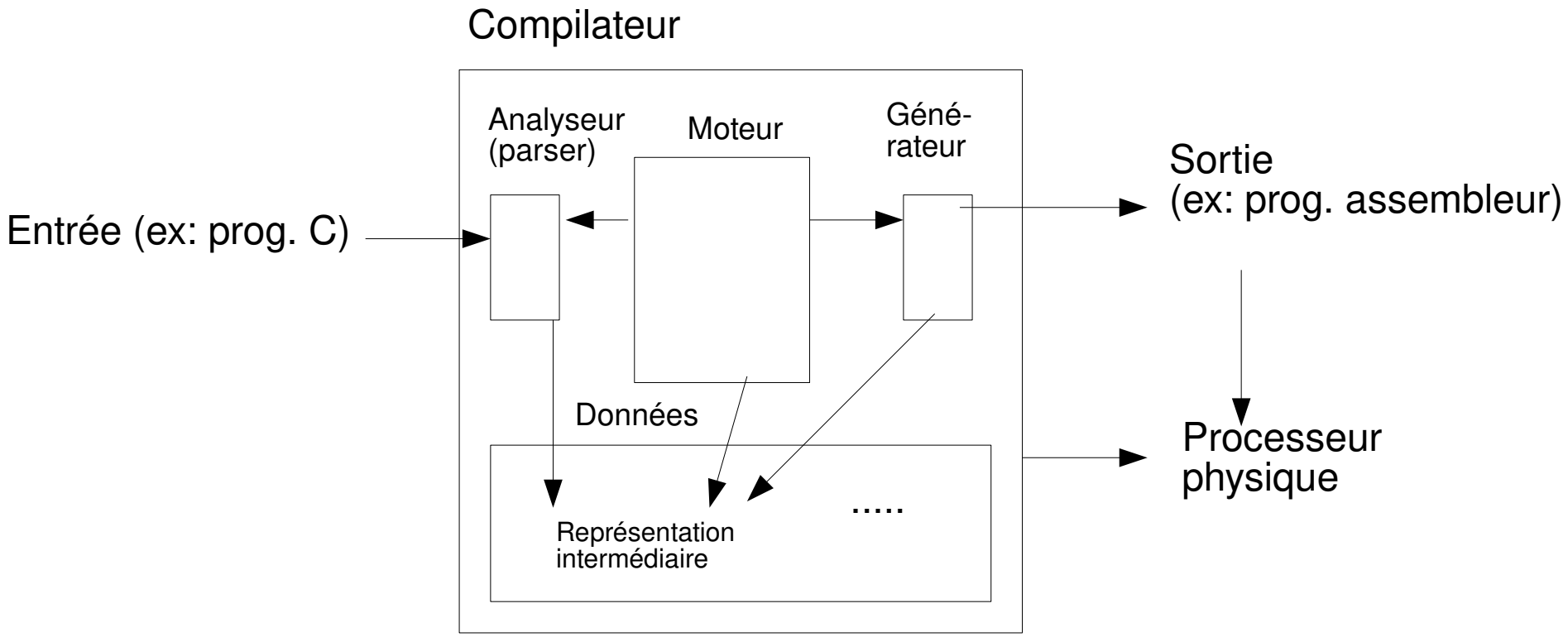
```
<?xml version="1.0"?>

<configuration>

   <menu id="MainMenu" columns="2">
   <entry
         index = "1"
         on    = "../../resources/mmbox/icon/dvd.2.png"
         off   = "../../resources/mmbox/icon/dvd.1.png"
         x     = "86"
         y     = "50">DVDPlayer</entry>
    </menu>

  <menu id="DVDPlayer">
   <entry
         index = "1"
         on    = "../../resources/mmbox/icon/audiocd.2.png"
         off   = "../../resources/mmbox/icon/audiocd.1.png"
         x     = "86"
         y     = "50">menuid_01</entry>
  </menu>

</configuration>
```



Programmation et traductions en chaîne



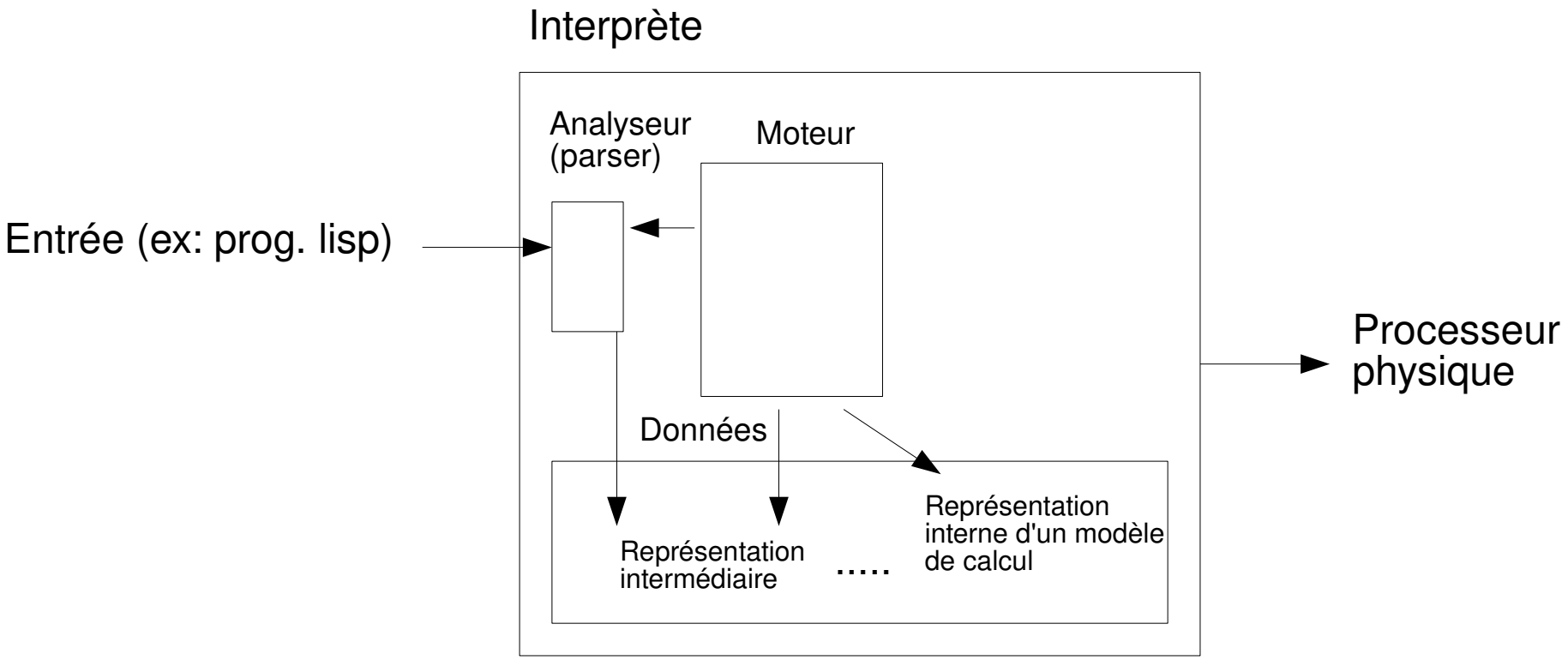
Catégories de traducteurs en programmation: compilateur



- Analyse de l'entrée
- Représentation intermédiaire
- Génération sortie
- Exécution déléguée au processeur physique

Exemples:
- gcc
- javac
- as

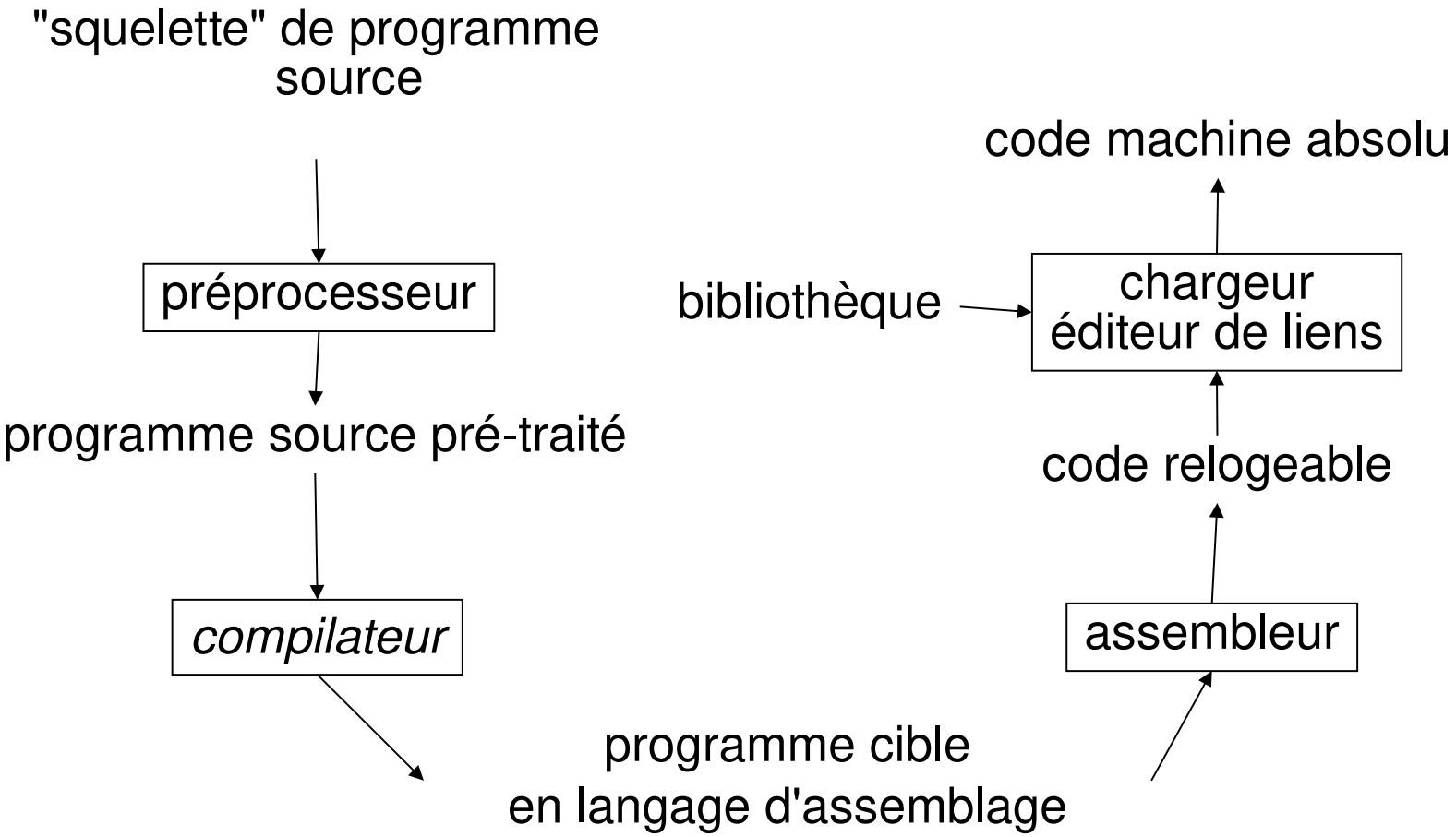
Catégories de traducteurs en programmation: interprète



- Analyse de l'entrée
- Représentation intermédiaire
- Le moteur simule l'action des instructions dans une implantation d'un modèle de calcul

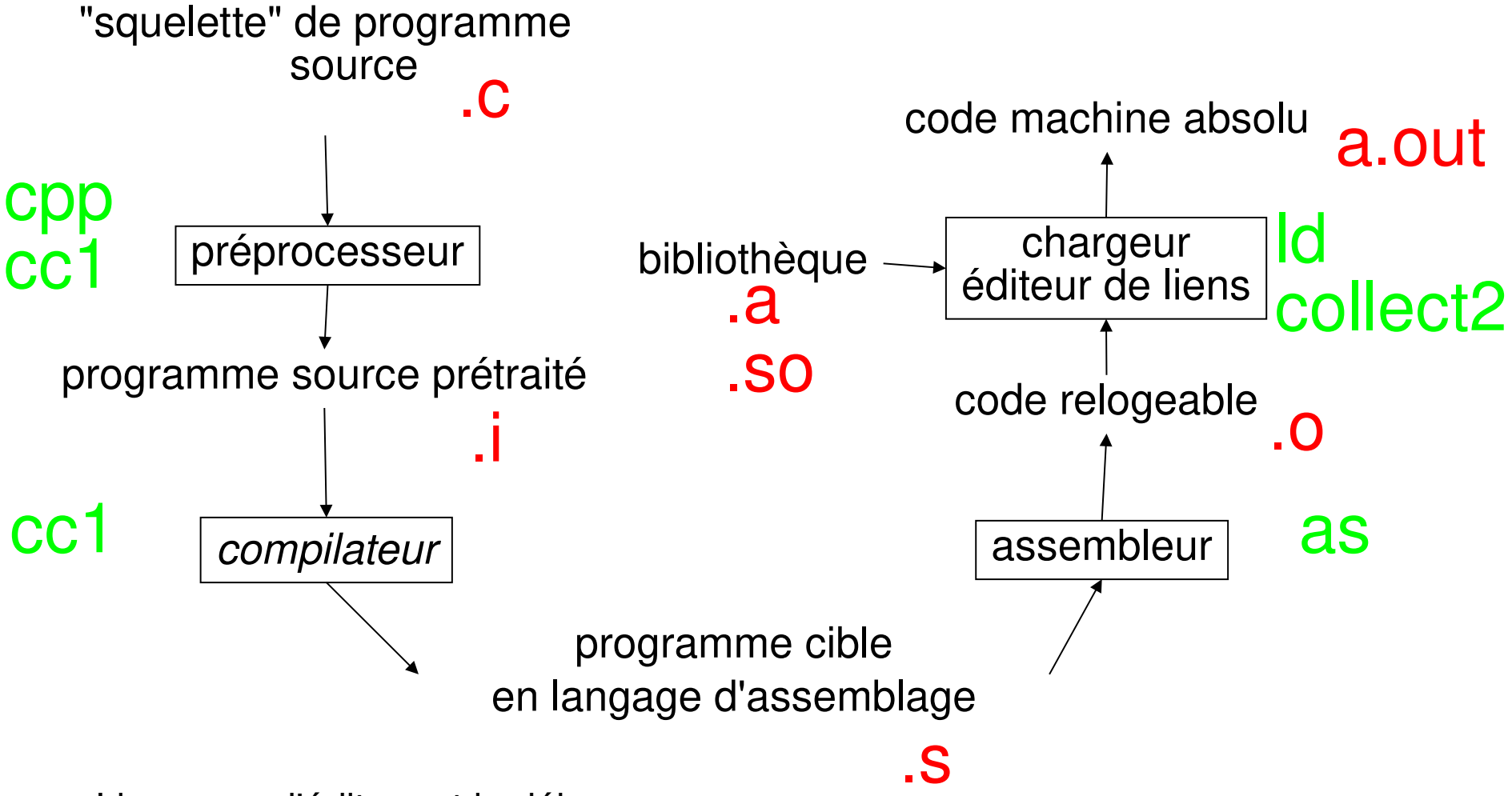
Exemples:
- lisp
- ML
- java

Du code source au code exécutable



Liens avec l'éditeur et le débogueur

Du code source au code exécutable : exemple du C (gcc sous linux)



Liens avec l'éditeur et le débogueur

Du code source au code exécutable : exemple du C (gcc sous linux)

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

int main(void) {
  printf("Hello world !\n");
  return EXIT_SUCCESS;
}
```

cpp hello.c -o hello.i

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdio.h" 1 3 4
...
# 1 "/usr/include/features.h" 1 3 4
# 374 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
...
typedef unsigned char __u_char;
...
extern int printf (const char *__restrict __format, ...);
...
# 3 "hello.c" 2

int main(void) {
  printf("Hello world !\n");
  return 0;
}
```

Les #include sont remplacés par le contenu des fichiers

Les macros sont remplacées par leur valeur

Des informations sont ajoutées pour permettre au compilateur d'avoir des messages d'erreurs significatifs

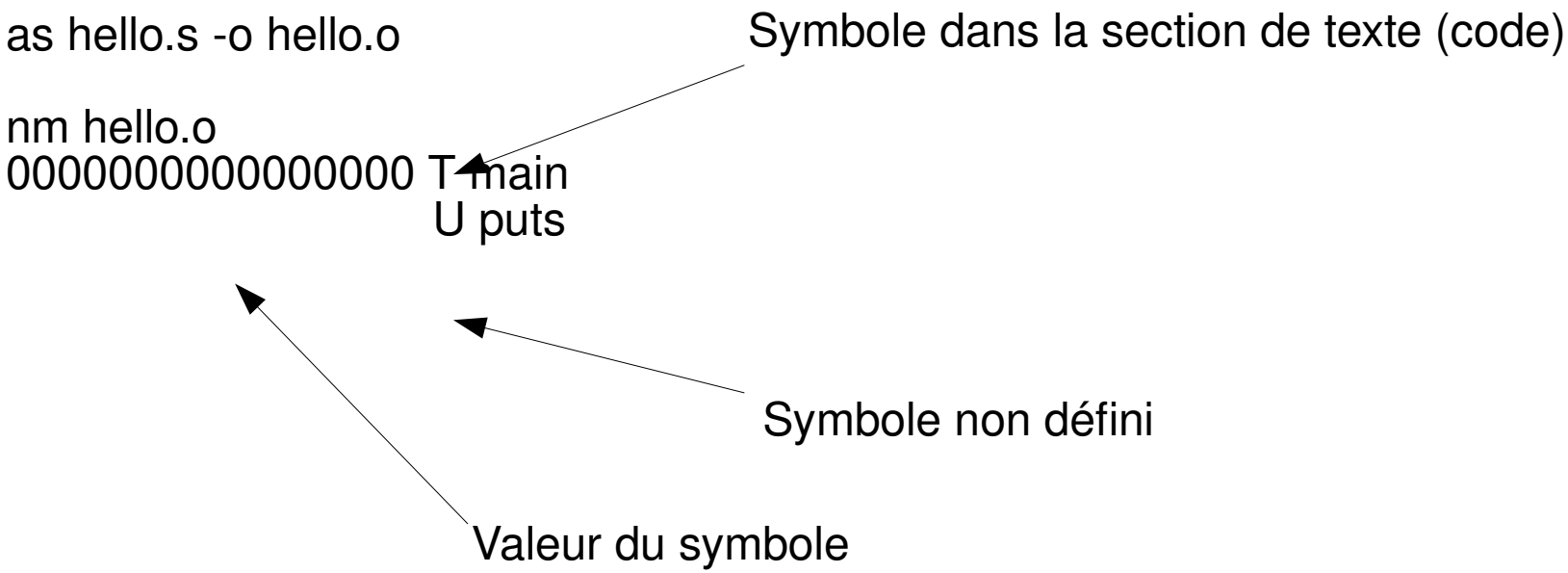
Du code source au code exécutable : exemple du C (gcc sous linux)

gcc -fpreprocessed hello.i -S

(produit un fichier texte : hello.s)

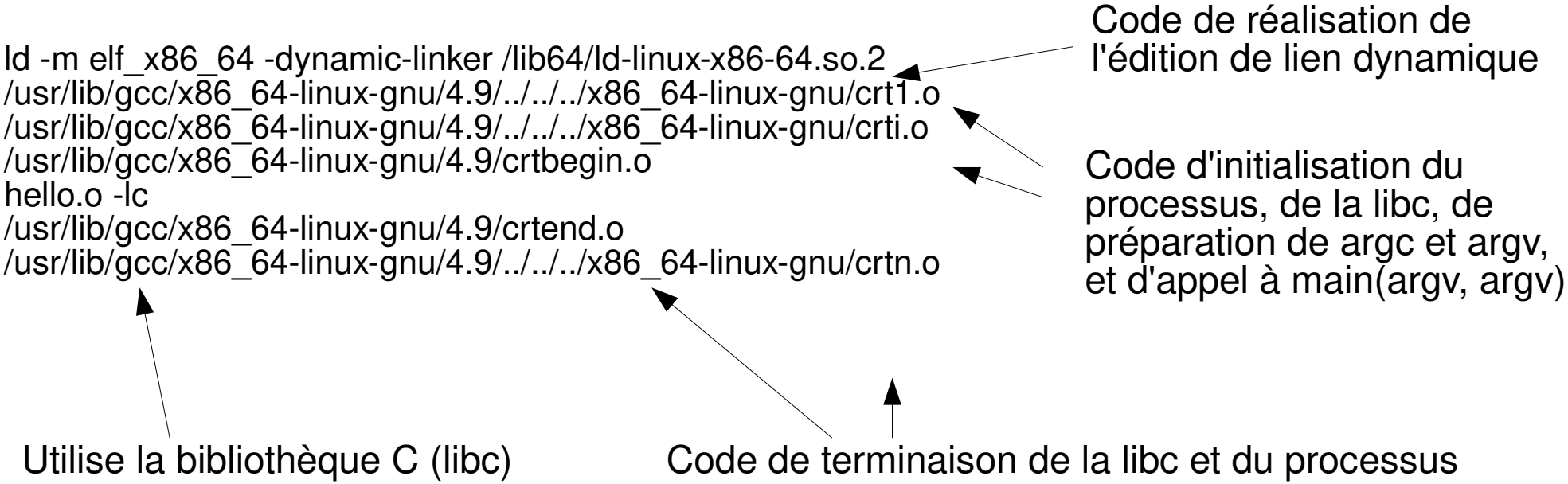
```
.file     "hello.c"
.section        .rodata
.LC0:
.string "Hello world !"
.text
.globl  main
.type   main, @function
main:
.LFB2:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq  %rsp, %rbp
.cfi_def_cfa_register 6
movl   $.LC0, %edi
call     puts
movl   $0, %eax
popq   %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE2:
.size   main, .-main
.ident  "GCC: (Debian 4.9.2-10) 4.9.2"
.section        .note.GNU-stack,"",@progbits
```

Du code source au code exécutable : exemple du C (gcc sous linux)



- Produit un fichier binaire (chaque instruction assembleur a été traduite en son code opératoire)
- Les noms symboliques (variables, routines) des objets définis
  dans l'unité de compilation ont des adresses relatives
- Les autres pas encore

Du code source au code exécutable : exemple du C (gcc sous linux)



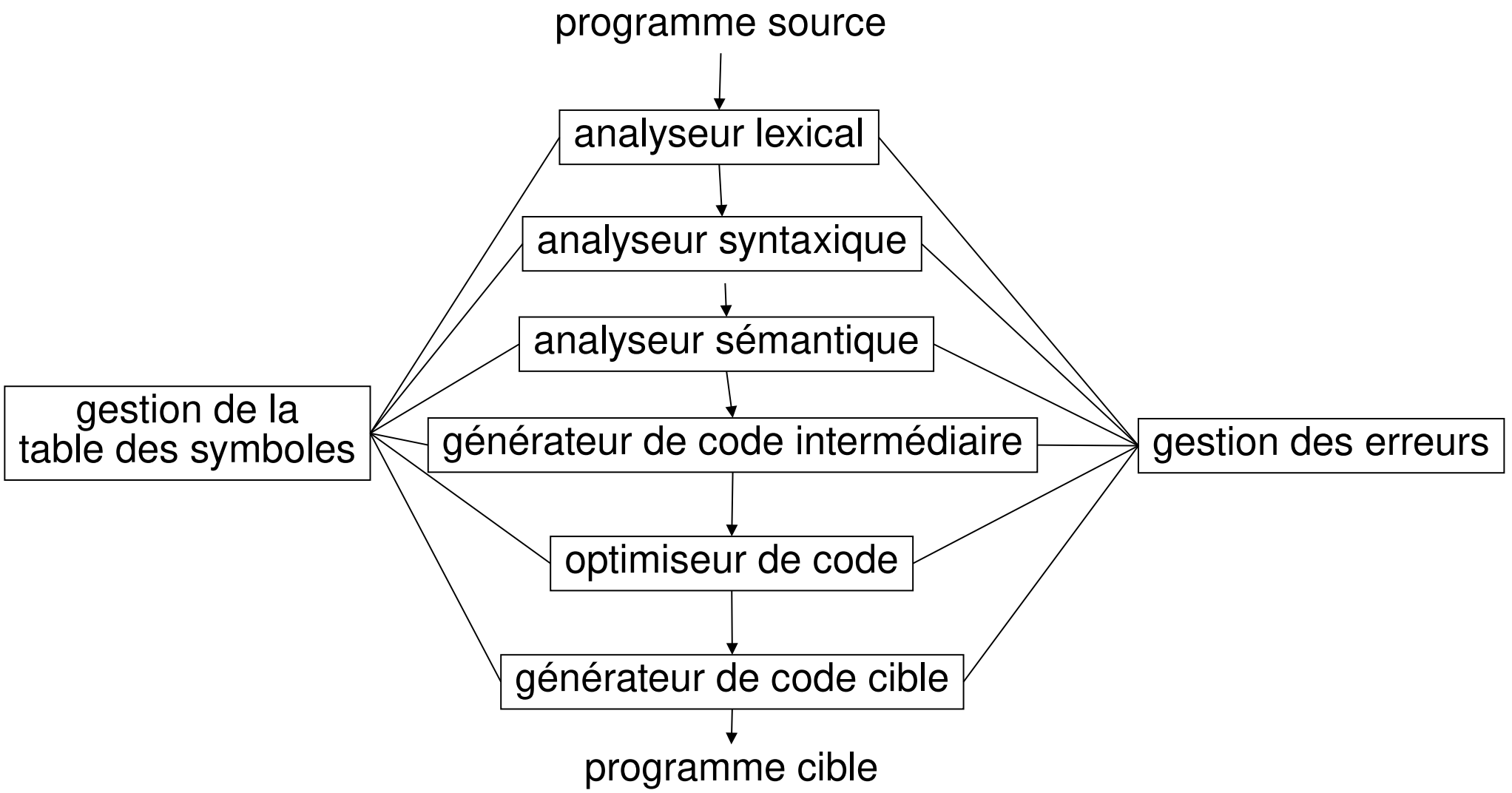
- Ajoute le code nécessaire à l'exécution du processus
- Résout les liens entre les différents fichiers objets
- Spécifie comment doivent être résolus les autres liens (ici, chargement dynamique)
- Recalcule les adresses en prévision de la création du processus
- Créé l'exécutable a.out

Du code source au code exécutable : exemple du C (gcc sous linux)

nm a.out

```
0000000000600890 B __bss_start
0000000000600890 b completed.6661
0000000000600880 D __data_start
0000000000600880 W data_start
00000000004003f0 t deregister_tm_clones
0000000000400470 t __do_global_dtors_aux
0000000000600668 t __do_global_dtors_aux_fini_array_entry
0000000000600888 D __dso_handle
0000000000600678 d _DYNAMIC
0000000000600890 D _edata
0000000000600898 B _end
0000000000400544 T _fini
0000000000400490 t frame_dummy
0000000000600660 t __frame_dummy_init_array_entry
0000000000400658 r __FRAME_END__
0000000000600850 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000400358 T _init
0000000000600668 t __init_array_end
0000000000600660 t __init_array_start
0000000000400550 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000600670 d __JCR_END__
0000000000600670 d __JCR_LIST__
                 w _Jv_RegisterClasses
0000000000400540 T __libc_csu_fini
00000000004004d0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000004004b6 T main
                 U puts@@GLIBC_2.2.5
0000000000400430 t register_tm_clones
00000000004003c0 T _start
0000000000600890 D __TMC_END__
```

Structure interne d'un compilateur



Analyse lexicale

Analyse du programme source en constituants minimaux, les **lexèmes**
On passe de

```
position = initial + vitesse * 60
```

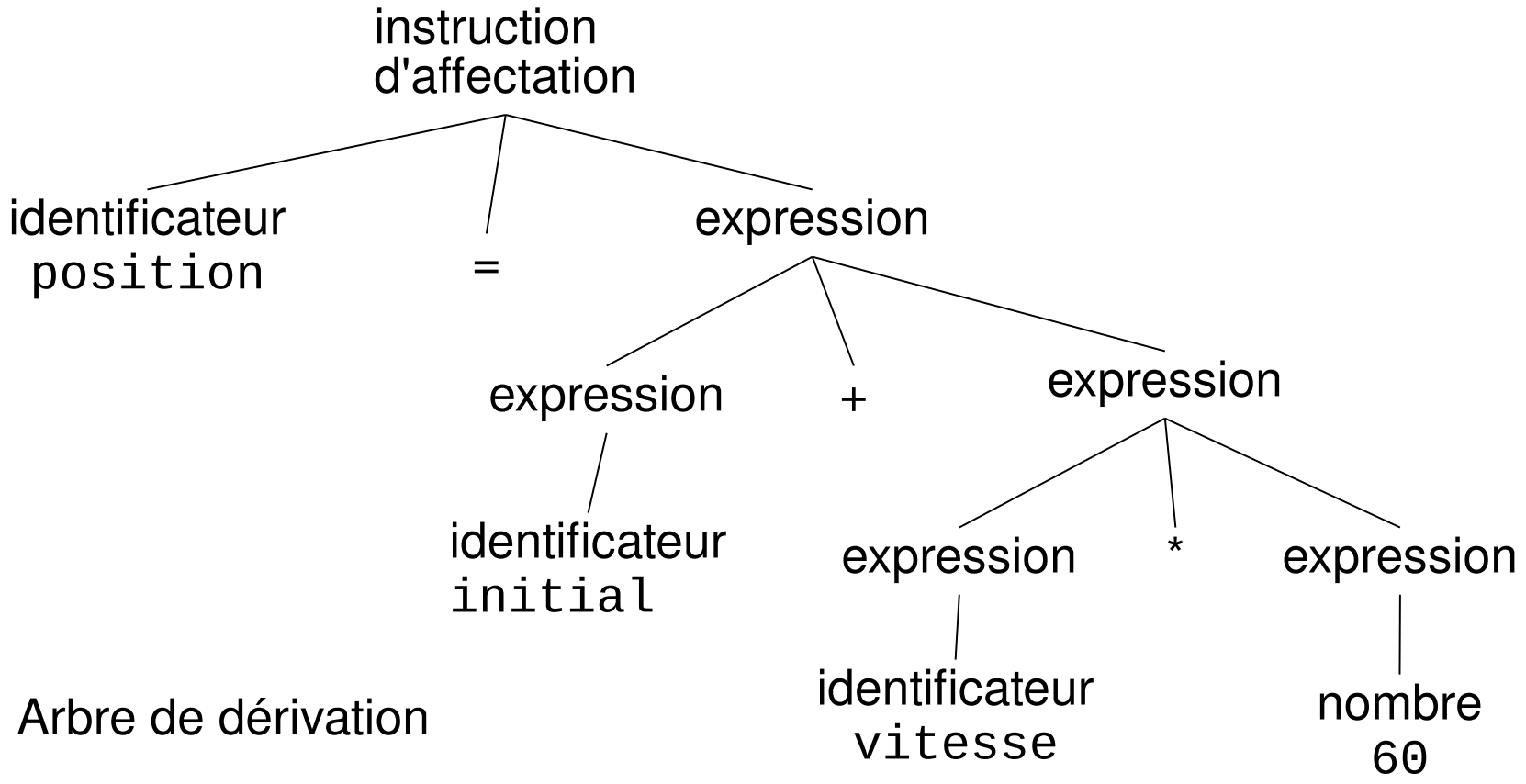
à

```
[id, 1] [=] [id, 2] [+] [id, 3] [*] [60]
```

Les identificateurs rencontrés sont placés dans la table des symboles
Les blancs et les commentaires sont éliminés

Analyse syntaxique

On reconstruit la structure syntaxique de la suite de lexèmes fournie par l'analyseur lexical



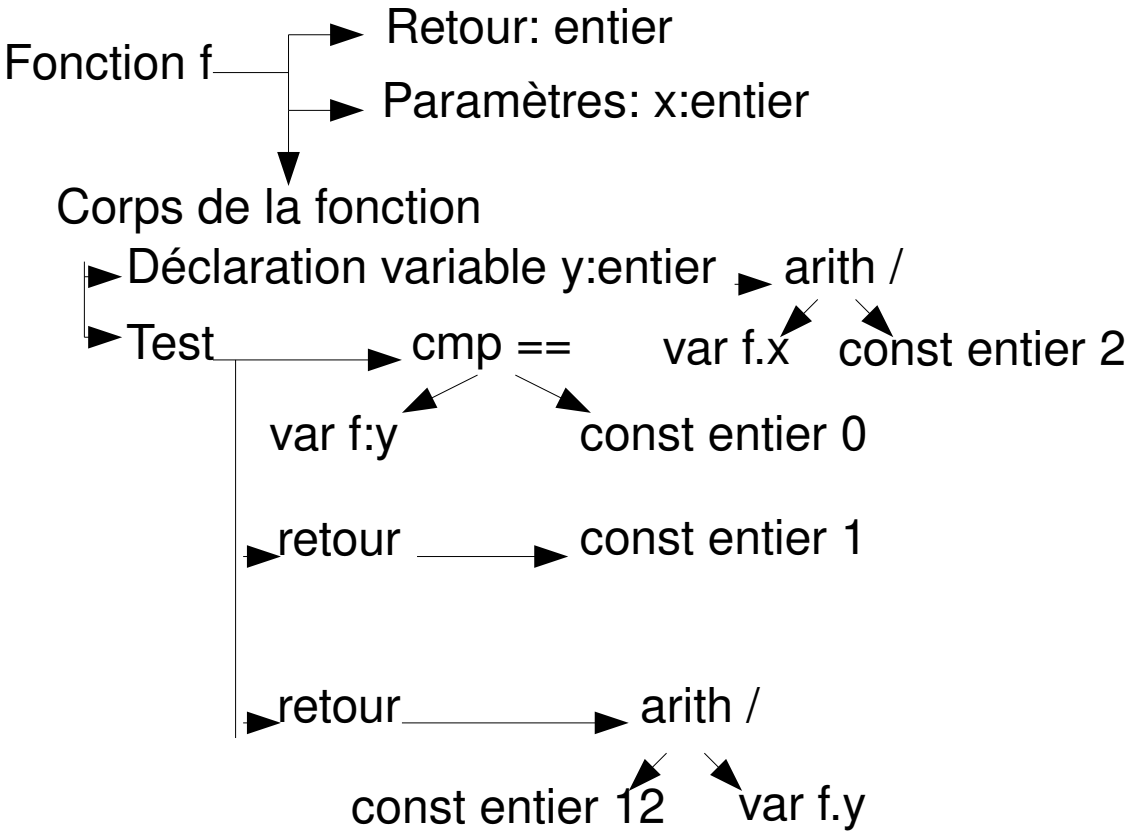
Arbre de dérivation

Représentation intermédiaire

```
int f(int x) {
        int y=x/2;
        if (y==0)
                return 1;
        else
                return 12/y;
        }
```

Table des symboles

f:entier->entier:infos
f.x:entier:infos
f.y:entier:infos



Génération de code intermédiaire

Programme pour une machine abstraite

Représentations utilisées

- Code à trois adresses

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

- Arbres syntaxiques

Optimisation de code

Élimination des opérations inutiles pour produire du code plus efficace.

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

La constante est traduite en réel flottant à la compilation, et non à l'exécution

La variable temp3 est éliminée

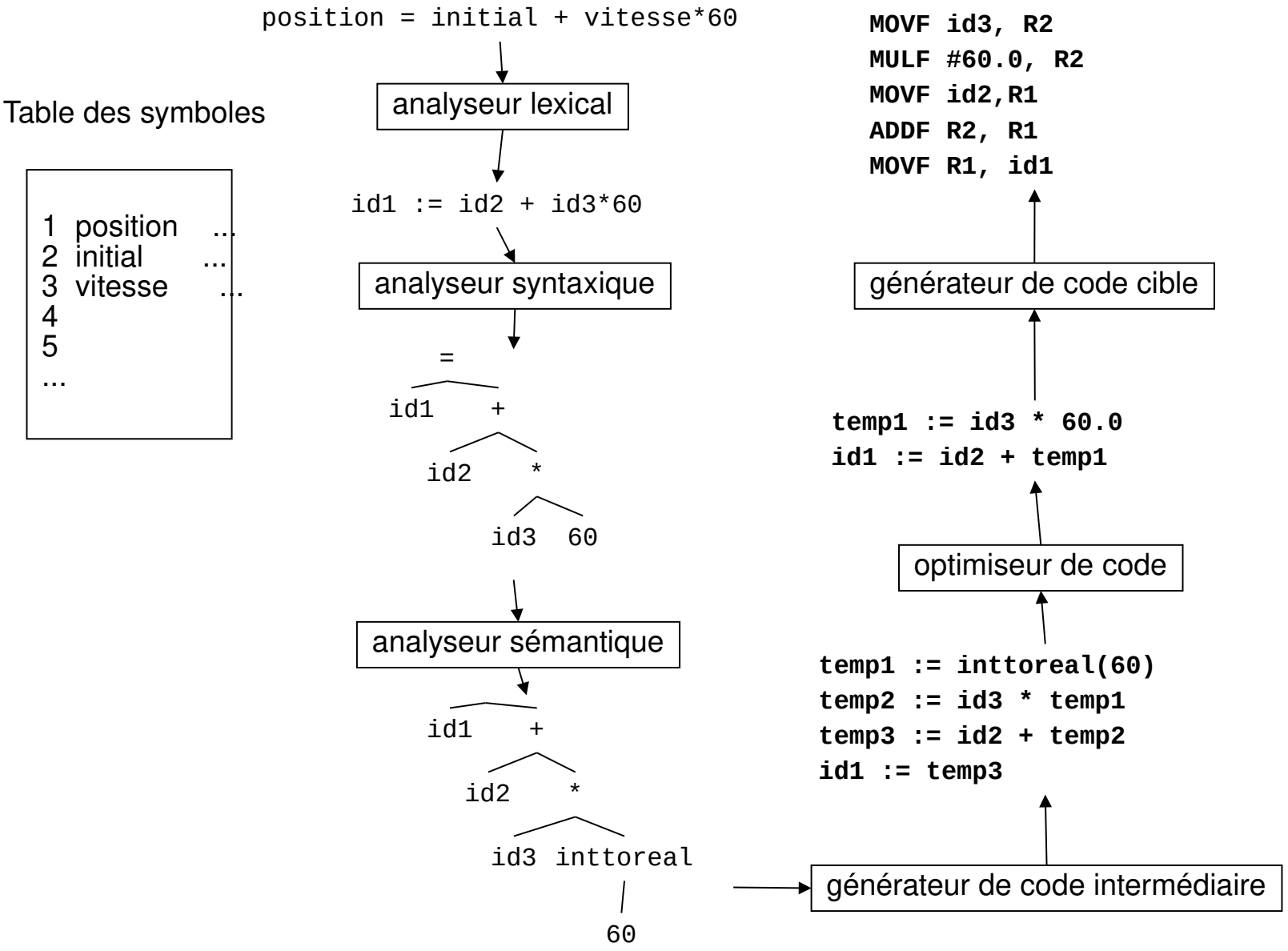
Génération de code cible

La dernière phase produit du code en langage d'assemblage
Un point important est l'utilisation des registres

```
temp1 := id3 * 60.0          MOVF id3, R2
id1 := id2 + temp1           MULF #60.0, R2
                             MOVF id2,R1
                             ADDF R2, R1
                             MOVF R1, id1
```

F = flottant.
La première instruction transfère le contenu de id3 dans le registre R2
La seconde multiplie le contenu du registre R2 par la constante 60.0

Vue d'ensemble



Outils

Outils d'aide à la construction de compilateurs

Générateurs d'analyseurs lexicaux

Engendrent un analyseur lexical (*scanner, lexer*) sous forme d'automate fini à partir d'une spécification sous forme d'expressions rationnelles

Flex, Lex (C)

Générateurs d'analyseurs syntaxiques

Engendrent un analyseur syntaxique (*parser*) à partir d'une grammaire

Bison, Yacc (C)

Générateurs de traducteurs

Engendrent un traducteur à partir d'un schéma de traduction (grammaire + règles sémantiques)

Bison, Yacc (C)

TP

- Processeur simplifié (virtuel) : SImple PROcessor (SIPRO), outil « maison »
- Assembleur pour SIPRO : ASIPRO, outil « maison »

Analyse lexicale

Rôles

- Filtre permettant de « nettoyer » le code source des informations inutiles aux autres phases de compilation
  - suppression des commentaires
  - suppression de l'espacement (espaces, tabulations, retours à la ligne)

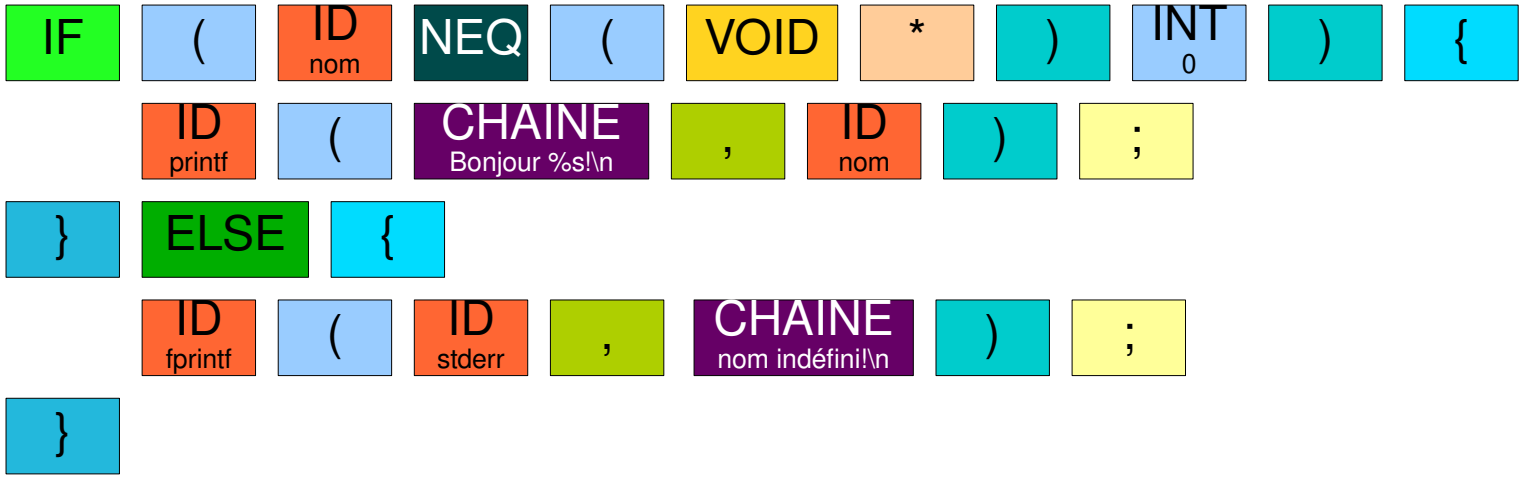
- Première analyse :
  - découpage du flot d'entrée en *lexèmes*

Lexème (*unité lexicale*, *token*) = groupe de caractères consécutifs

Les lexèmes ont des *catégories* et des *attributs*
Exemples de catégories : mots clefs, constante entière, constante réelle, chaîne de caractères
Les attributs sont souvent (mais pas obligatoirement) la valeur

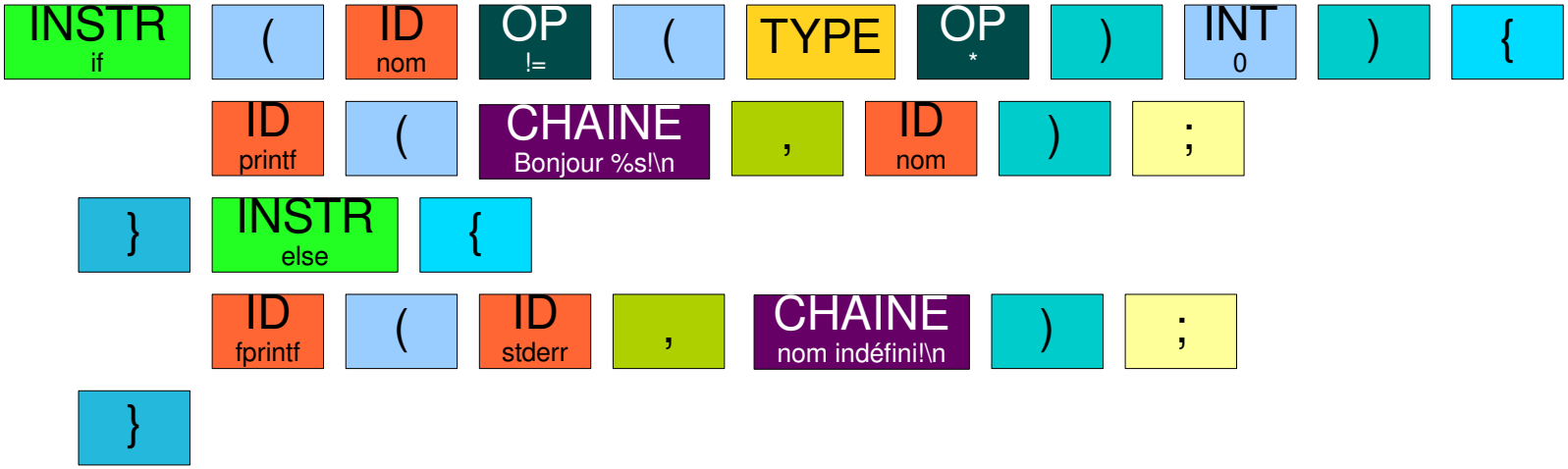
Flot de caractères -> flot de lexèmes

```
if ( nom != (void *)0 ) { // Nom défini
        printf(« Bonjour %s!\n », nom) ;
} else { // Nom indéfini
        fprintf(stderr, « nom indéfini!\n ») ;
}
```



Flot de caractères -> flot de lexèmes

```
if ( nom != (void *)0 ) { // Nom défini
        printf(« Bonjour %s!\n », nom) ;
} else { // Nom indéfini
        fprintf(stderr, « nom indéfini!\n ») ;
}
```



Comment choisir les catégories ?

- L'analyseur lexical transforme un flot de caractères en un flot de lexèmes
- Le flot de lexème est ensuite transmis à l'analyseur syntaxique
  - La catégorie des lexèmes sert de base à l'analyse syntaxique
  - Les attributs ne sont (typiquement) pas utilisés avant l'analyse sémantique
- Le choix des catégories n'est pas unique mais, par exemple,
  - la virgule est en C à la fois un séparateur d'arguments/paramètres et un opérateur
  - l'étoile est en C à la fois un constructeur de type et un opérateur
  - les parenthèses sont à la fois des éléments de construction des expressions, des éléments de syntaxe encadrant les conditions, et des opérateurs
- Comment choisir leur catégorie ?
- Ça sera à l'analyseur syntaxique de décider, en fonction du contexte (de la virgule, de l'étoile, ...)
- L'analyseur lexical doit juste transmettre la virgule (ou l'étoile, etc) à l'analyseur syntaxique, qui décidera de son interprétation (séparateur, opérateur, etc)
  - la virgule forme donc, à elle seule, sa propre catégorie (idem pour l'étoile, les parenthèses)

Comment choisir les catégories ?

Pour les langages de programmation, les catégories les plus courantes sont :

- une catégorie pour chaque mot-clef
- une catégorie pour chaque type numérique
  - entiers
  - réels
- une catégorie pour les caractères
- une catégorie pour les booléens
- une catégorie pour les chaînes de caractères
- une catégorie pour chaque symbole de ponctuation
- une catégorie pour les identificateurs
  - fonctions
  - variables
  - paramètres
  - types
  - ...

Flex et Bison

- Années 70 : Lex & Yacc
  - Lex : Lexical analyser (analyseur lexical)
  - Yacc : Yet Another Compiler Compiler (analyseur syntaxique)
  - Lex et Yacc sont conçus pour fonctionner en binôme
  - Outils d'écriture de compilateurs en C
  - Outils propriétaires (laboratoires Bell)
- Années 90 : Flex et Bison
  - Flex (Fast Lex) : version libre de Lex
  - Bison (jeu de mot sur Yacc) : version libre de Yacc
- Adaptations en C++, ocaml, etc
- Il existe aussi de nombreux autres outils pour écrire des compilateurs
- Ces deux là sont des standards largement répandus
- Utilisés pour analyser Ruby, Php, Go, Bash, Lilypond, PostgreSQL, GNU Octave, Perl 5
- Ont été utilisés pour GCC, mais GCC utilise maintenant une descente récursive manuelle

Flex

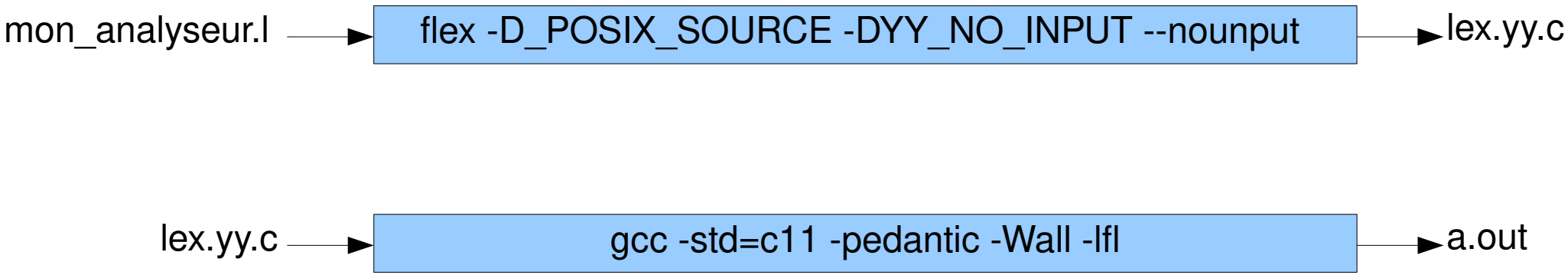
Principe :

- Analyse l'entrée standard
- Génère un flot texte (le flot de lexèmes) sur la sortie standard
- Les catégories sont définies par des expressions rationnelles
- A chaque catégorie est associée du code C à exécuter quand elle est reconnue

Détails de fonctionnement :

- La configuration de flex est décrite dans un fichier texte (extensions .l ou .lex)
- Flex traduit les expressions rationnelles en automates
- Le code C associé à chaque catégorie est associé aux états finaux de l'automate de la catégorie
- Les automates, et le programme qui les fait fonctionner sur l'entrée standard, sont générés dans un fichier C (typiquement yy.lex.c)

- lex.yy.c peut être lié à la bibliothèque C Flex (-lfl) pour former un programme



Flex : un premier exemple (simpliste)

```
cat ex1.txt
if ( nom != (void *)0 ) { // Nom défini
        printf(« Bonjour %s!\n », nom) ;
} else { // Nom indéfini
        fprintf(stderr, « nom indéfini!\n ») ;
}
```

```
cat ex1.l
%%
if              printf("IF");
then            printf("THEN");
else            printf("ELSE");
void            printf("VOID");
\"[^"]*\"       printf("CHAINE");
[a-zA-Z]+       printf("ID");
[0-9]+          printf("INT");
!=              printf("NEQ");
\/\/.*          /* Ne rien faire */
%%
```

```
flex -D_POSIX_SOURCE --nounput
        -DYY_NO_INPUT ex1.l
gcc -std=c11 -pedantic -Wall lex.yy.c -lfl
./a.out <ex1.txt
IF ( ID NEQ (VOID *)INT ) {
        ID(CHAINE, ID) ;
} ELSE {
        ID(ID, CHAINE) ;
}
```

Format d'un fichier flex

%{
    Code C
%}
Définitions flex

%%
$regexp_1$ $code_1$
…
$regexp_n$ $code_n$
%%

Code C

Exemple

```
%{
#include <stdlib.h>
static size_t keyword_count = 0;
static size_t string_count = 0;
static size_t id_count = 0;
static size_t int_count = 0;
static size_t op_count = 0;
static size_t type_count = 0;
static size_t comment_count = 0;
static size_t other_count = 0;
static void print_count(void);
%}
NUMBER [[:digit:]]+
ID [a-zA-Z]+[[:digit:]a-zA-Z]*
STRING \"[^"]*\"
%%
```

```
if              ++keyword_count;
then            ++keyword_count;
else        ++keyword_count;
void        ++type_count;
{STRING}        ++string_count;
{ID}            ++id_count;
{NUMBER}        ++int_count;
!=              ++op_count;
\/\/.*          ++comment_count;
.|\n            ++other_count;
<<EOF>>     { print_count(); return 0; }
%%

static void print_count(void) {
printf(  "#keyword=%zu\n"  "#string=%zu\n"  "#id=%zu\n"
  "#int=%zu\n"  "#op=%zu\n"  "#type=%zu\n"
  "#comment=%zu\n"  "#other=%zu\n",
  keyword_count,  string_count, id_count, int_count,
  op_count, type_count, comment_count, other_count);
}
```

Ou bien...

```
<même section que précédemment>
%%
if              ++keyword_count;
then            ++keyword_count;
else            ++keyword_count;
void            ++type_count;
{STRING}        ++string_count;
{ID}            ++id_count;
{NUMBER}        ++int_count;
!=              ++op_count;
\/\/.*          ++comment_count;
.|\n            ++other_count;
<<EOF>>     { print_count(); return 0; }
%%
<même fonction print_count() que précédemment>

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

Format des expressions rationnelles de flex

Essentiellement, mais pas exactement, les expressions rationnelles étendues POSIX

- caractères « standards »
- . : tout sauf la fin de ligne
- \n : la fin de ligne
- [*caractères*] : classe de caractères. Les caractères entre [] sont en général despécialisés
  - [abc] : {a,b,c}
  - [a-zA-Z] : les caractères des alphabets minuscules et majuscules.
    - Les intervalles sont définis en utilisant le code machine des caractères (ex : code ASCII)
  - [^caractères] : classe de caractères définie par négation
  - [a-z]{-}[mn] : classe de caractères définie par différence ensembliste
- ^ : début de ligne
- $: fin de ligne
- * : s'applique à un ensemble, pour le répéter un nombre quelconque de fois (y compris 0 fois)
- + : s'applique à un ensemble, pour le répéter un nombre quelconque de fois (au moins une fois)
- {*nombre*} : s'applique à un ensemble, pour le répéter *nombre* fois
- {*nombre1*,*nombre2*} : s'applique à un ensemble pour le répéter entre *nombre1* et *nombre2* fois
- *caractère* : despécialisation du caractère
- « ... » : tout ce qui se trouve entre les « » est interprété littéralement
- | : la disjonction
- :digit: : la classe des chiffres (il existe d'autres classes définies de cette manière)
- / : expression rationnelle contextuelle (ex : 0/1 signifie 0, mais seulement quand il est suivi de 1)
- beaucoup d'autres choses : cf. le manuel de flex

[-+]?([0-9]*\.?[0-9]+|[0-9]+\.)(E(+|-)?[0-9]+)?: nombres en Fortran

Ambiguïtés (plusieurs candidats)

Quand plusieurs expressions rationnelles peuvent être utilisées pour consommer l'entrée :

- flex sélectionne celles qui consomment le plus de caractères sur l'entrée
- s'il n'y en a qu'une, c'est elle qui est utilisée
- s'il y en a plusieurs, alors c'est la première apparaissant dans le fichier flex qui est utilisée

```
%%
[[:digit:]]+   printf("Nombre");
.*             /* Rien */
%%
```

```
% cat ex.txt
un mot
456456
123 45
% ./a.out <ex.txt

Nombre

%
```

```
%%
.*             /* Rien */
[[:digit:]]+   printf("Nombre");
%%
```

```
% cat ex.txt
un mot
456456
123 45
% ./a.out <ex.txt


%
```

En général, les expressions rationnelles . ou .* sont en dernier !

Aucun candidat

Quand aucune expression rationnelle ne peut être sélectionnée pour lire l'entrée la règle par défaut s'applique :

- recopier le caractère courant sur la sortie
- passer au caractère suivant

L'analyseur lexical le plus court recopie le flot d'entrée sur le flot de sortie :

```
$ cat ex.l
%%
%%
$ ./a.out <ex.l
%%
%%
$
```

Contextes (start conditions)

Il est possible d'activer/désactiver des expressions rationnelles au fur et à mesure de l'analyse

```
%x comment
%%
"/*"                    BEGIN(comment);
<comment>[^*\n]*        /* consomme ce qui n'est pas un '*' */
<comment>"*"+[^*/\n]*   /* consomme les * qui ne sont pas suivies par un / */
<comment>\n
<comment>"*"+"/"        BEGIN(INITIAL);
```

- Le contexte « comment » active les 4 expressions rationnelles quand /* a déjà été lu
- Toute expression est activée dans un contexte. Le contexte initial s'appelle INITIAL
- BEGIN permet de changer de contexte (il n'y a pas de END!)
- Les contextes peuvent être déclarés exclusifs (%x) ou pas (%s)

```
%s exemple
%%
<exemple>toto
titi
```

est équivalent à

```
%x exemple
%%
<exemple>toto
<exemple,INITIAL>titi
```

- L'utilisation des contextes peut permettre d'éviter d'écrire des expressions rationnelles trop complexes, ou permet de reconnaître des ensembles de lexèmes non rationnels

Pile de contextes

Les noms des contextes sont en fait des entiers

Flex met à la disposition du programmeur un mécanisme de pile pour les contextes
%option stack

La pile grossit dynamiquement.

- void yy_push_state (int new_state)
  Sauve le contexte courant sur la pile et réalise un BEGIN new_state

- void yy_pop_state ()
  Retire un contexte c du sommet de pile et réalise un BEGIN c

- int yy_top_state ()
  Retourne le contexte en sommet de pile en le laissant dans la pile

Quelques variables et macros utiles de flex

- yytext : contient le motif lu
  - char * ou char[YYLMAX] (voir %option)
  - contenu modifiable par l'utilisateur, à condition de ne pas changer la longueur !
- yyleng : longueur du motif lu
- ECHO : envoie le motif lu sur yyout
- REJECT :
  cette expression rationnelle rejette le motif,
  l'analyse reprend avec les autres expressions rationnelles
- yyin : le flot d'entrée de yylex
- yyout : le flot de sortie de yylex

Il y en a bien d'autres. Par exemple, pour permettre d'empiler les flots
(gestion des #include en C). Voir le manuel.

```
%%
a       ECHO; REJECT;
ab      ECHO; REJECT;
abc     ECHO; REJECT;
abcd    ECHO; REJECT;
%%
```

Sur l'entrée abcd, sort abcdabcabaabcd

Quelques fonctions utiles de flex

- yymore() : indique à flex que le prochain motif doit être concaténé au courant dans yytext
  plutôt que de le remplacer
- yyless(n) : indique à flex que le motif courant privé de son préfixe de taille n
  doit être remis sur l'entrée
- unput(c) : le caractère c doit être remis sur l'entrée (attention : détruit yytext si c'est un char *)
- input() : lit le prochain caractère de l'entrée
- yyterminate() :
  - peut remplacer un return dans yylex().
  - termine l'analyseur lexical
  - retourne 0 à l'appelant
  - c'est une macro, qui peut être redéfinie.

yywrap()

- Appelée par flex quand il rencontre EOF sur yyin
- Retourne 1 pour indiquer à flex que l'analyse est terminée
- Retourne 0 pour indiquer à flex qu'il faut continuer : dans ce cas, yywrap() aura au préalable fait le nécessaire pour assurer que le flot d'entrée continue
- En fait, c'est une macro

```
%{
int g_argc;
char **g_argv;
int current;
size_t cc=0, lc=0, wc=0;
%}
SEP     [ \t]
NSEP  [^ \t\n]
%%
{NSEP}+         cc += yyleng; ++wc;
{SEP}+          cc += yyleng;
\n              ++lc; ++cc;
%%

int yywrap() {
  if ( yyin != stdin ) fclose(yyin);
  if ( current == g_argc-1 ) return 1;
  if ( ( yyin = fopen(g_argv[++current], "r") ) == NULL ) {
    perror(g_argv[current]) ; exit(EXIT_FAILURE);
  }
  return 0;
}
```

```
int main(int argc, char *argv[]) {
  g_argc = argc;
  g_argv = argv;
  if ( argc == 1 ) {
    yyin = stdin;
    current = 1;
  } else {
    current = 1;
    if ( ( yyin = fopen(argv[current], "r") ) == NULL ) {
      perror(argv[current]) ;
      return EXIT_FAILURE;
    }
  }
  yylex();
  printf("%8zu %8zu %8zu\n", lc, wc, cc );
  return EXIT_SUCCESS;
}
```

main() de flex

- Celui de la bibliothèque de flex appelle yylex() dans une boucle

```
while ( yylex() ) ;
```

- Celui généré par %option main se contente d'appeler yylex()

```
int main(int argc, char *argv[]) {
        yylex() ;
        return 0;
}
```

- L'utilisateur peut toujours écrire le sien !

  A préférer pour une meilleure maîtrise du programme (environnement, locale, etc)

%option

- array : yytext est déclaré comme un tableau de YYLMAX caractères
- pointer : yytext est un char *, dont la taille varie dynamiquement en fonction des besoins
- main : lex intègre un main dans le code qu'il produit, qui se contente d'appeler yylex()
- never-interactive : le code généré considère que son entrée n'est pas interactive
  (ex : un terminal)
- always-interactive : opposé de %never-interactive
- stdinit : yyin et yyout sont initialisées à stdin et stdout
- yylineno : lex compte le numéro de la ligne analysée dans une variable globale yylineno
- yywrap : lex appelle yywrap() en fin de flot d'entrée
- noyywrap : lex n'appelle pas yywrap() en fin de flot d'entrée
- outfile= : change le nom de la sortie (lex.yy.c) par un autre (=option -o de flex)
- prefix= : change le prefixe « yy » utilisé pour tous les identificateurs du code généré,
  par un autre (=option -P de flex)

La bibliothèque de flex

Contient deux implantations par défaut pour

- yywrap : retourne toujours 1
- main : appelle yylex() dans une boucle

```
while ( yylex() ) ;
```

Utilisation : -lfl en fin de ligne de compilation gcc

Si aucune des deux implantations par défaut ci-dessus n'est nécessaire, autant ne pas l'utiliser !

Analyse syntaxique

Analyse syntaxique

Entrée : un flot de lexèmes

Sortie : au choix, typiquement un arbre et une table des symboles

But : analyser la structure du flot de lexèmes

Types (usuels) :

- analyses descendantes usuelles : LL(k)
- analyses ascendantes usuelles : LR(k), SLR(k), LALR(k), GLR(k)

Outil formel usuel : grammaires

Outil de programmation : pour nous, Bison (analyse de type LALR(1))

Grammaires algébriques (non contextuelles)

G = (S,N,T,P) avec

T un ensemble (non vide) de symboles appelés *terminaux* (qui pour nous seront les lexèmes)

N un ensemble (non vide) de symboles appelés *non-terminaux* (N disjoint de T)

S un symbole de N, appelé *axiome*

P un ensemble de *productions*

Production : objet de la forme A -> γ avec

A un non-terminal, appelé *membre gauche* de la production

γ une suite d'éléments de T et de N (un mot sur l'alphabet T∪N),

appelé *membre droit* de la production

Exemple : G=(S,N,T,P) avec T = {0,1,…,9,+,-,*,/}, N={E}, S=E,

P = { E->E+E, E->E-E, E->E*E, E->E/E} ∪ {E->x : x ∈ {0,1,…,9}}

Dérivation

Remplacement, dans un mot γ, d'un non-terminal X par le membre droit d'une production de la forme X -> β

Exemple : E+E se dérive en E+E+E (ou E+E+E),
mais aussi en 4+E, en E+5, en E-E+E, en E+E/E, en 1+2, etc

La relation de dérivation se note =>, sa fermeture transitive et réflexive =>*

Exemple : E => E+E => E+E-E => 1+E-E => 1+E-4 => 1+E/E-4 => 1+E/3-4 => 1+2/3-4
E =>* 1+2/3-4

Le langage d'une grammaire est l'ensemble des mots sur ses terminaux qu'on obtient à partir de l'axiome

Exemple : le langage de la grammaire G précédente est l'ensemble des expressions arithmétiques sur les chiffres

Un langage est algébrique s'il peut être défini par une grammaire algébrique
Tous les langages ne sont pas algébriques (cf. cours de théorie des langages)

Arbre de dérivation

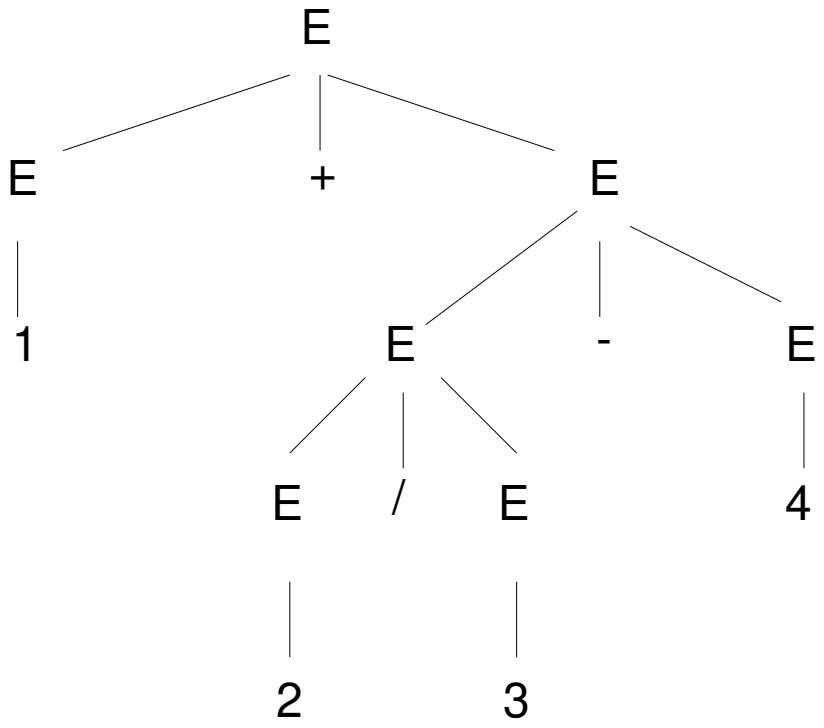
Les dérivations successives appliquées sur un non-terminal peuvent être représentées graphiquement sous la forme d'un arbre, appelé *arbre de dérivation*

E => E+E => E+E-E => 1+E-E => 1+E-4 => 1+E/E-4 => 1+E/3-4 => 1+2/3-4

Un arbre de dérivation
pour 1+2/3-4 à partir de E

La suite des feuilles lues de la
gauche vers la droite est 1+2/3-4

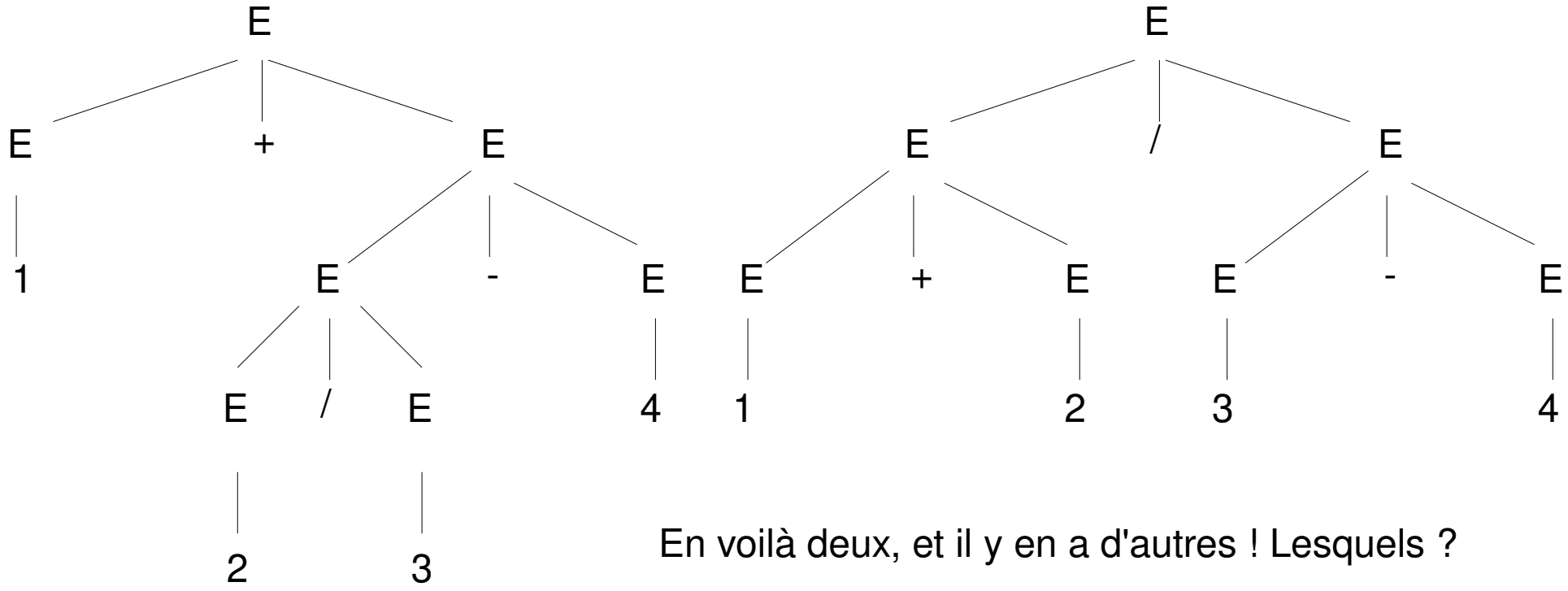
L'arbre de dérivation donne
la structuration des lexèmes entre eux



Grammaire ambiguë

Une grammaire est *ambiguë* s'il existe plusieurs arbres de dérivation pour un mot *u* à partir d'un non-terminal X (donc, plusieurs structurations des lexèmes entre eux)

La grammaire de l'exemple précédent est ambiguë car il existe plusieurs arbres de dérivation pour 1+2/3-4 à partir de E

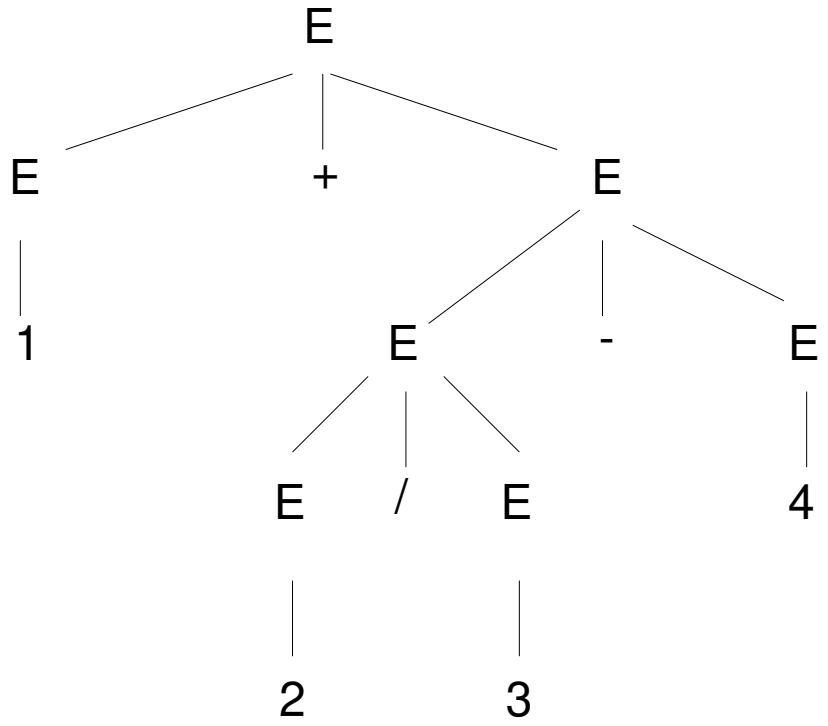


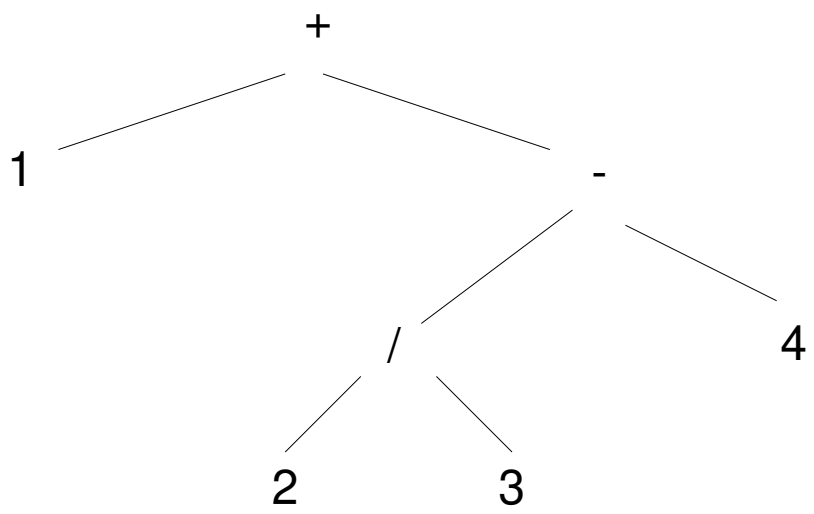
En voilà deux, et il y en a d'autres ! Lesquels ?

Arbre de syntaxe abstraite (AST)

Version simplifiée, sans les non-terminaux, de l'arbre de dérivation

Comme l'arbre de dérivation, l'AST donne une structuration des lexèmes entre eux d'après les productions de la grammaire





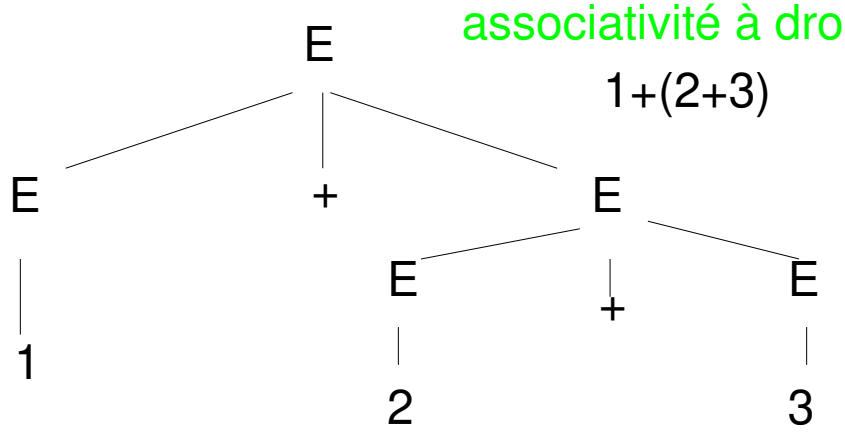
Retour sur l'ambiguïté

Notre grammaire pour les expressions arithmétiques est ambiguë car

Elle ne prend pas en compte l'associativité des opérateurs (ex : E =>* 1+2+3)

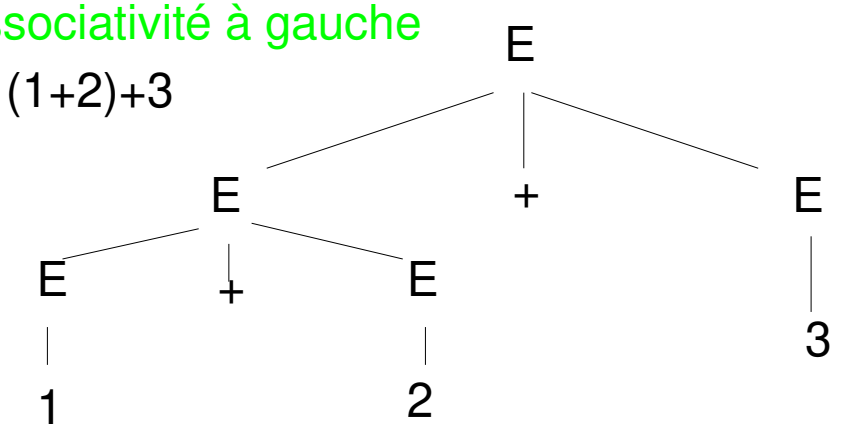
associativité à droite

1+(2+3)



associativité à gauche

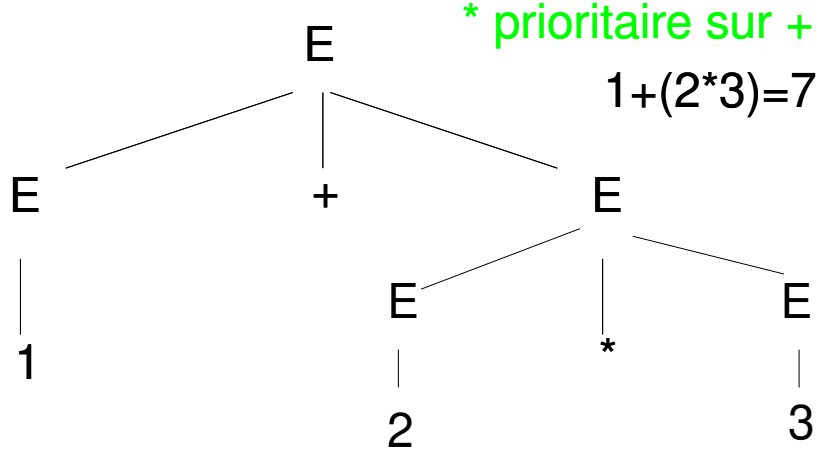
(1+2)+3



Elle ne prend pas en compte la priorité des opérateurs (ex : E =>* 1+2*3)

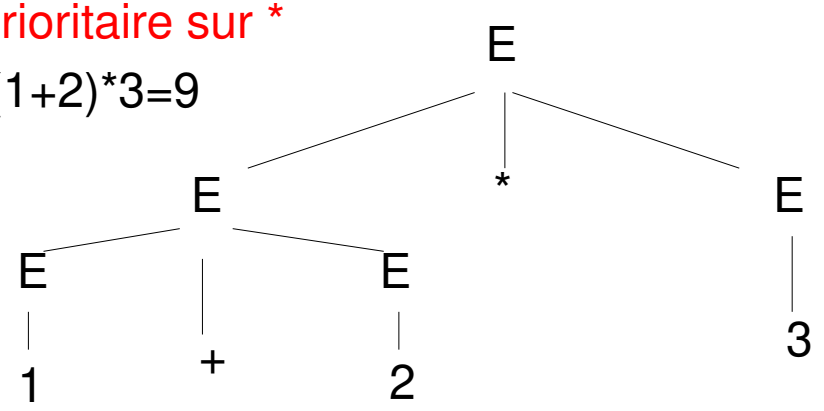
* prioritaire sur +

1+(2*3)=7



\+ prioritaire sur *

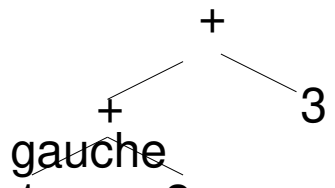
(1+2)*3=9



Transformer la grammaire pour supprimer les ambiguïtés

On supprime les ambiguïtés liées à l'associativité de la somme en changeant

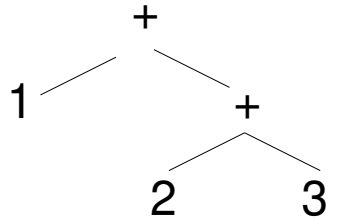
- E -> E+E par E -> E+x, $x \in \{0,1,\ldots,9\}$:
  - on impose l'associativité à gauche par des productions récursives gauches
  - 1+2+3 est interprété comme (1+2)+3



les arbres penchent à gauche

- ou E -> E+E par E -> x+E, $x \in \{0,1,\ldots,9\}$:
  - on impose l'associativité à droite par des productions récursives droites
  - 1+2+3 est interprété comme 1+(2+3)

les arbres penchent à droite

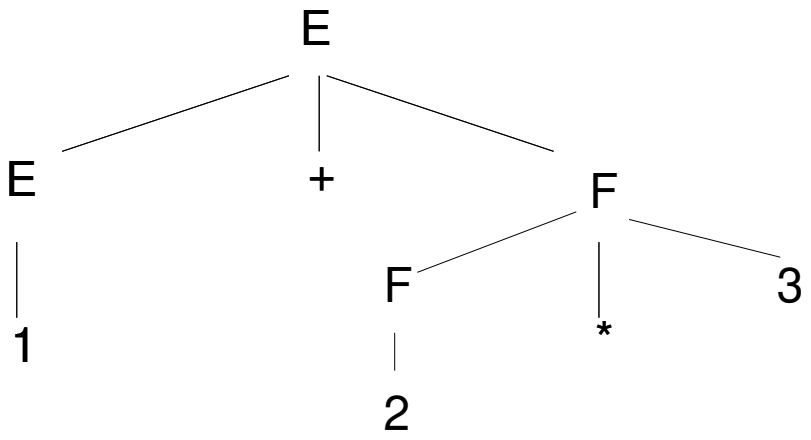


Transformer la grammaire pour supprimer les ambiguïtés

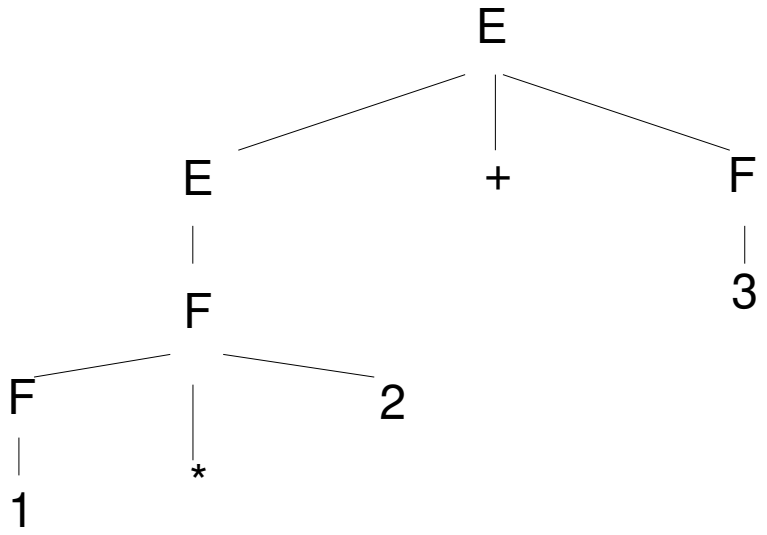
Pour rendre * prioritaire sur +, il faut imposer que
- * apparaissent plus bas, dans les arbres, que +
- ou, de manière équivalente, que la production pour + soit plus proche de l'axiome que *

E-> E+F, E->F, F->F*x, F->x, x ∈ {0,1,…,9}

1+2*3



1*2+3



Une grammaire complète pour les expressions arithmétiques

- On peut toujours s'arranger pour que l'axiome n'apparaisse qu'en membre gauche
- A partir de maintenant on notera toujours l'axiome S, les terminaux en minuscules et les non-terminaux en majuscules
- On notera X -> γ | γ' quand X -> γ et X -> γ' sont des productions

S -> E
E -> E+F | E-F | F
F -> F*T | F/T | T
T -> (E) | C
C -> C0 | C1 | … | C9 | 0 | 1 | … | 9

- Les associativités des opérateurs sont fixées à gauche (elles auraient pu l'être à droite)
- Les priorités sont correctement gérées
- Le parenthésage a été ajouté pour permettre (1+2)*3, par exemple
- Les constantes ont maintenant un nombre quelconque de chiffres

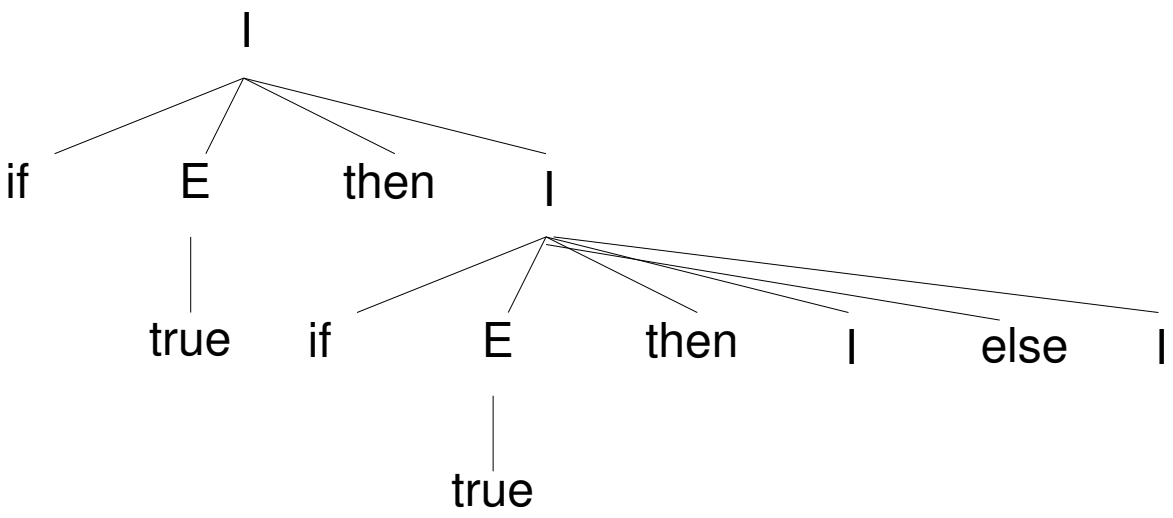
Un autre exemple : if...then...else...

S -> I
I -> if E then I else I | if E then I
E -> true | false | ...

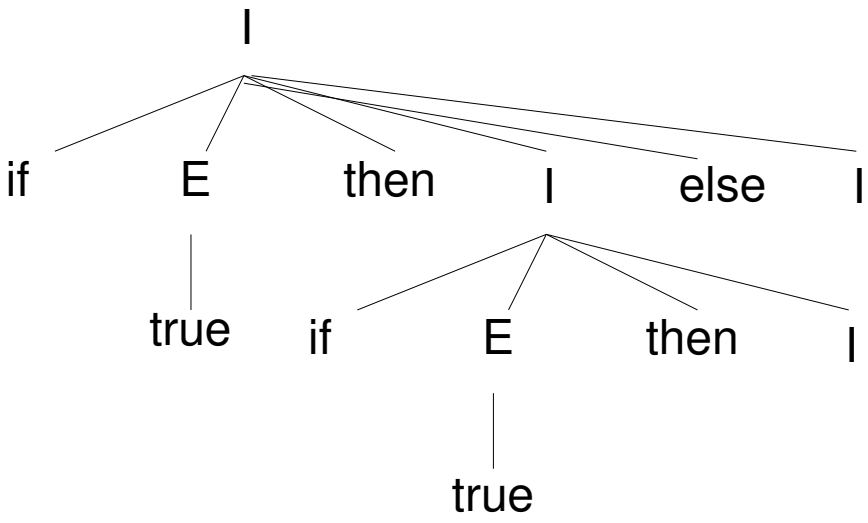
I : les instructions
E : les expressions booléennes

if true then if true then I else I

if true then
    if true then I
    else I



if true then
    if true then I
else I



Un autre exemple : if...then...else...

Le else doit porter sur le if le plus proche
if/then est plus haut, dans l'arbre, que if/then/else : on utilise le même système que pour l'expression de la priorité de * sur +

I' : les instructions sans le if

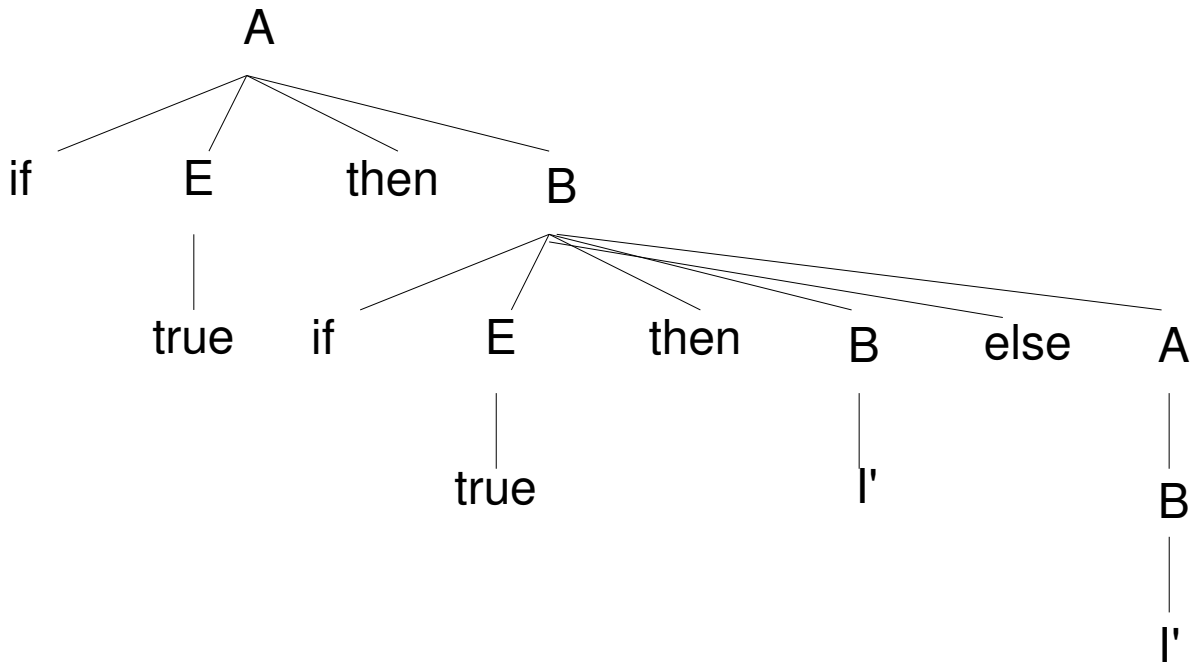
S -> A
A -> if E then B | B
B -> if E then B else A | I' | { A }

if true then
    if true then I
else I

n'a plus d'arbre de dérivation
mais

if true then {
    if true then I
} else I

en a un. Lequel ?



Algorithmes d'analyse syntaxique

Ils doivent être à la fois pratiques

- sens de lecture du flot de lexèmes de la gauche vers la droite (left-to-right)
- les grammaires doivent être le plus simple possible à écrire et analyser

et les plus efficaces possibles en temps d'exécution et mémoire utilisée.

Deux grandes classes d'algorithmes d'analyse syntaxique sont utilisées en pratique

- l'analyse *prédictive* (ou *descendante*) : Left-to-right Leftmost (LL): la lecture du flot de lexèmes se fait de gauche à droite, et on essaie d'abord de dériver le non-terminal le plus à gauche des productions
- l'analyse *ascendante* : Left-to-right Rightmost (LR) : la lecture du flot de lexèmes se fait de gauche à droite, et on essaie d'abord de dériver le non-terminal le plus à droite

Les analyses LL(k) ou LR(k) lisent d'avance k lexèmes (qu'on assimilera à des *caractères*) pour tenter de prédire les actions à réaliser. Souvent, k=1.

Algorithmes d'analyse syntaxique

On suppose k=1 à partir de maintenant. La généralisation à k>1 n'est pas difficile.

Les deux classes d'algorithmes calculent les ensembles suivants sur la grammaire :

- Annulable (Nullable) : l'ensemble des non-terminaux X tels que X =>* ε
- Premier(X) (First(X)) : l'ensemble des terminaux *a* tels que X =>* aγ
- Suivant(X) (Follow(X)) : l'ensemble des terminaux *a* tels que S =>* γXaγ'

en résolvant des systèmes d'équations ensemblistes.

Ces trois ensembles sont utilisés pour produire, à partir d'une grammaire donnée et si c'est possible, des analyseurs descendants ou ascendants avec un caractère de prédiction.

Annulable

Ensemble des non-terminaux X tels que X =>* ε

S -> XYS | a
Y -> c | ε
X -> a | Y

Y est annulable car Y => ε
X est annulable car X => Y et Y est annulable
S n'est pas annulable

Tmp <- {}
Annulable <- { X : X -> ε }
Tant que Annulable != Tmp
    Tmp <- Annulable
    Annulable += { X : X -> $X_1...X_n$, $X_i \in$ Tmp pour tout i $\in$ 1..n }

Premier(X)

Ensemble des terminaux *a* tels que X =>* aγ

S -> E$
E -> E+F | E-F | F
F -> F*T | F/T | T
T -> (E) | C
C -> 0 | 1 | … | 9

Premier(C) = { 0,1, …, 9 }
Premier(T) = { (} ∪ Premier(C)
Premier(F) = Premier(T)
Premier(E) = Premier(F)
Premier(S) = Premier(E)

Premier(X) est le plus petit ensemble vérifiant :

- si X -> aγ alors a $\in$ Premier(X)
- si X -> Yγ alors Premier(Y) $\subseteq$ Premier(X)
- si X -> X_1...X_naγ et X_1...X_n $\in$ Annulable alors a $\in$ Premier(X)
  et Premier(X_i) $\subseteq$ Premier(X) pour tout i $\in$ 1...n
- si X -> X_1...X_nYγ et X_1...X_n $\in$ Annulable alors Premier(Y) $\subseteq$ Premier(X)
  et Premier(X_i) $\subseteq$ Premier(X) pour tout i $\in$ 1...n

Premier(X)

Ensemble des terminaux *a* tels que X =>* aγ

S -> XYS | b
Y -> c | ε
X -> a | Y

Annulable = { X, Y }

Premier(X) = { a } ∪ Premier(Y)
Premier(Y) = { c }
Premier(S) = { b } ∪ Premier(X) ∪ Premier(Y) ~~∪ Premier(S)~~
= { a, b, c }

S -> Yd | ε
Y -> Sc | ε

Annulable = { S, Y }

Premier(S) = { d } ∪ Premier(Y)
Premier(Y) = { c } ∪ Premier(S)

Premier(S) = { d } ∪ { c } ~~∪ Premier(S)~~
Premier(Y) = { c } ∪ Premier(S)

Suivant(X)

Ensemble des terminaux *a* tels que S =>* γXaγ'

Suivant(X) est le plus petit ensemble vérifiant :

- si Y -> γXaγ' alors a $\in$ Suivant(X)
- si Y -> γXZγ' alors Premier(Z) $\subseteq$ Suivant(X)
- si Y -> γX alors Suivant(Y) $\subseteq$ Suivant(X)
- si Y -> γXZ$_1$...Z$_n$aγ et Z$_1$...Z$_n$ sont Annulable alors a $\in$ Suivant(X)
  et Premier(Z$_i$) $\subseteq$ Suivant(X) pour tout i $\in$ 1...n
- si Y -> γXZ$_1$...Z$_n$Kγ et Z$_1$...Z$_n$ sont Annulable alors Premier(K) $\subseteq$ Suivant(X)
  et Premier(Z$_i$) $\subseteq$ Suivant(X) pour tout i $\in$ 1...n
- si Y -> γXZ$_1$...Z$_n$ et Z$_1$...Z$_n$ sont Annulable alors Suivant(Y) $\subseteq$ Suivant(X)
  et Premier(Z$_i$) $\subseteq$ Suivant(X) pour tout i $\in$ 1...n

Par convention, $ est dans les suivants de l'axiome

Suivant(X)

Ensemble des terminaux *a* tels que S =>* γXaγ'

S -> E$
E -> E+F | E-F | F
F -> F*T | F/T | T
T -> (E) | C
C -> 0 | 1 | … | 9

Annulable = {}
Premier(C) = { 0, …, 9 }
Premier(S) = Premier(E) = Premier(F) = Premier(T) = { (, 0, …, 9 }

Suivant(E) = { +, -,), $ }
Suivant(T) contient Suivant(F)
Suivant(F) contient Suivant(E) et { *, / }
Suivant(F) = { +, -, *, /,), $ }
Suivant(T) = { +, -, *, /,), $ }
Suivant(C) = Suivant(T)
Suivant(S)={$}

S -> XYS | b
Y -> c | ε
X -> a | Y

Annulable = { X, Y }
Premier(X) = { a, c }
Premier(Y) = { c }
Premier(S) = { a, b, c }

Suivant(X) = Premier(Y) ∪ Premier(S) = { a, b, c }
Suivant(Y) = Premier(S) ∪ Suivant(X) = { a, b, c }
Suivant(S) = {$}

Annulable, Premier, Suivant

S -> E$
E -> EOE | F
O -> + | - | ε
F -> n | (E)

Annulable = { O }
Premier(F) = { n, (} Premier(O) = { +, - }
Premier(E) = { n, (} Premier(S) = { n, (}
Suivant(S) = {$}
Suivant(O) = Premier(E) = { n, (}
Suivant(E) = {),$ } ∪ Premier(O) ∪ Premier(E) = { n, (,), +, -,$ }
Suivant(F) = Suivant(E) = { n, (,), +, -,$ }

S -> E$
E -> TE'
E' -> +TE' | -TE' | ε
T -> FT'
T' -> *FT' | /FT' | ε
F -> n | (E)

Annulable = { T', E' }
Premier(F) = { n, (} Premier(T') = { *, / } Premier(T) = { n, (}
Premier(E') = { +, - } Premier(E) = { n, (} Premier(S) = { n, (}
Suivant(S) = {$} Suivant(E) = {), $ } Suivant(E') = Suivant(E)
Suivant(T) = Premier(E') ∪ Suivant(E) ∪ Suivant(E') = { +, -,), $ }
Suivant(T') = Suivant(T)
Suivant(F) = Premier(T') ∪ Suivant(T) ∪ Suivant(T') = { *, /, +, -,), $ }

Extension de Annulable et Premier

Ils peuvent s'étendre à une suite de terminaux et non-terminaux

$X_1...X_n$ est annulable si tout les X_i le sont

$a \in$ Premier($X_1...X_n a\gamma$) si n=0 ou tous les X_i sont annulables
Premier(X_n) $\subseteq$ Premier($X_1...X_n\gamma$) si tous les X_j, j <n, sont annulables
Premier($X\gamma$) = Premier(X) si X n'est pas annulable

S -> E$
E -> TE'
E' -> +TE' | -TE' | ε
T -> FT'
T' -> *FT' | /FT' | ε
F -> n | (E)

Annulable = { T', E' }
Premier(F) = { n, (}
Premier(E') = { +, - }
Premier(T') = { *, / }
Premier(E) = { n, (}
Premier(T) = { n, (}
Premier(S) = { n, (}
Premier(TE) = Premier(T)
Premier(T'E) = Premier(T') ∪ Premier(E)

Grammaire transformée, prédiction

Dans les algorithmes d'analyse syntaxique, on utilise en général une grammaire légèrement transformée :

- l'axiome n'apparaît jamais en membre droit de production
- la fin de flot est marquée par un caractère spécial : $

On remplace l'axiome S par un nouveau non-terminal S', et on ajoute S' -> S$

S -> Yd | ε
Y -> Sc | ε

devient

S' -> S$
S -> Yd | ε
Y -> Sc | ε

Les algorithmes d'analyse syntaxique peuvent utiliser k caractères (lexèmes) lus en avance sur le flot d'entrée, pour déterminer ce qu'ils doivent faire : ce sont les *caractères de prédiction.* Souvent, k=1.

Analyse descendante
LL(k) : *L*eft-to-right *L*eftmost derivation

Left-to-right : le flot de lexèmes est lu de la gauche vers la droite
Leftmost derivation : dans le membre droit de la production en cours d'utilisation,
on dérive d'abord le non-terminal le plus à gauche
k caractères de prédiction sont utilisés (pour nous, k=1)

On démarre de la production S -> S'$

Une pile est utilisée : elle contient γ, le mot sur l'alphabet des terminaux et des non-terminaux
qu'il reste à analyser pour une analyse réussie

Au départ, γ vaut « S ». Supposons que le caractère de prédiction soit *a* (la fin de flot est marquée par $)

Si $\gamma = X_1...X_n a\gamma'$, $n \geq 0$, et tous les X_i sont annulables
On consomme le prochain lexème et on remplace γ par γ'. Si a=$ l'analyse est terminée. Elle est réussie ssi γ' est vide
Si $\gamma = X_1...X_n Y\gamma'$, $n>0$, $a \in$ Premier(Y) et tous les X_i sont annulables
On remplace γ par $Y\gamma'$
Si $\gamma = X\gamma'$, $a \in$ Premier(X), X -> γ'' avec $a \in$ Premier(γ'')
On remplace γ par $\gamma''\gamma'$
On recommence

Si pendant l'analyse plusieurs choix sont possibles, ou aucun choix n'est possible, l'analyse échoue.

LL(1) en pratique

Par soucis d'efficacité, on ne programme pas le principe précédent tel quel.

On préfère traiter la grammaire *avant* de démarrer l'analyse, afin de garantir :

- qu'aucun choix multiple ne se produira pendant l'analyse
- que le choix de l'action à réaliser pendant l'analyse se fasse en temps constant

On construit une table qui indique, pour chaque terminal possible et pour chaque non-terminal X en sommet de pile, l'unique production X -> γ à utiliser si elle existe

Si cette table ne peut pas être construite, on dit que la grammaire n'est pas LL(1)

Pour construire la table T (lignes : les non-terminaux, colonnes : les terminaux):

-

Pour chaque *X* non terminal et *a* terminal

on calcule T[X,a] le plus petit ensemble vérifiant :

- Si $X \rightarrow \gamma \in P$ avec γ annulable, et $a \in$ Suivant(X)
  - $\gamma \in$ T[X,a]
- Si $X \rightarrow \gamma \in P$ avec $a \in$ Premier(γ)
  - $\gamma \in$ T[X,a]

Si |T[X,a]|>1 erreur

Suppression de la récursivité à gauche

Hypothèse de départ : pas d'ε-production (cf. cours de Théorie des langages pour les supprimer)

Récursivité gauche immédiate (cas simple (γ' ne commence pas par A))
A → Aγ | γ' produit γ', $\gamma'\gamma$, $\gamma'\gamma\gamma$, ... = $\gamma'\gamma^*$ et se transforme en A -> γ'B, B→ γB | ε

Récursivité gauche immédiate (cas général (les γ' ne commencent pas par A))
A → Aγ_1 | ... |Aγ_n |γ'_1|...|γ'_m se transforme en A → γ'_1B | … | γ'_mB, B -> γ_1B | … | γ_nB | ε

Algorithme général d'élimination de la récursivité gauche, immédiate ou non :
Soit G une grammaire sans cycle (A->A) et sans production vide (A->ε)
Commencer par ordonner les non-terminaux de G : A_1,…,A_n
Pour i allant de 1 à n
Pour j allant de 1 à i-1
Remplacer chaque production de la forme A_i -> $A_j\gamma$ par A_i → $\gamma'_1\gamma$ | … | $\gamma'_k\gamma$
quand A_j → γ'_1 | … | γ'_k sont toutes les productions dont A_j est membre gauche
Éliminer la récursivité gauche immédiate des productions dont A_i est membre gauche

Construction de la table LL(1)

Pour chaque *X* non terminal et *a* terminal
on calcule T[X,a] le plus petit ensemble vérifiant :

- Si X → γ ∈ P avec γ annulable, et a ∈ Suivant(X)
  γ ∈ T[X,a]
- Si X → γ ∈ P avec a ∈ Premier(γ)
  - γ ∈ T[X,a]

Si |T[X,a]|>1 erreur

S → E$
E → TE'
E' → +TE' | -TE' | ε
T → FT'
T' → *FT' | /FT' | ε
F → n | (E)

Annulable = { T', E' }
Les Premier se calculent facilement
Suivant(S) = { $ }
Suivant(E) = {), $ }
Suivant(E') = Suivant(E)
Suivant(T) = { +, -,), $ }
Suivant(T') = Suivant(T)
Suivant(F) = { *, /, +, -,), $ }

	n	+	-	*	/	(	)	$
S	E$					E$		
E	TE'					TE'		
E'		+TE'	-TE'				ε	ε
T	FT'					FT'		
T'		ε	ε	*FT'	/FT'		ε	ε
F	n					(E)		

P est l'ensemble des productions de la grammaire

Utilisation de la table LL(1)

X <- sommet de pile
a <- caractère courant dans le flot d'entrée
Tant que X!= $
    si X est un terminal ou $
        Si X = a
            dépiler X, avancer d'un caractère sur le flot d'entrée
        Sinon erreur
    sinon
        Si T[X,a] = $X_1...X_n$
            Dépiler X, empiler $X_n,\ldots,X_1$, émettre $X \rightarrow X_1...X_n$ en sortie
        Sinon erreur (car T[X,a] est vide)
Si la fin du flot n'est pas atteinte, erreur

	n	+	-	*	/	(	)	$
S	E$					E$		
E	TE'					TE'		
E'		+TE'	-TE'				ε	ε
T	FT'					FT'		
T'		ε	ε	*FT'	/FT'		ε	ε
F	n					(E)		

Entrée	Pile	Sortie
1+2*3$	S	
1+2*3$	E$	S→ E$
1+2*3$	TE'$	E→ TE'
1+2*3$	FT'E'$	T→ FT'
1+2*3$	1T'E'$	F→ n
+2*3$	T'E'$	
+2*3$	E'$	T'→ ε
+2*3$	+TE'$	E'→ +TE'
2*3$	TE'$	
2*3$	FT'E'$	T→ FT'
2*3$	2T'E'$	F→ n
*3$	T'E'$	
*3$	*FT'E'$	T'→ *FT'
3$	FT'E'$	
3$	3T'E'$	F→ n
$	T'E'$	
$	E'$	T'→ ε
$	$	E'→ ε

Fin (succès) !

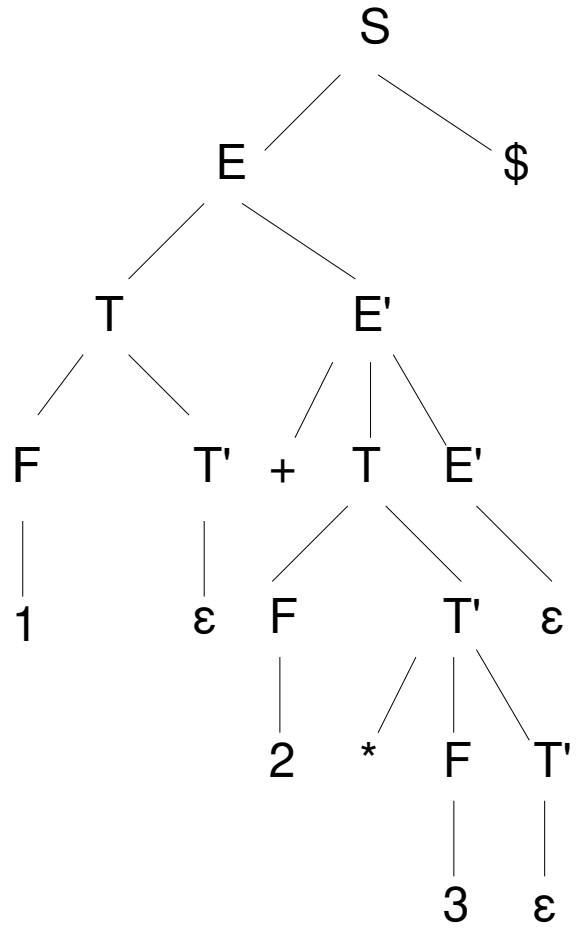
Utilisation de la table LL(1)

Entrée	Pile	Sortie
1+2*3$	S	
1+2*3$	E$	S→ E$
1+2*3$	TE'$	E→ TE'
1+2*3$	FT'E'$	T→ FT'
1+2*3$	1T'E'$	F→ n
+2*3$	T'E'$	
+2*3$	E'$	T'→ ε
+2*3$	+TE'$	E'→ +TE'
2*3$	TE'$	
2*3$	FT'E'$	T→ FT'
2*3$	2T'E'$	F→ n
*3$	T'E'$	
*3$	FT'E'$	T'→ FT'
3$	FT'E'$	
3$	3T'E'$	F→ n
$	T'E'$	
$	E'$	T'→ ε
$	$	E'→ ε

Fin (succès) !

La lecture de la sortie du haut vers le bas donne une dérivation gauche

L'arbre de dérivation peut être reconstruit à partir des changements de pile (sortie)



Programmation récursive de LL(1)

```
bool analyse_E(const char **s) {
  switch (**s) {
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9': return analyse_T(s) && analyse_Ep(s); /* E -> TE' */
  case '(': return analyse_T(s) && analyse_Ep(s); /* E -> TE' */
  default: return false;
  }
}
bool analyse_Ep(const char **s) {
  switch (**s) {
  case '+': case '-': ++*s; return analyse_T(s) && analyse_Ep(s); /* E' -> (+|-)TE' */
  case ')': case '$': return true; /* E' -> epsilon */
  default: return false;
  }
}
bool analyse_T(const char **s) {
  switch (**s) {
  case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9': return analyse_F(s) && analyse_Tp(s); /* T -> FT' */
  case '(': return analyse_F(s) && analyse_Tp(s); /* T -> FT' */
  default: return false;
  }
}
bool analyse_Tp(const char **s) {
  switch (**s) {
  case '*': case '/': ++*s; return analyse_F(s) && analyse_Tp(s); /* T' -> (*|/)FT' */
  case '+': case '-': case ')': case '$': return true; /* T' -> epsilon */
  default: return false;
  }
}
bool analyse_F(const char **s) {
  switch (**s) {
  case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9': ++*s; return true; /* F-> chiffre */
  case '(': ++*s; return analyse_E(s) && ( **s==')' ? ++*s, true : false ); /* F ->(E) */
  default: return false;
  }
}
```

Facile !
Pile de l'algorithme
= pile des appels récursifs

Conflits LL(1)

Une grammaire récursive gauche bien définie génère un conflit
E -> E+n | n : T[E,n] peut être E+E ou n
On peut toujours la transformer en grammaire récursive droite
E -> n+E | n

Une grammaire avec deux productions X -> γ et X -> γ' telles que Premier(γ)∩Premier(γ')≠∅ génère un conflit.
E -> n+E | n
Dans ce cas, il faut factoriser
E -> nY Y -> +E | ε

Problèmes :
- les grammaires obtenues par ces transformations sont peu naturelles, difficiles à comprendre, et les arbres de dérivations, les AST le seront aussi
- les grammaires LL(1) ont de fortes contraintes : il peut être très difficile, voire impossible, d'écrire une grammaire LL(1) pour un langage donné

Analyse ascendante
LR(k) : *L*eft-to-right *R*ightmost derivation

Left-to-right : le flot de lexèmes est lu de la gauche vers la droite
Rightmost derivation : dans le membre droit de la production en cours d'utilisation, on dérive d'abord le non-terminal le plus à droite
k caractères de prédiction sont utilisés (pour nous, k=1)

Le principe est de considérer toutes les productions possibles à un instant donné de l'analyse.
On matérialise, pour chacune de ces productions, où on en est dans son analyse par un • (*production marquée*)

E -> E•+E signifie que l'analyse d'une somme est en cours, que le membre gauche de la somme a déjà été analysée, et que le prochain caractère attendu sur l'entrée pour progresser est +.
Si le prochain lexème est +, alors quand il sera consommé on passera à E -> E+•E (*décalage par +*)
Sinon, le flot d'entrée n'est pas une somme et on abandonnera l'analyse de E-> E+E au profit d'autres productions

E -> •E+E signifie qu'on va tenter de dériver un E sous la forme de E+E, et que dans l'analyse de E+E rien n'a été encore lu

E -> E+E• signifie que les derniers caractères lus dans le flot d'entrée sont de la forme E+E, donc E (*réduction de E+E en E*)

Départ de l'analyse :
On démarre l'analyse de l'axiome : S -> •γ$ pour chaque production dont S est membre gauche
Idem pour tout non-terminal à gauche du point (opération de *fermeture*)

Fin de l'analyse :
S -> γ$• en fin de flot

Fermeture

Soit C un ensemble de productions marquées.

Fermeture de C : pour toute production de C où • est à la gauche d'un non-terminal X,
on ajoute à C toutes les productions dont X est membre gauche,
avec un • complètement à gauche.
on recommence jusqu'à stabilité du calcul de la fermeture

Exemple : Grammaire : S -> E$ E -> EOE | F O -> + | - | ε F -> n | (E)
C = { S -> •E$ }
Fermeture(C) = { S -> •E$, E -> •EOE, E -> •F, F-> •n, F-> •(E) }

Automate d'analyse LR

Les états sont des ensembles fermés de productions marquées compatibles avec l'analyse en cours

L'état initial est le plus petit contenant S -> •γ$ pour toute production S -> γ$ dont S est membre gauche (hypothèse : grammaire augmentée)

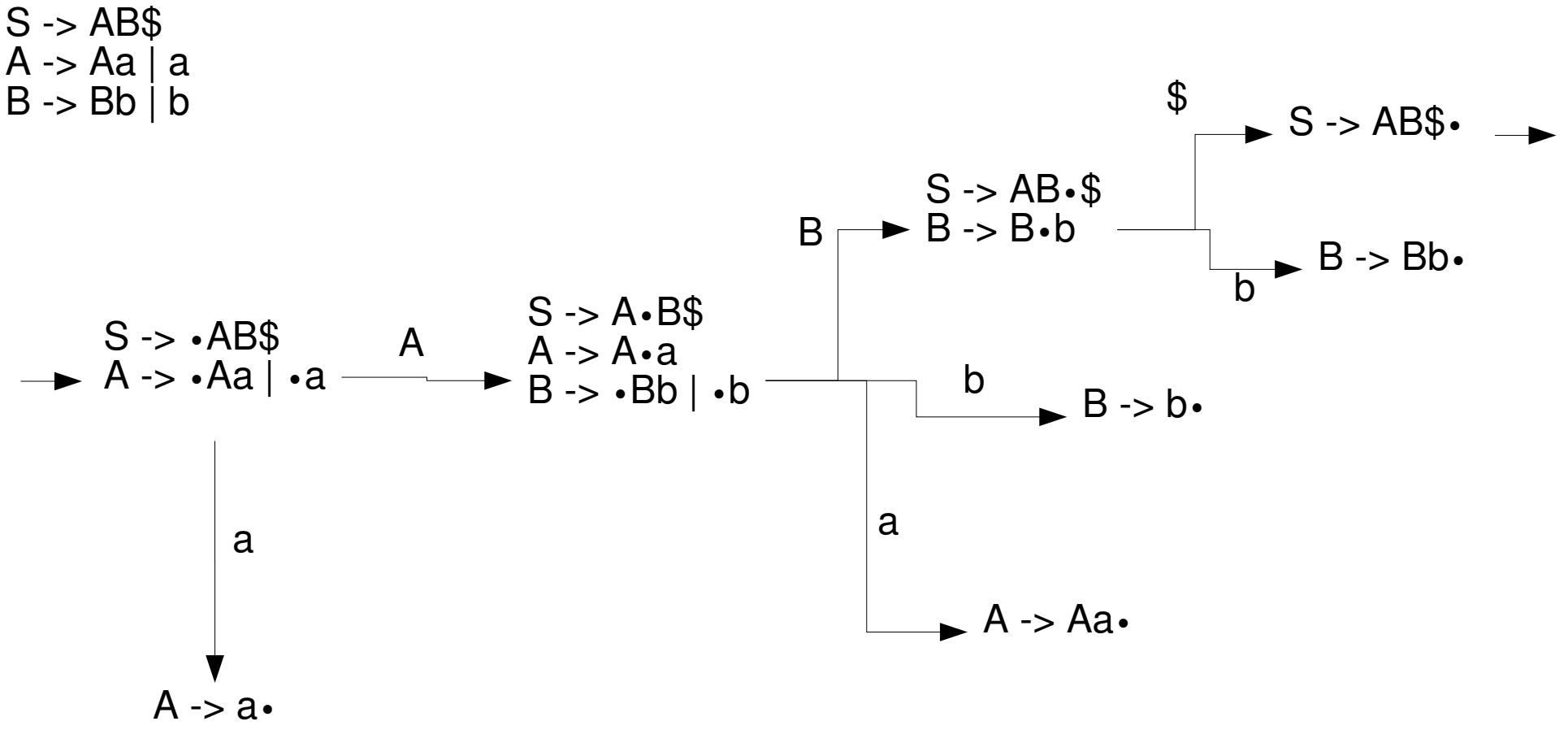
L'état final est celui contenant une production marquée S -> γ$• (hypothèse : grammaire augmentée)

Les états sont reliés par des actions :

- *décalage* par lecture d'un terminal ou non-terminal *X* :
  - similaire à une transition étiquetée *X,*
  - consomme *X* sur l'entrée si X est un terminal, ne consomme rien si X est un non-terminal
  - on passe d'un état contenant X -> γ•Xγ' vers un autre contenant X -> γX•γ'
- *réduction* :
  - ne consomme pas l'entrée
  - on passe d'un état contenant X -> γ• (signalant la fin de l'analyse de γ, donc de X), vers un autre contenant une transition marquée de la forme Y -> γ'X•γ''

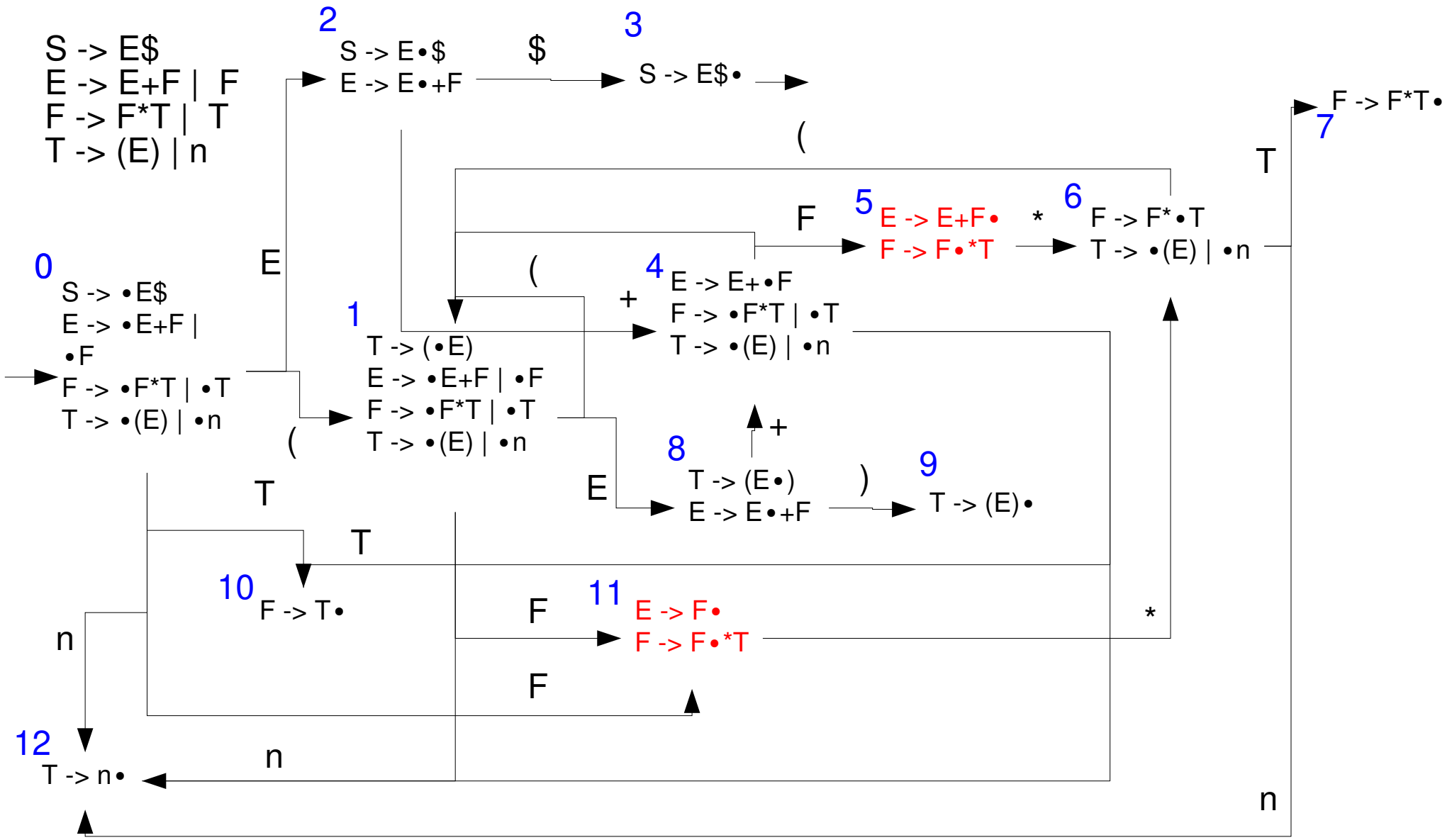
Le choix de l'action doit être déterminé de manière unique en fonction de ce qu'il y a à lire sur l'entrée (*grammaire LR*)

Exemple



Toutes les actions sont déterminées de manière unique en fonction de ce qu'il y a à lire, la grammaire est LR(0)

Exemple



Plusieurs états avec plusieurs possibilités d'action, la grammaire n'est pas LR(0)

Utilisation de l'automate d'analyse

Hypothèse : grammaire où

- les productions dont l'axiome S est membre gauche sont de la forme S -> γ$
- S n'apparaît pas dans le membre droit d'une production

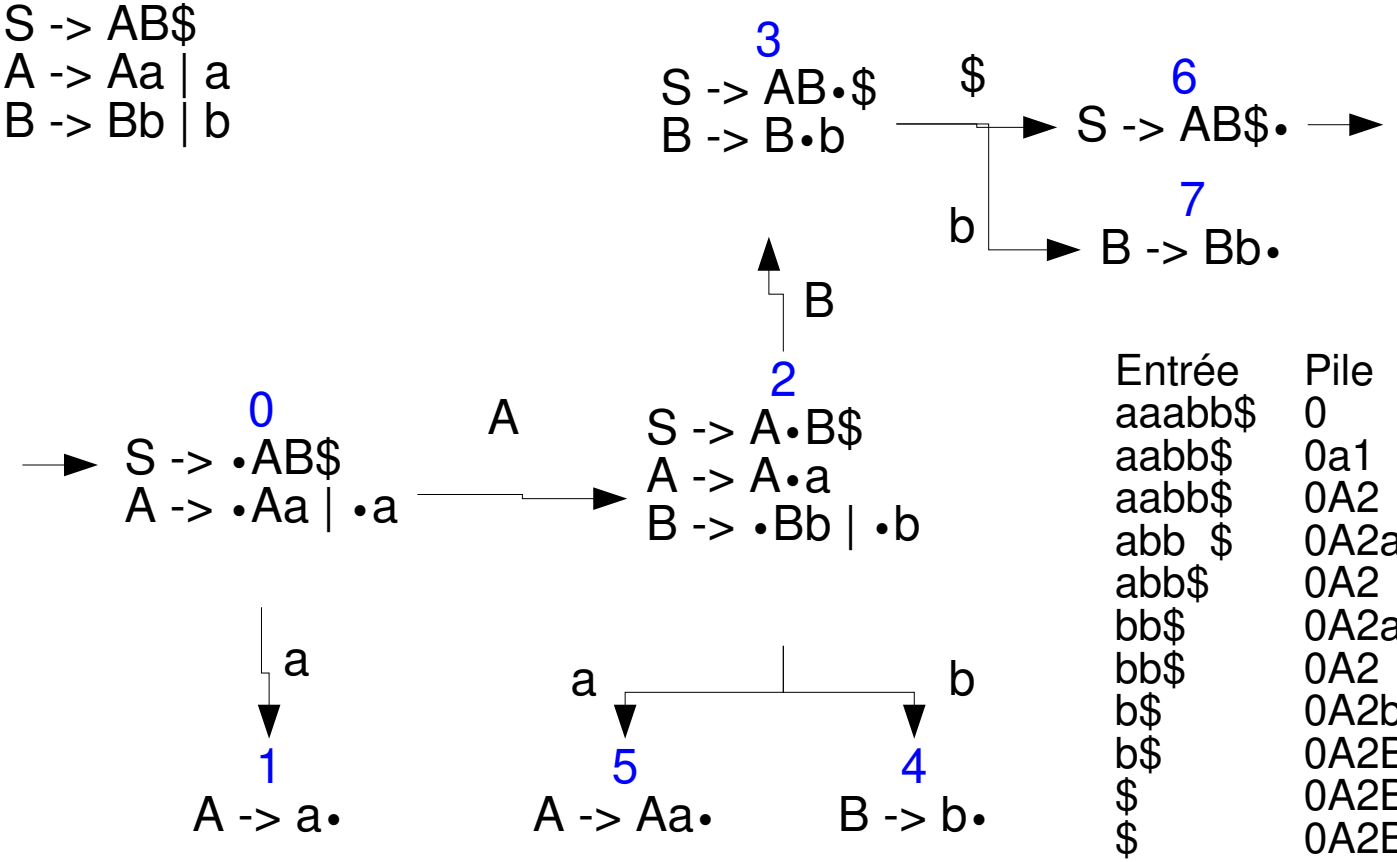
E: ensemble d'états de la machine LR

Une pile contenant un mot de $E((T \cup N)E)^*$

Au départ, la pile contient l'état initial 0
Tant qu'on ne réduit pas l'axiome
    x <- prochain symbole de l'entrée
    Si action(état de sommet de pile,x) = réduction par $X \rightarrow X_1 \ldots X_n$
        dépiler, dans l'ordre, $s_n, X_n, s_{n-1}, \ldots, s_1, X_1$
        empiler X
        empiler l'état pointé par la transition par X et partant de l'état en sommet de pile
    Sinon /* décalage */
        empiler x et l'état pointé par la transition par x et partant de l'état en sommet de pile
        (erreur s'il n'existe pas)
L'analyse est réussie si on réduit l'axiome tout en atteignant la fin du flot !

Exemple

S -> AB$
A -> Aa | a
B -> Bb | b



Entrée	Pile	Action
aaabb$	0	
aabb$	0a1	dec. 0 par a
aabb$	0A2	red. A->a
abb $	0A2a5	dec. 2 par a
abb$	0A2	red. A->Aa
bb$	0A2a5	dec. 2 par a
bb$	0A2	red. A->Aa
b$	0A2b4	dec. 2 par b
b$	0A2B3	red. B->b
$	0A2B3b7	dec. 3 par b
$	0A2B3	red. B->Bb
	0A2B3$6	dec. 3 par $

Acceptation, car il faut réduire S->AB$!

Conflits LR

Deux types de conflits (choix non déterministe d'action) :

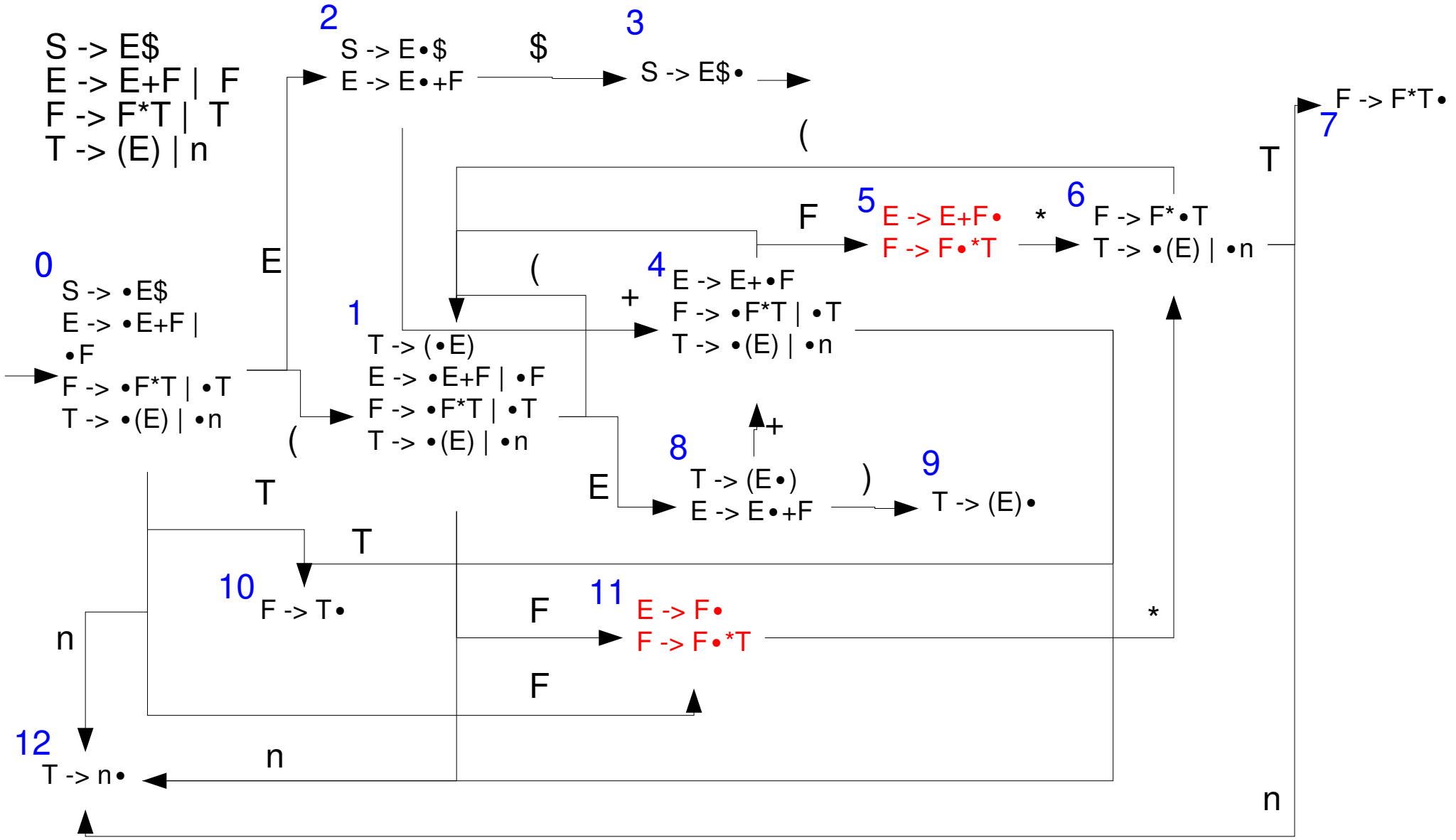
- *Réduction/réduction* (*reduce/reduce, R/R*):
  dans le même état il est possible de réduire deux productions différentes
- *Décalage/réduction* (*shift/reduce, S/R*) :
  un état contient à la fois une réduction et une possibilité de décalage

Stratégies pour résoudre les conflits : regarder ce qu'il y a à lire, jusqu'à k caractères (symboles de prédiction)

- SLR (Simple LR) : utilisation de Suivant(X) avec la même machine
  - en cas de conflit entre une réduction X ->γ• et une réduction X' -> γ'•,
    si l'intersection de Suivant(X) et Suivant(X') est vide , alors le conflit n'existera plus à l'exécution
  - en cas de conflit entre une réduction X ->γ• et un décalage X ->γ'•*c*γ"
    si $c \notin$ Suivant(X), décaler par *c*
- LR(k) : les *k* suivants possibles de chaque production sont stockés dans les états
- LALR(k) : même chose, avec simplification
- ...

Exemple

Suivant(E) = { +,), $ }
Suivant(F) = { +, *,), $ }
Suivant(T) = { +, *,), $ }



Les suivants permettent de résoudre les conflits, la grammaire est SLR

Exemple

Entrée	Pile	Action
1+2+3$	0	
+2+3$	0112	dec. 0 par 1
+2+3$	0T10	red. T->n
+2+3$	0F11	red. F->T
+2+3$	**0E2**	**red. E->F**
2+3$	0E2+4	dec. 2 par +
+3$	0E2+4212	dec. 4 par 2
+3$	0E2+4T10	red. T->n
+3$	0E2+4F5	red. F->T
+3$	**0E2**	**red E->E+F**
3$	0E2+4	dec. 2 par +
$	0E2+4312	dec. 4 par 3
$	0E2+4T10	red. T->n
$	0E2+4F5	red. F->T
$	**0E2**	**red E->E+F**
	0E2$	dec. 2 par $

Acceptation, car il faut réduire S->E$!

La lecture des réductions dans la colonne « action » du bas vers le haut donne une dérivation droite

L'arbre de dérivation peut être reconstruit à partir des actions

S -> **E**$ -> E+**F**$ -> E+**T**$ -> **E**+n$
-> E+**F**+n$ -> E+**T**+n$ -> **E**+n+n$
-> **F**+n+n$ -> **T**+n+n$ -> n+n+n$

Analyse LR(k), k>0

Même principe que LR(0), mais on mémorise pour chaque production dans les états, l'ensemble des mots de longueur ≤ k par lesquels la réduction de la production peut être suivie

Il s'agit d'un *raffinement* de la notion de suivant dans SLR, qui va donc permettre de résoudre plus de conflits : il y aura davantage de grammaires LR(1) que de grammaires SLR(1).

On note X -> γ, E la production de X en cours d'utilisation, dans un contexte où les suivants de X sont les éléments de E

Fermeture de C : pour toute production de X -> γ•Yγ', E, avec Y non-terminal,
on ajoute à C toutes les productions dont Y est membre gauche,
avec un • complètement à gauche, avec
- Premier(γ') comme contexte droit si γ n'est pas annulable
- Premier(γ')∪E comme contexte droit sinon

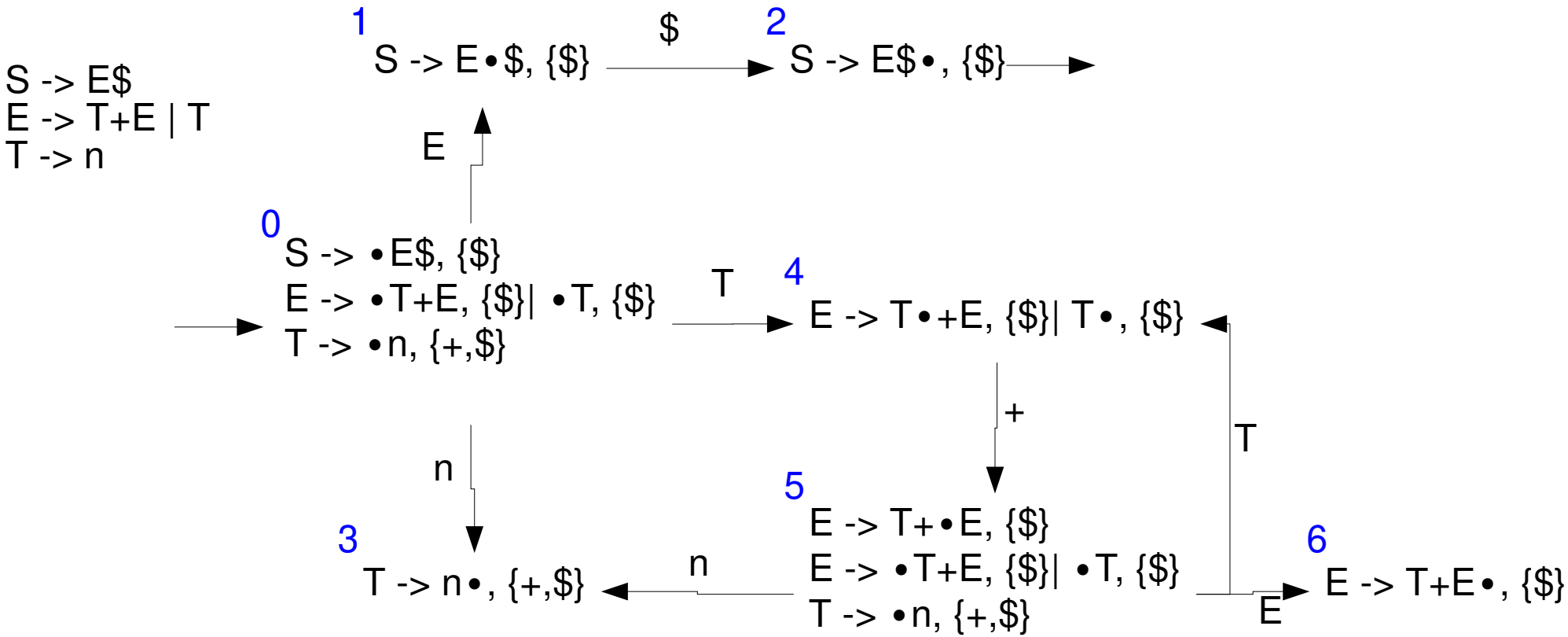
on recommence jusqu'à stabilité du calcul de la fermeture

Une transition d'un état vers un autre laisse invariants les contextes droits associés à une production.

Un état ne peut pas contenir deux fois la même production, avec le • au même endroit, avec des contextes droits différents : si X -> γ•Yγ', E et X -> γ•Yγ', E' avec E!=E' sont des productions devant apparaître dans le même état, alors c'est X -> γ•Yγ', E∪E' qu'il faut mettre dans l'état

$ est considéré comme suivant de l'axiome

Exemple



Cette grammaire n'est pas LR(0)
car en retirant les contextes droits de chaque production, il existe un conflit S/R

Elle est LR(1)
car les suivants de chaque production permettent de résoudre le conflit S/R :
- dans l'état 4, il n'y a pas de • avant un $: si le caractère à venir est $, on réduit, sinon on décale.

Exemple

Entrée	Pile	Action
1+2+3$	0	
+2+3$	013	déc. 0 par 1
+2+3$	0T4	red. T->n
2+3$	0T4+5	déc. 4 par +
+3$	0T4+523	déc. 5 par 2
+3$	0T4+5T4	red. T->n
3$	0T4+5T4+5	déc. 4 par +
$	0T4+5T4+533	déc. 5 par 3
$	0T4+5T4+5T4	red. T->n
$	0T4+5T4+5E6	red. E->T
$	0T4+5E6	red. E->T+E
$	0E1	red. E->T+E
	0E1$2	déc. 1 par $

Acceptation, car il faut réduire S->E$!

LR(0), SLR(1), LR(1), ...

LR(0) :

Peu de grammaires « naturelles » (langages de programmation, etc)

Nombre d'états de la machine = $2^{M|P|}$ en théorie dans le pire des cas,

acceptable en pratique (M=plus grande taille de membre droit)

SLR(1) :

Un peu plus de grammaires « naturelles » que LR(0), mais toujours insuffisant

Nombre d'états : pareil que pour LR(0)

Taille des suivants : |N||T|

LR(1) :

La très grande majorité des grammaires « naturelles »

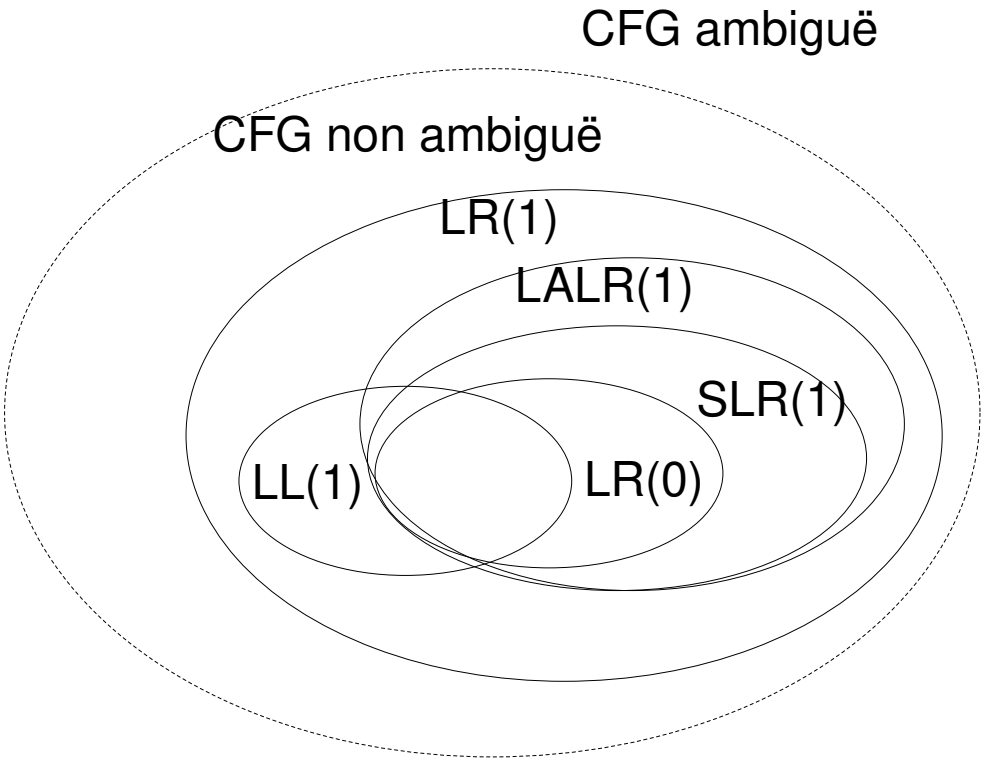
Nombre d'états : $2^{M|P|2\hat{}|T|}$ trop important

LALR(1) : *LookAhead LR*

Simplification de la machine LR(1) pour rendre le nombre d'états acceptable

Une grosse majorité de grammaires « naturelles »

LR(0), SLR(1), LR(1), ...



Décidable

Indécidable

Grammaire LL(1) et non LALR(1)

S -> aA | bB
A -> Ca | Db
B -> Cb | Da
C -> E
D -> E
E -> ε

Analyse LALR(1)

Idée : simplification de la machine LR(1) par

- fusion des états contenant exactement les mêmes productions avec le point à la même position (sans regarder les contextes droits associés aux productions)
- pour chaque production, les contextes droits sont l'union des contextes droits de la production dans les états qui sont fusionnés

Machine LR(1)

$X_1 \rightarrow \gamma_1 \bullet Y_1 \gamma'_1, E_1$
…
$X_n \rightarrow \gamma_n \bullet Y_n \gamma'_n, E_n$

$X_1 \rightarrow \gamma_1 \bullet Y_1 \gamma'_1, E'_1$
…
$X_n \rightarrow \gamma_n \bullet Y_n \gamma'_n, E'_n$

Fusion →

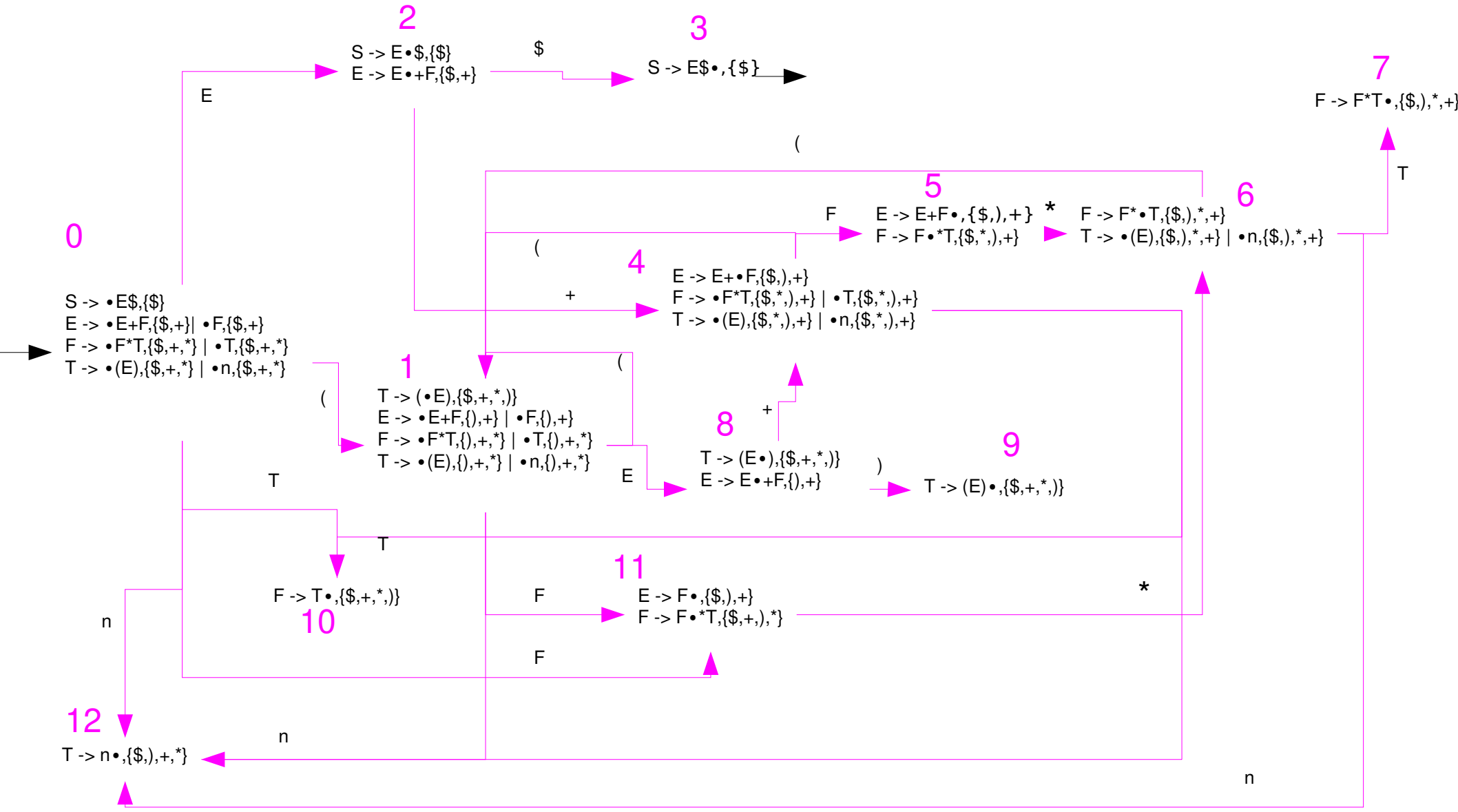
Machine LALR(1)

$X_1 \rightarrow \gamma_1 \bullet Y_1 \gamma'_1, E_1 \cup E'_1$
…
$X_n \rightarrow \gamma_n \bullet Y_n \gamma'_n, E_n \cup E'_n$

- Il s'agit d'un raffinement de SLR(1), mais d'une stratégie plus grossière que LR(1)
- Le nombre d'états de la machine LALR(1) est celui de la machine LR(0)
- La machine LALR(1) peut aussi se construire sans construire la machine LR(1) (compliqué)

S -> E$
E -> E+F | F
F -> F*T | T
T -> (E) | n

Exemple



A partir de tout état, une seule action possible : la grammaire est LALR(1)

Analyse montante et AST

La suite des actions réalisées pendant une analyse montante correspond à un parcours en profondeur et post-fixe de l'arbre de dérivation :

- décalage (terminal) : passage au frère droit, visite d'une feuille
- réduction : passage au père après les fils, visite d'un nœud interne (non-terminal)

- On peut en profiter pour construire très facilement un AST, ou reconstruire l'arbre de dérivation en mémoire

  - En supposant qu'il existe un mécanisme qui permet de calculer et de retourner quelque chose pour chaque action, décalage ou réduction, qui permet, au moment où on dérive X->γ, de récupérer la valeur calculée pour chaque terminal et non-terminal de γ. Bison possède un tel mécanisme.

Grammaire : E -> E+E | n Analyse de 1+2

décalage par '1'	r<-nouveau nœud(1) ; retourner r
réduction E->n	retourner le nœud issu de l'analyse de n
décalage par '+'	r<-nouveau nœud(+) ; retourner r
décalage par '2'	r<-nouveau nœud(2) ; retourner r
réduction E->n	retourner le nœud issu de l'analyse de n
réduction E->E+E	r<-nouveau nœud dont les fils sont issus de l'analyse de E, de + et de E

Mécanismes de résolution de conflits

Les conflits peuvent être résolus en transformant la grammaire
- peut être difficile
- peut rendre la grammaire compliquée, peu naturelle, et rendre difficile la construction de l'AST

Des mécanismes peuvent aussi être intégrés à l'analyse montante pour prioriser décalages et réductions en cas de conflit.

Exemple : E -> E+E | n

La machine LR a un état contenant E->E+E• et E->E•+E :
- conflit S/R car + est dans les suivants de E+E

Si on souhaite + associatif à gauche, il faut privilégier la réduction sur le décalage, sinon l'inverse
- attribuer des associativités aux terminaux permet de résoudre des conflits S/R

Exemple : E -> E+E | E*E | n

La machine LR a un état contenant E->E+E• et E->E•*E :
- conflit S/R car * est dans les contextes droits de E+E

Si on souhaite * prioritaire sur +, il faut privilégier le décalage sur la réduction, sinon l'inverse
- attribuer des priorités aux terminaux permet de résoudre des conflits S/R

Mécanismes de résolution de conflits

Une grammaire ambiguë n'est pas LR

Un conflit R/R est souvent signe que la grammaire est mal écrite : il faut la refaire

Souvent, les analyseurs LR disposent de mécanismes de résolution de conflits S/R par priorité et associativité.

- on fixe priorité et associativité aux terminaux
- on en déduit des priorités pour les productions :
  priorité production = priorité du dernier terminal de la production
- on peut aussi parfois définir explicitement une priorité pour les non-terminaux
- Résolution du conflit
  - on compare la priorité de la production par rapport à celle du terminal de décalage
    - si elle est supérieure, on réduit
    - si elle est inférieure, on décale
    - à priorité égale on regarde l'associativité
      - associativité à gauche on réduit
      - associativité à droite on décale

Ces mécanismes permettent d'utiliser des grammaires non LR « naturelles » dans des analyseurs LR, comme par exemple E-> E+E | E*E | n

Traiter les erreurs

Il peut être simple de récupérer les erreurs de syntaxe dues à l'absence d'action de décalage sur le terminal en entrée.

Les analyseurs LR ont deux manières principales de traiter les erreurs :

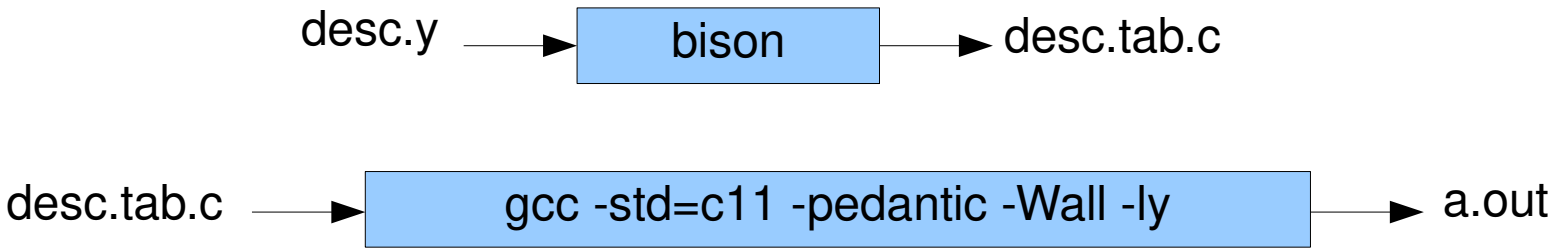
- par insertion/suppression automatique de terminaux, pour tenter de récupérer l'erreur
  - Exemple : on lit (*3) pour E->(E). au moment de la lecture de *, l'analyseur attend un nombre ou une (. L'analyseur peut décider d'afficher le message « nombre attendu », insérer un terminal « nombre » dans le flot et continuer
- par utilisation d'un non-terminal *error* dans la grammaire
  - la grammaire prévoit les erreurs syntaxiques sur l'entrée par règles de la forme
    E -> (error)
    signifie que si une erreur de syntaxe intervient après une '(', il faut
    - abandonner la partie de l'analyse qui a eu lieu juste après le décalage par '('
      - troncature de pile
    - lire l'entrée jusqu'à une ')'
    - reprendre l'analyse

Bison

Analyseur LR(1), LALR(1), GLR, …

Principe :

- Construit un analyseur LALR(1) à partir de la description d'une grammaire contenue dans un fichier texte (extension usuelle : .y)
- Produit du code C implantant l'analyseur
- L'analyseur récupère les terminaux un par un en appelant une fonction d'analyse lexicale
  - int yylex() // Retourne un identifiant>0 de terminal, 0 si il n'y a plus rien à lire
  - Elle peut être écrite directement par le programmeur
  - Elle peut aussi être celle de flex !
- Du code C peut être inséré dans les règles de grammaire : il sera exécuté au fur et à mesure des décalages et des réductions



Un premier exemple

```
%{
  #include <ctype.h>
  #include <stdio.h>
  int yylex();
  void yyerror (char const *);
%}
%token NUMBER;
%%
S: S E '\n' | S '\n' | E '\n' | '\n';

E : E '+' F | F;
F : F '*' T | T;
T: '(' E ')' | NUMBER;
%%

int  yylex() {
  int c;
  while ( (c=getchar())!= EOF && isblank(c) ) // eat spaces
    ;
  if ( c == EOF )
    return 0; // Nothing left to read
  if ( isdigit(c) ) {
    while ( (c=getchar())!= EOF && isdigit(c) ) // eat other digits
      ;
    ungetc(c, stdin);
    return NUMBER;
  } else {
    return c;
  }
}
```

```
void yyerror (char const *s) {
  fprintf(stderr, "%s", s);
}
```

```
bison expr.y
gcc -std=c11 -pedantic -Wall expr.tab.c
./a.out
1+2+3
4
*5
syntax error
```

Voir la machine générée : mode texte

option -v de bison
bison expr.y -v
génère aussi expr.output

Extrait

État 6
6 E: F .
7 F: F . '*' T
'*' décalage et aller à l'état 14
$défaut réduction par utilisation de la règle 6 (E)

État 7
8 F: T .
$défaut réduction par utilisation de la règle 8 (F)

Voir la machine générée mode graphique

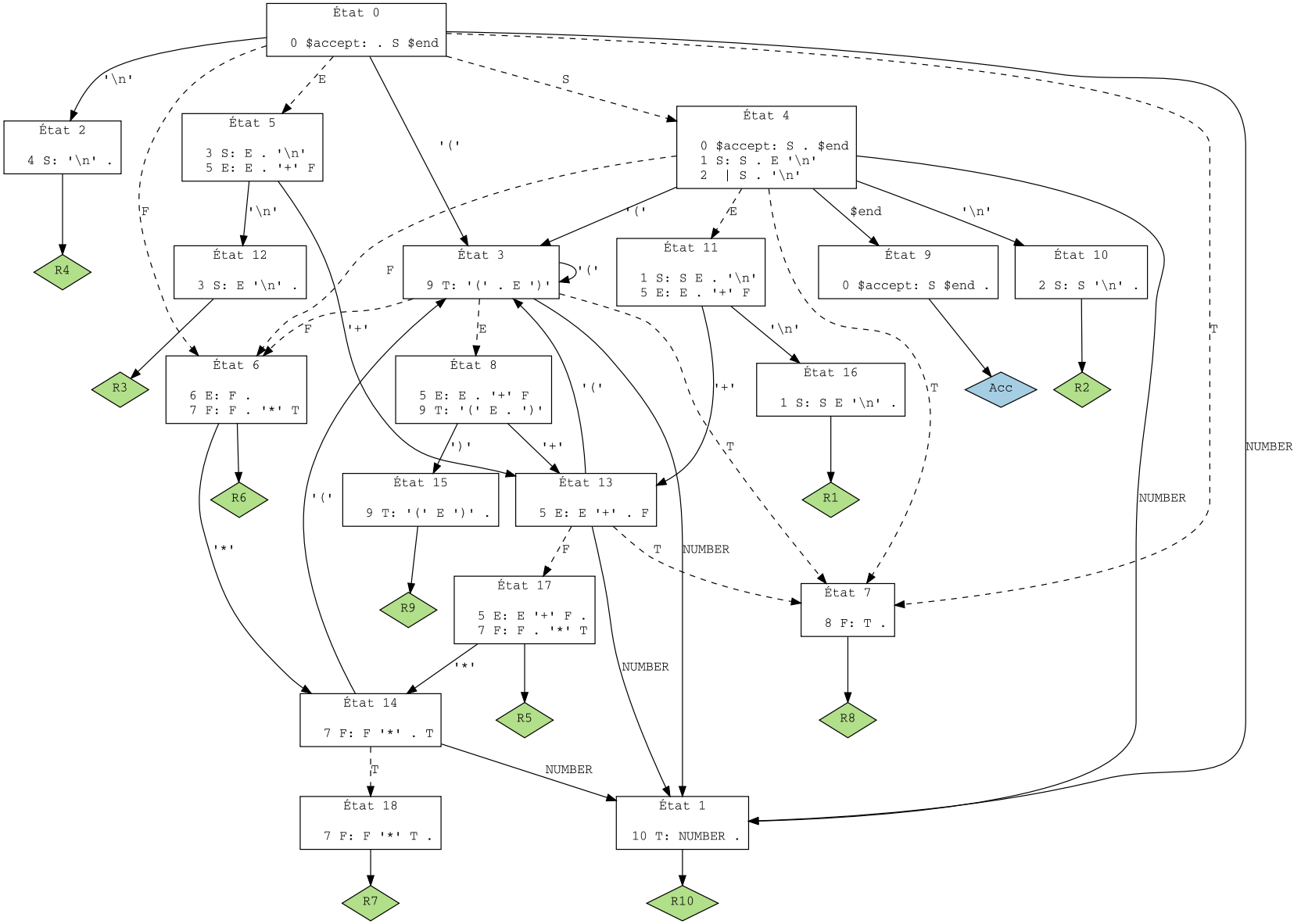
option --graph de bison

```
bison expr.y --graph
```

génère aussi expr.dot

Pour ouvrir un fichier au format dot, on peut employer par exemple xdot

xdot


État 0
0 $accept: . S $end
État 2
4 S: '\n' .
État 5
3 S: E . '\n'
5 E: E . '+' F
État 4
0 $accept: S . $end
1 S: S . E '\n'
2 | S . '\n'
État 12
3 S: E '\n' .
État 3
9 T: '(' . E ')'
État 11
1 S: S E . '\n'
5 E: E . '+' F
État 9
0 $accept: S $end .
État 10
2 S: S '\n' .
R4
R3
État 6
6 E: F .
7 F: F . '*' T
État 8
5 E: E . '+' F
9 T: '(' E . ')'
État 16
1 S: S E '\n' .
Acc
R2
R6
État 15
9 T: '(' E ')' .
État 13
5 E: E '+' . F
R1
R9
État 17
5 E: E '+' F .
7 F: F . '*' T
État 7
8 F: T .
R5
R8
État 14
7 F: F '*' . T
État 18
7 F: F '*' T .
État 1
10 T: NUMBER .
R7
R10
'\n'
E
S
'('
F
'+'
')'
'*'
T
NUMBER
$end

Voir ce que fait bison à l'exécution

```
%{
  #include <ctype.h>
  #include <stdio.h>
  int yylex();
  void yyerror (char const *);
  // Activer dynamiquement la trace
  int yydebug = 1 ;
%}
%token NUMBER;
%%
S: S E '\n' | S '\n' | E '\n' | '\n';

E : E '+' F | F;
F : F '*' T | T;
T: '(' E ')' | NUMBER;
%%
…
```

Compiler le code pour tracer :
gcc -DYYDEBUG y.tab.c

```
Starting parse
Entering state 0
Reading a token: Next token is token NUMBER ()
Shifting token NUMBER ()
Entering state 1
Reducing stack by rule 10 (line 14):
   $1 = token NUMBER ()
-> $$ = nterm T ()
Stack now 0
Entering state 7
Reducing stack by rule 8 (line 13):
   $1 = nterm T ()
-> $$ = nterm F ()
Stack now 0
Entering state 6
Reading a token: Next token is token '+' ()
Reducing stack by rule 6 (line 12):
   $1 = nterm F ()
-> $$ = nterm E ()
Stack now 0
Entering state 5
Next token is token '+' ()
Shifting token '+' ()
Entering state 13
Reading a token: Next token is token NUMBER ()
Shifting token NUMBER ()
Entering state 1
Reducing stack by rule 10 (line 14):
   $1 = token NUMBER ()
-> $$ = nterm T ()
Stack now 0 5 13
Entering state 7
Reducing stack by rule 8 (line 13):
   $1 = nterm T ()
-> $$ = nterm F ()
Stack now 0 5 13
Entering state 17
Reading a token: Next token is token '\n' ()
Reducing stack by rule 5 (line 12):
   $1 = nterm E ()
   $2 = token '+' ()
   $3 = nterm F ()
-> $$ = nterm E ()
Stack now 0
Entering state 5
Next token is token '\n' ()
Shifting token '\n' ()
Entering state 12
Reducing stack by rule 3 (line 10):
   $1 = nterm E ()
   $2 = token '\n' ()
-> $$ = nterm S ()
Stack now 0
Entering state 4
Reading a token: Now at end of input.
Shifting token $end ()
Entering state 9
Stack now 0 4 9
Cleanup: popping token $end ()
Cleanup: popping nterm S ()
```

Format d'un fichier bison

%{
    Code C
%}
Définitions bison

%%
production$_1$; // Les productions peuvent être annotées par du code C
…
production$_n$;
%%

Code C

Même exemple en utilisant flex

```
%{ /* expr2.l */
  #include "expr2.tab.h"
%}
%option noyywrap
%%
[[:digit:]]+      return NUMBER;
[[:blank:]]
.|\n              return yytext[0];
%%
```

```
%{ /* expr2.y */
  #include <ctype.h>
  #include <stdio.h>
  int yylex();
  void yyerror (char const *);
%}
%token NUMBER
%%
S: S E '\n' | S '\n' | E '\n' | '\n';

E : E '+' F | F;
F : F '*' T | T;
T: '(' E ')' | NUMBER;
%%

void yyerror (char const *s) {
  fprintf(stderr, "%s", s);
}
```

Avec l'option -d, bison génère, en plus de expr2.tab.c,
un fichier expr2.tab.h contenant entre autres les
définitions C pour les tokens (terminaux) qu'il utilise

```
bison -d expr2.y
flex -D_POSIX_SOURCE -DYY_NO_INPUT --nounput expr2.l
gcc -std=c11 -pedantic -Wall expr2.tab.c -c
gcc -std=c11 -pedantic -Wall lex.yy.c -c
```

L'option -ly permet d'utiliser le main de la bibliothèque bison
Attention : pas le même comportement que le main
par défaut de flex

```
gcc expr2.tab.o lex.yy.o -ly
```

Un Makefile s'impose

```
SHELL=/bin/sh
LEX=flex
YACC=bison
CC=gcc
CFLAGS=-g -std=c11 -pedantic -Wall
LDFLAGS=-ly
# --nounput: ne génère pas la fonction yyunput() inutile
# --DYY_NO_INPUT: ne prend pas en compte la fonction input() inutile
# -D_POSIX_SOURCE: déclare la fonction fileno()
LEXOPTS=-D_POSIX_SOURCE -DYY_NO_INPUT --nounput
YACCOPTS=

PROG=expr2

$(PROG): lex.yy.o $(PROG).tab.o
	$(CC) $+ -o $@ $(LDFLAGS)

lex.yy.c: $(PROG).l $(PROG).tab.h
	$(LEX) $(LEXOPTS) $<

lex.yy.h: $(PROG).l
	$(LEX) $(LEXOPTS) --header-file=$@ $<

$(PROG).tab.c $(PROG).tab.h: $(PROG).y lex.yy.h
	$(YACC) $(YACCOPTS) $< -d -v

%.o: %.c
	$(CC) $(CFLAGS) $< -c

clean:
	-rm $(PROG) *.o lex.yy.* $(PROG).tab.* *.err *.output *.out *.dot
```

Autre exemple

```
%{
  #include <ctype.h>
  #include <stdio.h>
  int yylex();
  void yyerror (char const *);
%}
%token NUMBER
%%
ligne : ligne E '\n' |   ligne '\n' | E '\n' | '\n';
E : E '+' E | E '*' E | NUMBER;
%%

void yyerror (char const *s) {
  fprintf(stderr, "%s", s);
}
```

```
bison  expr3.y -v
expr3.y: avertissement: 4 conflits par
        décalage/réduction [-Wconflicts-sr]
Extrait de expr3.output
État 12 conflits: 2 décalage/réduction
État 13 conflits: 2 décalage/réduction
...
État 12

    5 E: E . '+' E
    5  | E '+' E .
    6  | E . '*' E

    '+'  décalage et aller à l'état 9
    '*'  décalage et aller à l'état 10

    '+'       [réduction par utilisation de la règle 5 (E)]
    '*'       [réduction par utilisation de la règle 5 (E)]
    $défaut  réduction par utilisation de la règle 5 (E)
...
```

Levée des ambiguïtés : mécanisme de priorisation/associativité de bison

```
%{
  #include <ctype.h>
  #include <stdio.h>
  int yylex();
  void yyerror (char const *);
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%%
ligne : ligne E '\n' |   ligne '\n' | E '\n' | '\n';
E : E '+' E | E '*' E | E '-' E | E '/' E | NUMBER;
%%

void yyerror (char const *s) {
  fprintf(stderr, "%s", s);
}
```

- +,-,*,/ sont tous déclarés associatifs à gauche
- + et – ont une priorité identique
- * et / ont une priorité identique
- la priorité de + et – est inférieure à celle de * et /

ordre croissant de priorité

- 1+2+3 est analysé comme (1+2)+3 [associativité à gauche de +]
  - on favorise la réduction de E->E+E par rapport au décalage par +
- 1+2-3 est analysé comme (1+2)-3 [associativité à gauche de +]
  - on favorise la réduction de E->E+E par rapport au décalage par -
- 1+2*3 est analysé comme 1+(2*3) [priorité de * sur +]
  - on favorise le décalage par * sur la réduction E->E+E

Levée des ambiguïtés : mécanisme de priorisation/associativité de bison

Pour attribuer des priorités/associativités aux terminaux :

- %left : associativité à gauche
  - dans le conflit S/R on favorise R
- %right : associativité à droite
  - dans le conflit S/R on favorise S
- %nonassoc : pas d'associativité
  - Déclaration à bison que le conflit, même s'il existe dans la machine LALR(1), ne sera jamais utilisé à run-time.
  - Le conflit n'est plus indiqué par bison.
  - Si, au cours de l'exécution de la machine LALR(1), on se trouve dans l'état du conflit et que le prochain symbole d'entrée est celui du conflit, production d'une erreur de syntaxe
- %precedence : sert à déclarer une priorité sans fournir d'information d'associativité

Levée des ambiguïtés : mécanisme de priorisation/associativité de bison

Bison se sert des priorités des terminaux pour attribuer des priorités aux productions
= priorité du terminal le plus à gauche dans son membre droit

Pour résoudre un conflit S/R, on compare la priorité de la production avec celle du terminal à suivre
- plus forte : on réduit
- plus faible : on décale

Il est possible d'imposer à bison une priorité pour une production

```
%left '+' '-'
%left '*' '/'
%left OPUNAIRE
%%
E : E '+' E | E '-' E | E '*' E | E '/' E | '-' E %prec OPUNAIRE | '+' E %prec OPUNAIRE;
%%
```

Ici, on déclare un terminal (factice, car jamais produit par l'analyseur lexical) OPUNAIRE, dont la priorité est plus forte que celle des autres opérateurs
Le %prec dans la production E -> '-'E indique que cette production a la priorité la plus forte, ce qui favorise la réduction sur le décalage

Du code C dans les productions

- Peut apparaître n'importe où, mais pour l'instant on en mettra seulement en fin de production
- Le code C est exécuté quand le • dans la production est juste devant le code
- Donc, si le code est en fin de production, au moment de la réduction de la production

En changeant un peu les exemples précédents

```
%left '+' '-'
%%
ligne : ligne E '\n' |   ligne '\n' | E '\n' | '\n';
E : E '+' E     { printf("Oh, une somme !\n");       }
| E '-' E        { printf("Ah, une différence !\n");   }
| NUMBER    { printf("Tiens, un nombre !\n");    }
;
%%
```

Compilation, exécution :

```
1+2-3
Tiens, un nombre !
Tiens, un nombre !
Oh, une somme !
Tiens, un nombre !
Ah, une différence !
```

En changeant l'associativité :

```
%right '+' '-'
...
```

Compilation, exécution :

```
1+2-3
Tiens, un nombre !
Tiens, un nombre !
Tiens, un nombre !
Ah, une différence !
Oh, une somme !
```

Attribut d'un lexème

La fonction yylex() retourne
- la catégorie de la prochaine unité lexicale sous la forme d'un entier (int) >0
- 0 s'il n'y a plus rien à lire

L'attribut de l'unité lexicale est stocké dans une variable globale yylval de type YYSTYPE
- par défaut YYSTYPE est int
- il peut être changé par
  - un typedef (à éviter)
  - un #define (à éviter)
  - %define api.value.type
  - %union quand YYSTYPE doit être une union

Au niveau de bison, le $i^{ème}$ élément du membre droit d'une production a le type YYSTYPE et la valeur $i. Quand c'est un terminal, $i est l'attribut de l'unité lexicale.

Fixer l'attribut d'un lexème dans flex

```
%{
  #include <math.h>
  #include "expr3.tab.h" // Contient la définition de YYSTYPE à partir de bison
  void string_to_double(double *v, const char *s);
%}
%option noyywrap
%%
[[:digit:]]+  { string_to_double(&yylval, yytext); return NUMBER; }
[[:blank:]]
.|\n                    return yytext[0];
%%

void string_to_double(double *v, const char *s) {
  char *p;
  errno = 0;
  *v = strtod(s, &p);
  if ( *p != '\0' || ( errno == ERANGE && ( *v == HUGE_VALF || *v == -HUGE_VALF ) ) ) {
    fprintf(stderr, "Error converting string to double\n");
    exit(EXIT_FAILURE);
  }
}
```

Les valeurs des terminaux et non-terminaux dans bison

```
%{
  #include <stdio.h>
  int yylex();
  void yyerror (char const *);
%}
%define api.value.type {double}
%token NUMBER
%right '+' '-'
%%
ligne : ligne E '\n' |   ligne '\n' | E '\n' | '\n';
E : E '+' E {
  printf("Oh, une somme (%f+%f=%f)!\n", $1, $3, $1+$3);
  $$=$1+$3;
 }
| E '-' E   {
  printf("Ah, une différence (%f-%f=%f)!\n", $1, $3, $1-$3);
  $$=$1-$3;
  }
| NUMBER    {
  printf("Tiens, un nombre %f!\n", $1);
  $$=$1;
  }
;
%%
void yyerror (char const *s) {
  fprintf(stderr, "%s", s);}
```

Les valeurs des terminaux et non-terminaux dans bison

Les terminaux et non-terminaux peuvent avoir plusieurs types : en C, YYSTYPE devient alors une union de types.

Directive bison :

```
%union {
	double dValue;
	node_type *nValue;
}
```

Deux types possibles : double ou node_type *

yylval.dValue a le type double
yylval.nValue a la type node_type *

Le type des terminaux et non terminaux se déclare alors dans bison par

```
%type<dValue> NUMBER /* NUMBER  a le type de dValue (=double)          */
%type<nValue> E      /* E       a le type de nValue (=node_type *)    */
```

Quand l'union contient un type non standard du C, on définit ce type dans un fichier .h, qu'on inclut dans les fichiers bison et flex

Construction de l'AST

```
#ifndef _SIMPLE_TREE_TYPES_H
#define _SIMPLE_TREE_TYPES_H

typedef enum { NO_OP, ADD_OP, SUB_OP, MUL_OP, DIV_OP } op_type;

typedef struct _node {
  struct _node *left, *right;
  op_type op;
  double val;
} node_type;

#endif
```

Construction de l'AST

```
%{
  #include <stdlib.h>
  #include <stdio.h>
  #include "simpleTreeTypes.h"
  int yylex();
  void yyerror (char const *);
  static node_type *node_alloc(const node_type *);
  static void infix_node_printer(FILE *out, const node_type *p);
%}
%union {
  double dValue;
  node_type *nValue;
}
%token NUMBER
%left '+' '-'
%left '*' '/'
%type<dValue> NUMBER
%type<nValue> E
%%
sligne : sligne ligne | %empty ;
ligne : E '\n' {
  infix_node_printer(stdout, $1);
 }
| '\n'
;
```

```
E : E '+' E {
  node_type n = {
        .left=$1, .right=$3,
        .op=ADD_OP, .val=0.0 } ;
  $$ = node_alloc( &n );
 }
...
| NUMBER    {
  node_type n = {
        .left = NULL, .right = NULL,
        .op = NO_OP, .val = $1 } ;
  $$ = node_alloc( &n );
 }
;
%%

void yyerror (char const *s) {
  fprintf(stderr, "%s", s);
}
```

Construction de l'AST

```
%{
  #include <math.h>
  #include "simpleTreeTypes.h"       // Contient la définition du type des nœuds
  #include "simpleTree.tab.h"       // Contient la définition de YYSTYPE à partir de bison
  void string_to_double(double *v, const char *s);
%}
%option noyywrap
%%
[[:digit:]]+ { string_to_double(&yylval.dValue, yytext); return NUMBER; }
[[:blank:]]
.|\n         return yytext[0];
%%

void string_to_double(double *v, const char *s) {
  char *p;
  errno = 0;
  *v = strtod(s, &p);
  if ( *p != '\0' || ( errno == ERANGE && ( *v == HUGE_VALF || *v == -HUGE_VALF ) ) ) {
    fprintf(stderr, "Error converting string %s to double\n", s);
    exit(EXIT_FAILURE);
  }
}
```

Analyse sémantique

Elle donne une couche de sens à ce qui a été syntaxiquement analysé.

Détection d'autres erreurs :

- Typage statique

```
int add(int x, int y) ;
…
void f(void) {
        bool b ;
        add(b) ; /* Syntaxiquement correct, mais le nombre d'arguments est incorrect  */
                 /* Syntaxiquement correct, mais l'argument n'est pas du bon type       */
                 /* Syntaxiquement correct, mais b est utilisé avant d'avoir été initialisé */
                /* Syntaxiquement correct, mais valeur de retour non utilisée          */
}
```

- Gestion de la portée des identificateurs
- Rejet de constructions syntaxiquement correctes mais auxquelles on ne peut attribuer de sens
- Détection de variables non initialisées avant utilisation
- Détection de code inaccessible (code mort)

Préparation finale des structures de données avant la phase de génération de code

Certaines étapes de l'analyse sémantique peuvent être intégrées à l'analyse syntaxique.
D'ailleurs, beaucoup de phases de compilation postérieures à l'analyse syntaxique peuvent être intégrées à l'analyse syntaxique (traduction dirigée par la syntaxe)

Traduction dirigée par la syntaxe

Pour plus d'efficacité et de simplicité, elle mélange la phase d'analyse syntaxique avec des phases qui lui sont postérieures : analyse sémantique, production de code, etc.

Ajoute des actions portant sur les attributs des terminaux et des non-terminaux
- Les actions sont exécutées au fur et à mesure de l'analyse syntaxique

Analyse descendante :
- Dans le cas d'une programmation récursive,
  - les paramètres des fonctions d'analyse servent à transmettre des attributs
  - le retour de ces fonctions aussi
- Dans le cas d'une programmation par table/pile
  - les attributs sont placés sur la pile

Analyse ascendante :
- Les attributs sont des valeurs associées aux terminaux et aux non-terminaux
- Ils sont placés dans la pile d'exécution
- Mécanisme implanté par bison par les $$ et $i vus précédemment

Définition dirigée par la syntaxe

Règles sémantiques : code à exécuter par le compilateur pour réaliser son travail

Les règles sémantiques impliquent des dépendances entre les attributs

-> ordre d'évaluation des règles sémantiques

-> l'application des règles sémantiques permet effectivement le calcul des attributs

Les règles sémantiques du compilateur sont partiellement ordonnées

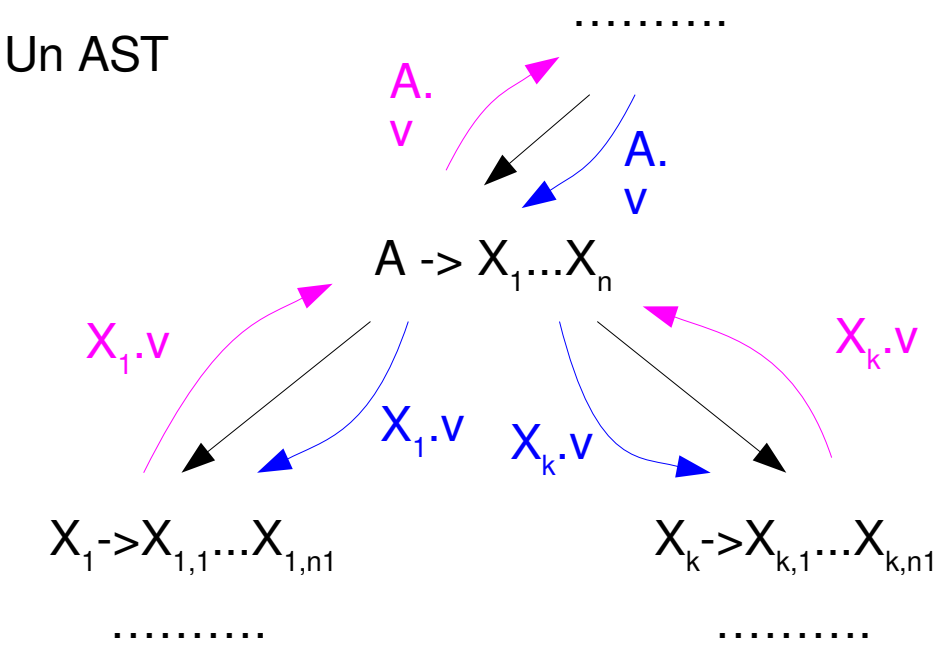
Grammaire attribuée : CFG dans laquelle chaque élément X de T∪N a des attributs

$X.a_1,\ldots,X.a_n$ qui peuvent être synthétisés ou hérités

Attributs synthétisés et hérités

Attribut synthétisé : dans une production A->γ associée à un nœud N d'un arbre de dérivation, un attribut synthétisé de A en membre gauche est défini par une règle sémantique de N qui ne dépend que des autres attributs dans N et dans ses fils

Attribut hérité : dans une production X->γYγ' associée à un nœud N d'un arbre de dérivation, un attribut hérité de Y en membre droit est défini par une règle sémantique du père de N qui ne dépend que des autres attributs dans N, dans son père et dans ses fils



Attribut synthétisé : exemple

Définition d'une calculette dirigée par la syntaxe

Un attribut synthétisé *val* par terminal et non-terminal de chaque production

(on numérote les terminaux et non-terminaux dans chaque production pour lever les ambiguïtés de nommage. Le membre gauche n'a pas de numéro.)

Production	Règle sémantique
E -> E + T	E.val := E_1.val + T.val
E -> T	E.val := T.val
T -> T * F	T.val := T_1.val * F.val
T -> F	T.val := F.val
F -> **(** E **)**	F.val := E.val
F -> nombre	F.val := nombre.val

nombre.val est la valeur du nombre

Les terminaux ne peuvent avoir que des attributs synthétisés (typiquement, par l'analyseur lexical)

Attribut hérité: exemple

Déclaration de variables en C

Un attribut hérité *val* par terminal et non-terminal de chaque production

(on numérote les terminaux et non-terminaux dans chaque production pour lever les ambiguïtés de nommage. Le membre gauche n'a pas de numéro.)

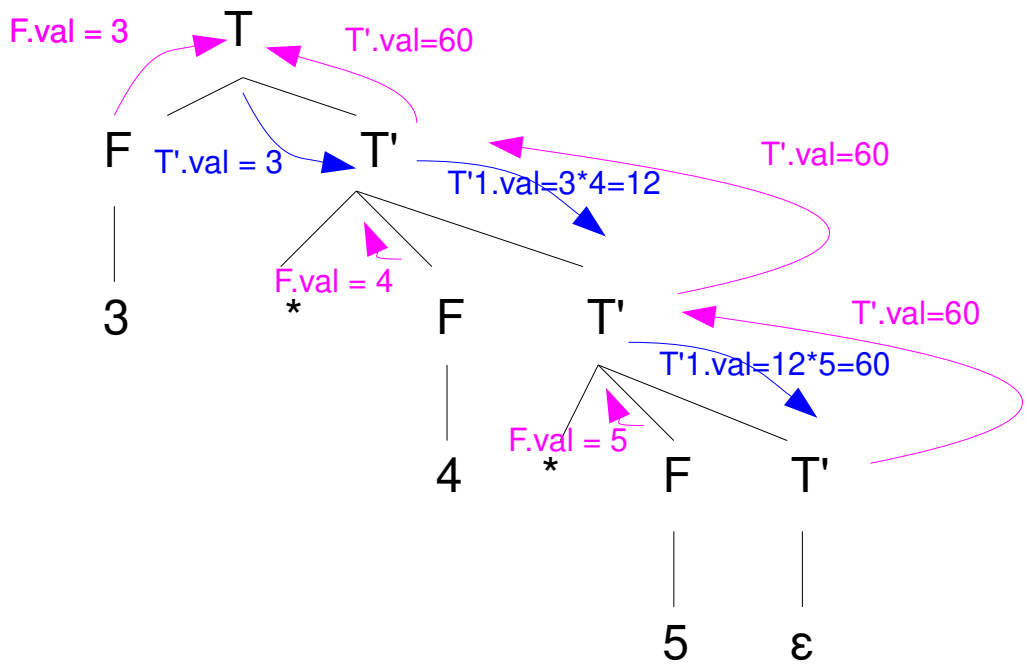
D -> T L	L.type := T.type
T -> **int**	T.type := int.type (=entier)
T -> **float**	T.type := float.type (=réel)
L -> L **, id**	L_1.type := L.type (qui est donc fixé par le nœud père)
	id.type := L.type (qui est donc fixé par le nœud père)
L -> **id**	id.type := L.type (qui est donc fixé par le nœud père)

L'attribut L.type est hérité

Attributs hérités et synthétisés

Les attributs hérités sont utiles quand la structure des AST diffère de celle des arbres de dérivation

Production	Règles sémantiques
T -> FT'	T'.val:= F.val
	T.val:=T'.val
T' -> *FT'	T'_1.val:=T'.val*F.val
	T'.val:=T_1'.val
T' -> ε	T'.val:=T'.val
F -> n	F.val:=entier(n)



Évaluation des attributs d'une grammaire attribuée

Si tous les attributs sont synthétisés : pas de problème !

Donné un arbre de dérivation T, n'importe quel parcours feuilles -> racine de T convient

Si des attributs sont hérités, des dépendances circulaires peuvent exister

Production	Règle sémantique
A -> B	A.s:= B.i
	B.i:= A.s+1

D'une manière générale, il est algorithmiquement difficile de dire, donnée une grammaire attribuée, s'il existe un arbre de dérivation avec des circularités d'évaluations.

Heureusement, pour certaines formes de grammaire, cette question se décide facilement.

Grammaire S-attribuée

Tous les attributs sont synthétisés.

Exemple : (on suppose qu'on dispose d'un type de listes et d'opérations associées)

L -> [S]	L.val:= S.val
L -> []	L.val:= emptyList()
S -> S , e	S.val:= S_1.val.addLast(e)
S -> e	S.val:= valeur(e)

Autre exemple :

S -> E$	S.val:= E.val
E -> T+E	E.val:=T.val+E.val
E -> T	E.val:=T.val
T -> n	T.val:=valeur(n)

Grammaire S-attribuée et analyse LR

L'analyse LR est naturellement adaptée à l'utilisation des attributs synthétisés.
La pile de l'analyseur LR peut servir au calcul des attributs.

Entrée	Pile	Action
1+2+3$	0	
+2+3$	013	déc. 0 par 1
+2+3$	0T(val=1)4	red. T->n
2+3$	0T(val=1)4+5	déc. 4 par +
+3$	0T(val=1)4+523	déc. 5 par 2
+3$	0T(val=1)4+5T(val=2)4	red. T->n
3$	0T(val=1)4+5T(val=2)4+5	déc. 4 par +
$	0T(val=1)4+5T(val=2)4+533	déc. 5 par 3
$	0T(val=1)4+5T(val=2)4+5T(val=3)4	red. T->n
$	0T(val=1)4+5T(val=2)4+5E(val=3)6	red. E->T
$	0T(val=1)4+5E(val=5)6	red. E->T+E
$	0E(val=6)1	red. E->T+E
	0E(val=6)1$2	déc. 1 par $

Acceptation, car il faut réduire S->E$!

Grammaire L-attribuée

- tout attribut est synthétisé ou hérité ;
- dans une règle A -> X_1 X_2 ...X_n, si un attribut X_i.val est calculé dans l'action associée,
  il ne dépend que des attributs des variables X_1 X_2 ...X_{i-1} (qui sont à gauche de X_i, d'où le L)
  ou des attributs hérités de A

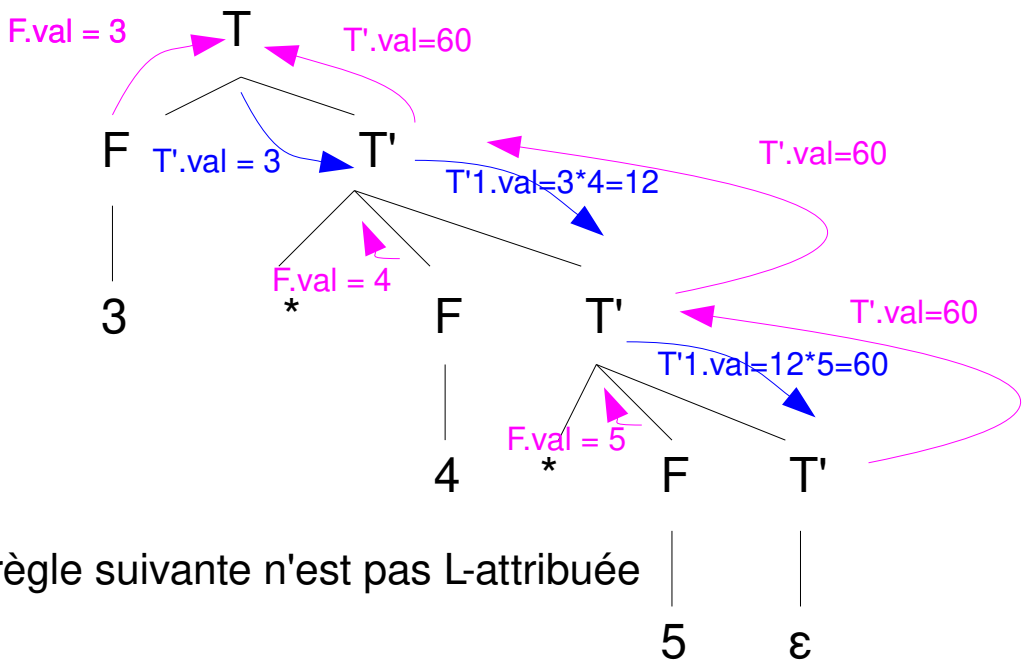
Le calcul des attributs peut être réalisé par un parcours en profondeur des arbres de dérivation

```
parcours(nœud n) {
    pour chaque fils m de n {
        calculer les attributs hérités de m ;
        parcours(m) ;
    }
    calculer les attributs synthétisés de n ;
}
```

Grammaire L-attribuée

Exemple : la grammaire suivante est L-attribuée

Production	Règles sémantiques
T -> FT'	T'.val:= F.val
	T.val:=T'.val
T' -> *FT'	T'_1.val:=T'.val*F.val
	T'.val:=T_1'.val
T' -> ε	T'.val:=T'.val
F -> n	F.val:=entier(n)



Exemple : n'importe quelle grammaire contenant la règle suivante n'est pas L-attribuée

A -> BC	A.val:=B.val
	B.val:=f(C.val, A.val)

Grammaire L-attribuée et analyse LL

Une implantation récursive d'analyse LL est naturellement adaptée pour les grammaires LL L-attribuées. Une implantation par table peut imiter l'implantation récursive.
Les paramètres et retour de fonction peuvent servir à la transmission des attributs hérités et synthétisés.

```
bool analyse_E(const char **s, int inherited, int *synthetised) {
  switch (**s) {
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    return analyse_T(s, 0, synthetised) && analyse_Ep(s, *synthetised, synthetised); /* E -> TE' */
  case '(':
    return analyse_T(s, 0, synthetised) && analyse_Ep(s, *synthetised, synthetised); /* E -> TE' */
  default: return false;
  }
}
bool analyse_Ep(const char **s, int inherited, int *synthetised) {
  switch (**s) {
  case '+': ++*s; return analyse_T(s, 0, synthetised) && analyse_Ep(s, inherited + *synthetised, synthetised); /* E' -> +TE' */
  case '-': ++*s; return analyse_T(s, 0, synthetised) && analyse_Ep(s, inherited - *synthetised, synthetised); /* E' -> -TE' */
  case ')': case '$': *synthetised = inherited; return true; /* E' -> Epsilon */
  default: return false;
  }
}
```

$S \rightarrow E\$$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid -TE' \mid \varepsilon$

$T \rightarrow FT'$
$T' \rightarrow *FT' \mid /FT' \mid \varepsilon$
$F \rightarrow n \mid (E)$

Schéma de traduction dirigée par la syntaxe

Grammaire contenant des actions dans son membre droit.

Les actions (on utilisera une syntaxe « à la C ») peuvent apparaître à n'importe quelle condition, mais

- si une action calcule un attribut d'un non-terminal du membre droit, elle doit être placée avant lui
- si une action utilise un attribut d'un non-terminal du membre droit, elle doit être placée après lui

Exemple : le schéma suivant est mal formé

S -> A A { A_1.val = 1 ; A_2.val = 2 ; }

A -> a { affiche A.val ; }

Celui-ci est correct

S -> { A_1.val = 1 ; } A { A_2.val = 2 ; } A
A -> a { affiche A.val ; }

Un schéma de traduction peut toujours s'implanter en construisant les arbres de dérivation et en les parcourant en profondeur.

Implantation des schémas de traduction

Un schéma de traduction peut toujours s'implanter en construisant les arbres de dérivation et en les parcourant en profondeur fils gauche d'abord.

On peut aussi utiliser facilement les implanter sans construire explicitement d'arbre de dérivation :

- pendant une analyse LL :
  - dans une implantation récursive, les actions s'insèrent facilement aux endroits appropriés dans le code des fonctions associées aux productions
- pendant une analyse LR :
  - on remplace Z-> γ{actions}γ' par Z->γXγ' avec X->ε {actions}.
  - les attributs hérités dans γ' et provenants de {actions} seront alors des attributs synthétisés de X : ils sont dans la pile
  - les attributs hérités en provenance du père peuvent poser difficulté (insurmontable)

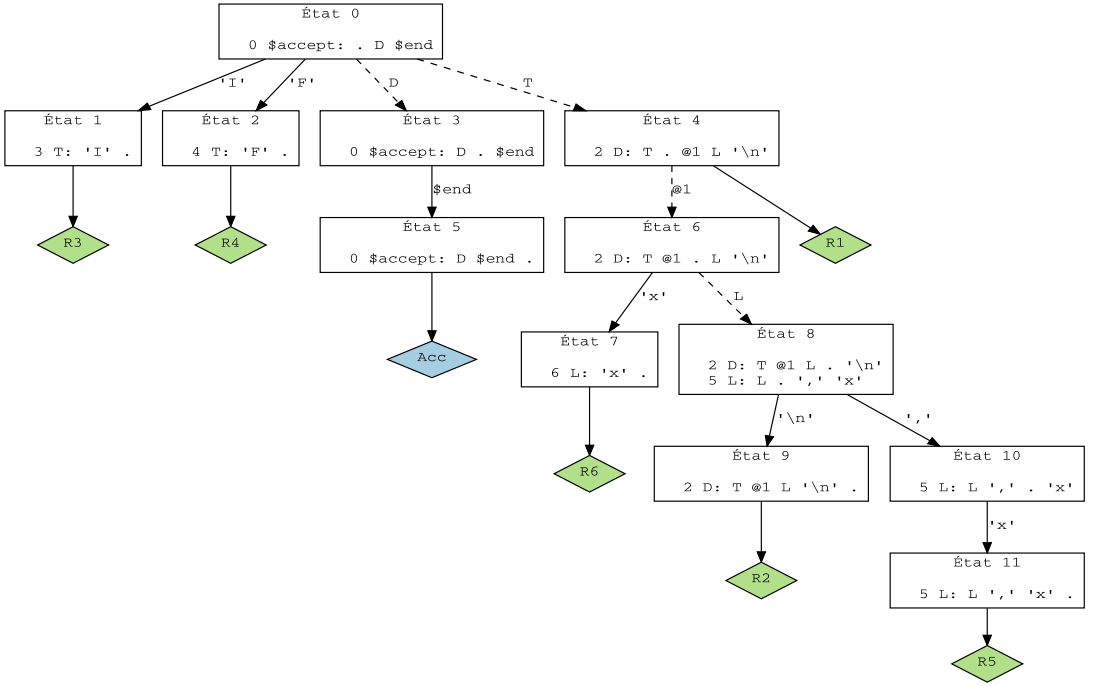
Implantation des schémas de traduction : analyse LR

D -> T { L.type = T.type ; } L '\n'
T -> int { T.type = integer ; }
T -> float { T.type = float ; }
L -> L, id { add_symbol_table(id.val, L.type); }
L -> id { add_symbol_table(id.val, L.type); }

D -> T A L '\n'
A -> ε { A.type = STACK(-1).type; }
T -> int { T.type = integer ; }
T -> float { T.type = float ; }
L -> L, id { add_symbol_table(id.val, STACK(-4).type); }
L -> id { add_symbol_table(id.val, STACK(-2).type); }

Entrée	Pile
int x,y\n$	0
x,y\n$	0int1
x,y\n$	0<T,int>4
x,y\n$	0<T,int>4<A,int>6
,y\n$	0<T,int>4<A,int>6x7
,y\n$	0<T,int>4<A,int>6L8
y\n$	0<T,int>4<A,int>6L8,10
\n$	0<T,int>4<A,int>6L8,10y11
\n$	0<T,int>4<A,int>6L8
$	0<T,int>4<A,int>6L8\n9
$	0D3
	0D3$5

Acceptation



Les schémas de traduction en bison

Du code peut être inséré n'importe où dans le membre droit

```
%type<type> type
%type<string> ID
%union {
        char *string ;
        int type ;
}
%%
decl : type    { $<type>$=$1 ; } id_list '\n'
;
type : INT              { $$=INT_TYPE ; }
| FLOAT         { $$=FLOAT_TYPE ; }
;
id_list : id_list ',' ID        { add_symbol_table($3, $<type>0); }
| ID                            { add_symbol_table($3, $<type>0); }
;
```

$0, $-1, $-2 : on remonte dans la pile. Attention : bison ne peut pas typer ce qu'il va y trouver ! (au programmeur de le faire correctement !)

Les schémas de traduction en bison

```
decl : type    { $<type>$=$1 ; } id_list '\n'
;
type : INT              { $$=INT_TYPE ; }
| FLOAT         { $$=FLOAT_TYPE ; }
;
id_list : id_list ',' ID        { add_symbol_table($3, $<type>0); }
| ID                            { add_symbol_table($3, $<type>0); }
;
```

est transformé par bison en

```
decl : type  @1 id_list '\n'
;
@1 :%empty { $<type>$ = $1 ; }
type : INT              { $$=INT_TYPE ; }
| FLOAT         { $$=FLOAT_TYPE ; }
;
id_list : id_list ',' ID        { add_symbol_table($3, $<type>0); }
| ID                            { add_symbol_table($3, $<type>0); }
;
```

Attention : dans le code de @1

- $$ est relatif à @1
- Comme la règle @1 n'est pas typée, il faut explicitement y typer $$
- $1 a la même signification qu'avant la transformation !

Les schémas de traduction en bison

La génération de règles automatiques @n peut amener des conflits

bloc : { code de préparation pour les déclarations } '{' decls instrs '}'
| '{' instrs '}' ;

Conflit : faut-il exécuter le code ou pas quand le prochain caractère est '{' ?

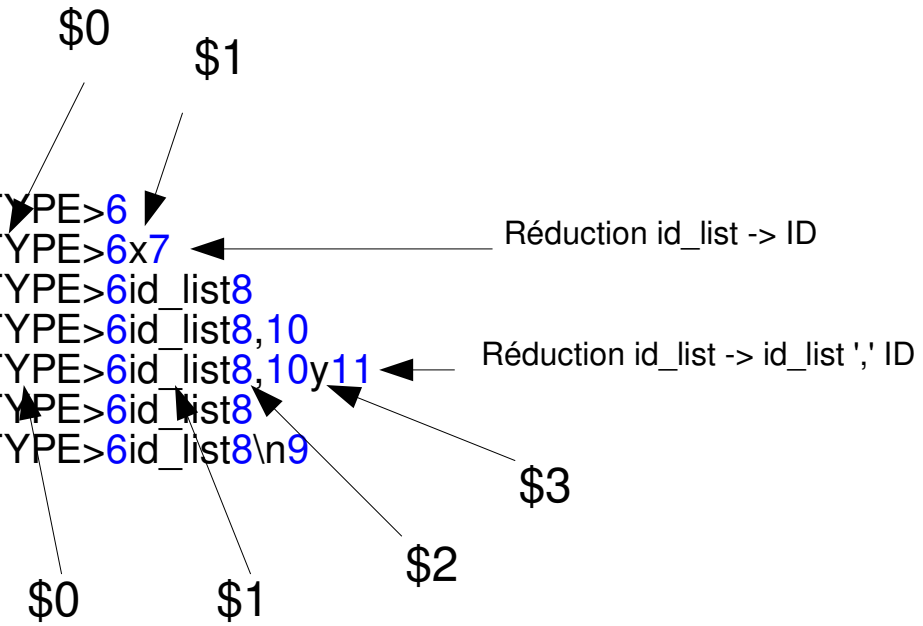
Solution : toujours l'exécuter
bloc : { code de préparation pour les déclarations } '{' decls instrs '}'
| { code de préparation pour les déclarations } '{' instrs '}' ;
mais ça ne fonctionne pas car bison génère deux règles @ différentes

Le programmeur doit le faire lui-même :
bloc : code '{' decls instrs '}'
| code '{' instrs '}' ;
code : % empty { code de préparation pour les déclarations } ;

Les schémas de traduction en bison

```
decl : type    { $<type>$=$1 ; } id_list '\n'
type : INT             { $$=INT_TYPE ; }
| FLOAT        { $$=FLOAT_TYPE ; }
;
id_list : id_list ',' ID      { add_symbol_table($3, $<type>0); }
| ID                          { add_symbol_table($3, $<type>0); }
```

Entrée	Pile
int x,y\n$	0
x,y\n$	0int1
x,y\n$	0<type,INT_TYPE>4
x,y\n$	0<type,INT_TYPE>4<@1,INT_TYPE>6
,y\n$	0<type,INT_TYPE>4<@1,INT_TYPE>6x7
,y\n$	0<type,INT_TYPE>4<@1,INT_TYPE>6id_list8
y\n$	0<type,INT_TYPE>4<@1,INT_TYPE>6id_list8,10
\n$	0<type,INT_TYPE>4<@1,INT_TYPE>6id_list8,10y11
\n$	0<type,INT_TYPE>4<@1,INT_TYPE>6id_list8
$	0<type,INT_TYPE>4<@1,INT_TYPE>6id_list8\n9
$	0decl3
	0decl3$5
Acceptation	



Priorité des opérateurs en C

Opérateur	Symbole	Exemple	Assoc.
Indexation	[]	t[i][j]	
Accès struct.	. ->	c.reel	->
Appel fct.	()	fact(30U)	
Adresse	&	&v	
Déréférecmnt.	*	*p	
Taille objets	sizeof	sizeof(int)	
Transtypage	()	(float)4	
Nég. Logique	!	!b	<-
Cplt à un	~	~0022	
pl/neg unaires	+ -	+1 -1	
pré/post inc.	++	++i i++	
pré/post dec.	--	--i i--	

Priorité des opérateurs en C

Opérateur	Symbole	Exemple	Assoc.
Produit div/modulo	* / %	a*b a/b a%b	->
somme/diff.	+ -	a+b a-b	->
Décalages	<< >>	a<<1 a>>1	->
Ordre	< <= => >	a<b ...	->
égalité/diff.	== !=	a==0	->
Et bit à bit	&	a&0022	->
XOR bit à bit	^	a^0022	->
OU bit à bit	|	a|0022	->
ET logique	&&	a&&b	->
OU logique	||	a||b	->
expr. cond.	?:	x>0 ? x: -x	<-
affectation affect. étendue	= += -= ...	a=3 a <<= 3U	<-
expr. comp.	,	i=2, j=6	->

Environnement d'exécution

Mécanisme final d'exécution du programme
- stockage des données
  - représentation des données
  - gestion des données
    - accès
    - registre, tas, pile, zone globale, zone des constantes
- fonctionnalités
  - instruction
    - sémantique d'exécution (effets sur l'environnement)
  - fonctions
    - passage des arguments
    - appel
    - valeur de retour
  - fonctions de la bibliothèque
    - sémantique

Rôle du compilateur
- implantation des abstractions du langage source dans l'environnement d'exécution
  - traduction du stockage des données
  - traduction des fonctionnalités du langage source

Typage

L'analyse de types fait partie de l'analyse sémantique

Le système de typage sert à :
- vérifier que les opérations faites sur les objets sont bien autorisées
- calculer la place nécessaire à la représentation des données en mémoire

Il peut être spécifié par des schémas de traduction

Il peut être
- statique : les opérations de typage sont réalisées par le compilateur.
  - Elles garantissent une exécution correcte (en ce qui concerne les types)
  - Le langage est souvent déclaratif
  - Difficultés de réutilisation de code
  - Permet des changements de type automatiques réalisées par le compilateur (gain de temps)
  - Exemples : C, C++, Java
- dynamique : les opérations de typages sont effectuées pendant l'exécution du programme
  - Implique des surcoûts dans les temps d'exécution et l'espace mémoire utilisés
  - Permet au programmeur de n'avoir qu'une connaissance partielle des types (souplesse)
  - Exemples : ocaml, Java
- mixte : combine les deux approches
  - Exemple : Java

Types

Un type définit
- un ensemble de valeurs
- pour certains types, une représentation (donc une taille) pour les valeurs

Des opérations permettent de manipuler les valeurs

Ils sont
- fournis par le langage
- définis par l'utilisateur

et peuvent être combinés en utilisant des constructeurs de types

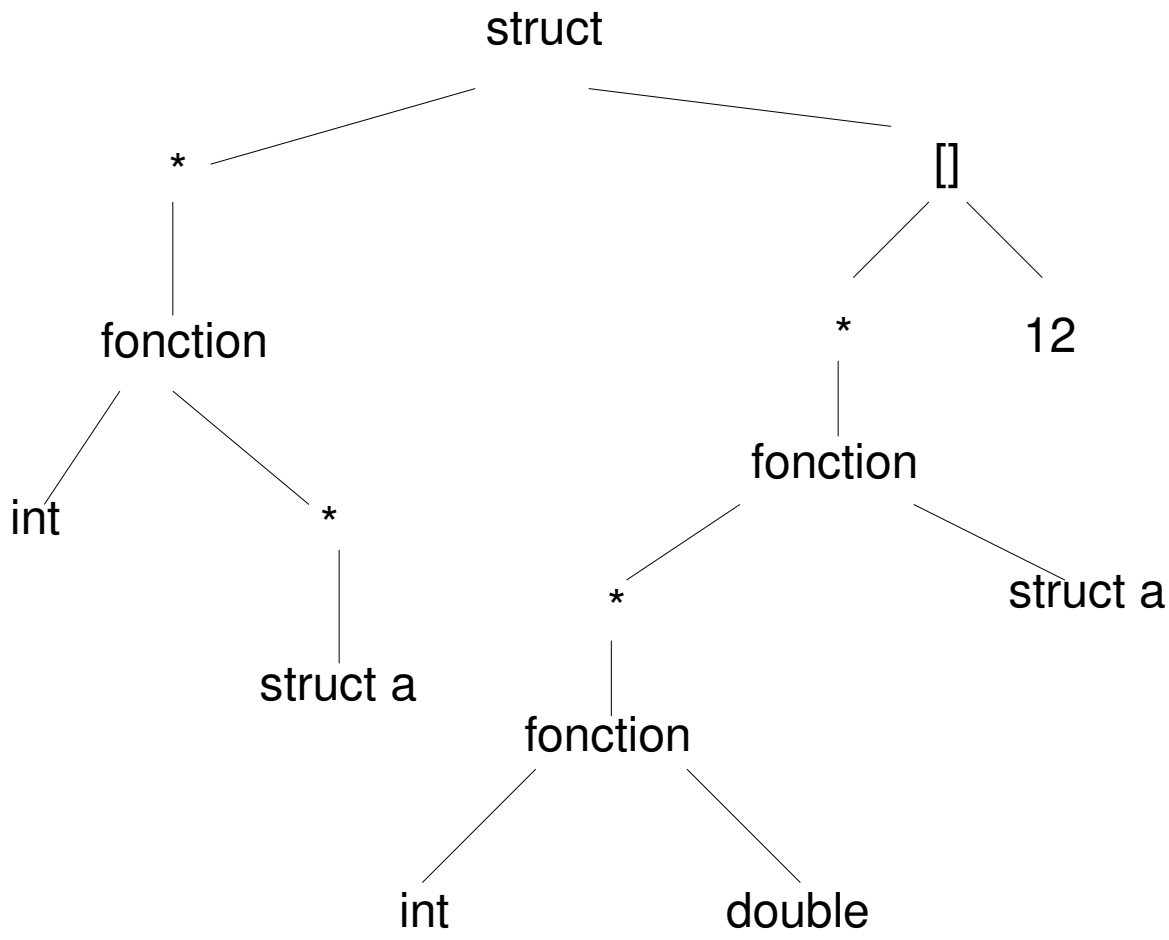
Types de base en C :
- void (type spécial), int, float, double, char, …

Constructeurs de types en C :
- t * : pointeur sur t
- t[] : tableau de t
- union : union de types
- struct : produit de types
- () : fonction

Expression de types

```
struct a {
	int (*f)(struct a *) ;
	int (*(*t[12])(struct a))(double) ;
}
```

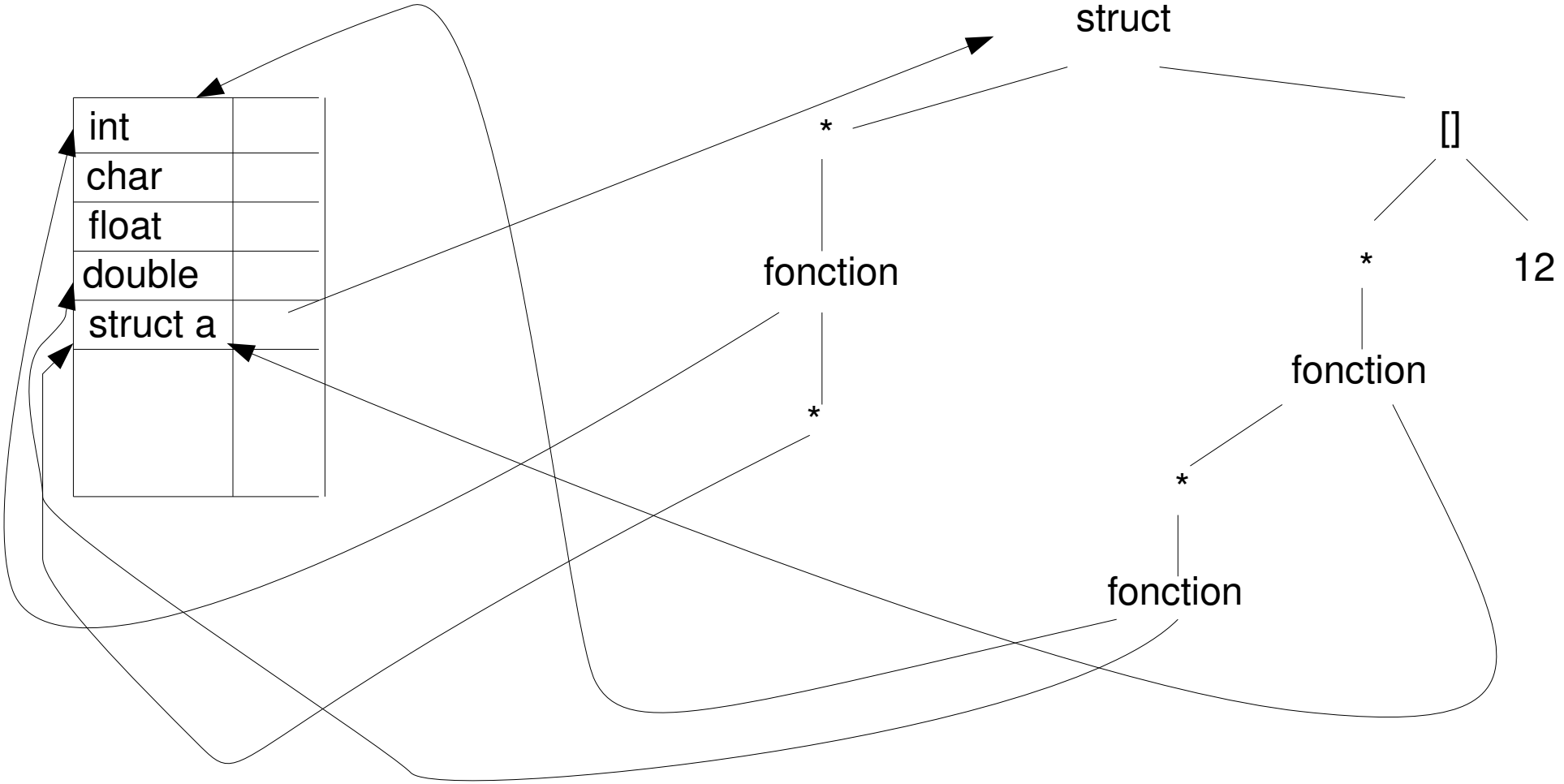
Représentation par arbre



Expression de types

```
struct a {
        int (*f)(struct a *) ;
        int (*(*t[12])(struct a))(double) ;
}
```

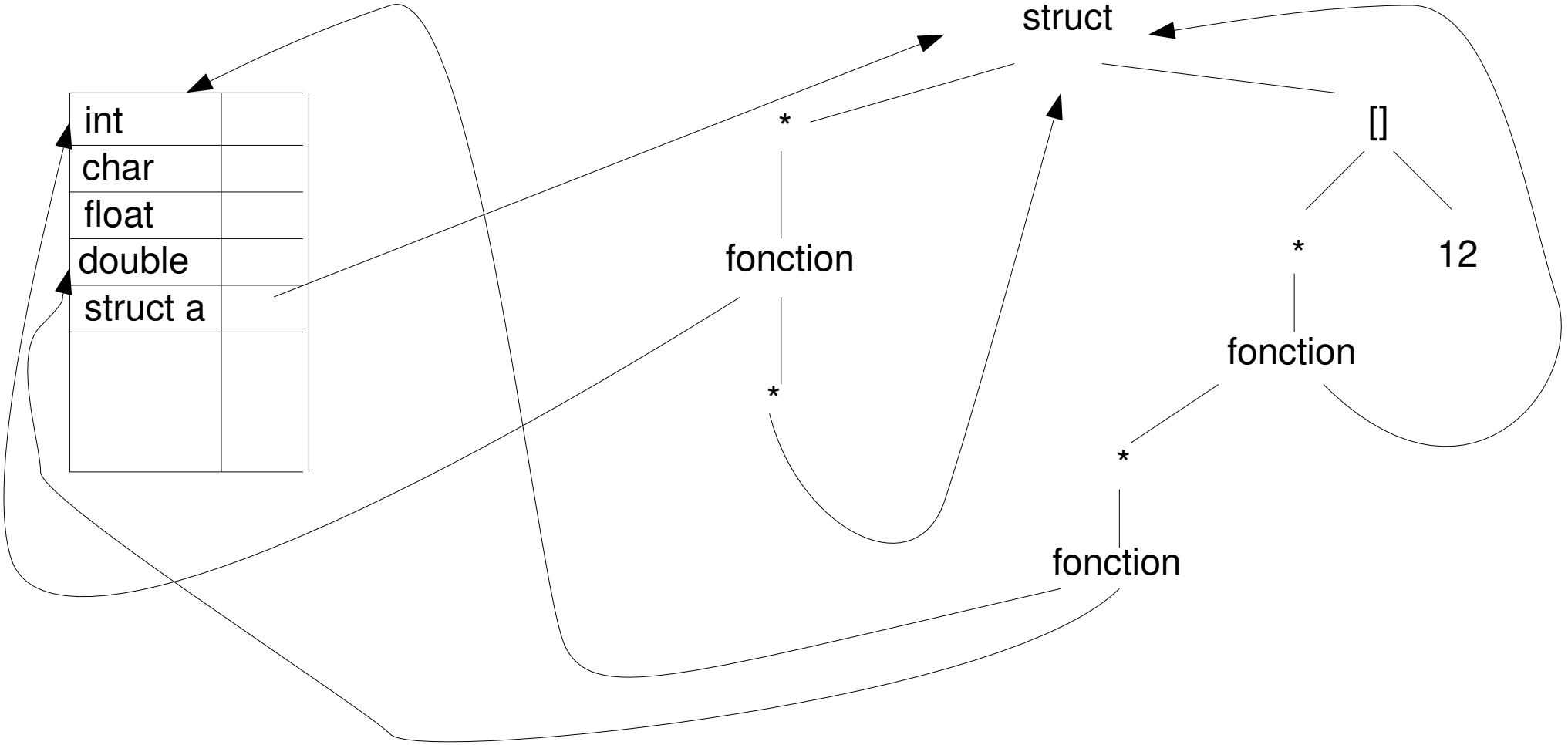
Représentation par DAG :
on s'arrête en revenant au tableau



Expression de types

```
struct a {
        int (*f)(struct a *) ;
        int (*(*t[12])(struct a))(double) ;
}
```

Représentation par graphe :
on continue après le tableau (en le court-circuitant)



Équivalences de types

Équivalence structurelle : équivalence des graphes représentant les types (en posant que la table est court-circuitée)

struct a { int a ; } ~ struct b { int a ; }

Équivalence par nommage : équivalence des DAG

struct a { int a ; } ≁ struct b { int a ; }

Exemple du C

```
struct a { int a ; } v1 ;
struct b { int a ; } v2 ;
v1 = v2 ; // Erreur : types incompatibles
```

Attention aux synonymes !

Exemple du C

```
struct a { int a ; } ;
typedef struct a ap ; // ap et struct a sont la même entrée dans la table des types
struct a v1 ;
ap v2 ;
v1 = v2 ; // Correct
```

Calcul d'un arbre/DAG/graphe pour le type

Exemple du C : int *t[5][3]

t est un tableau de 5 tableaux de 3 pointeurs sur int

[] est prioritaire sur *

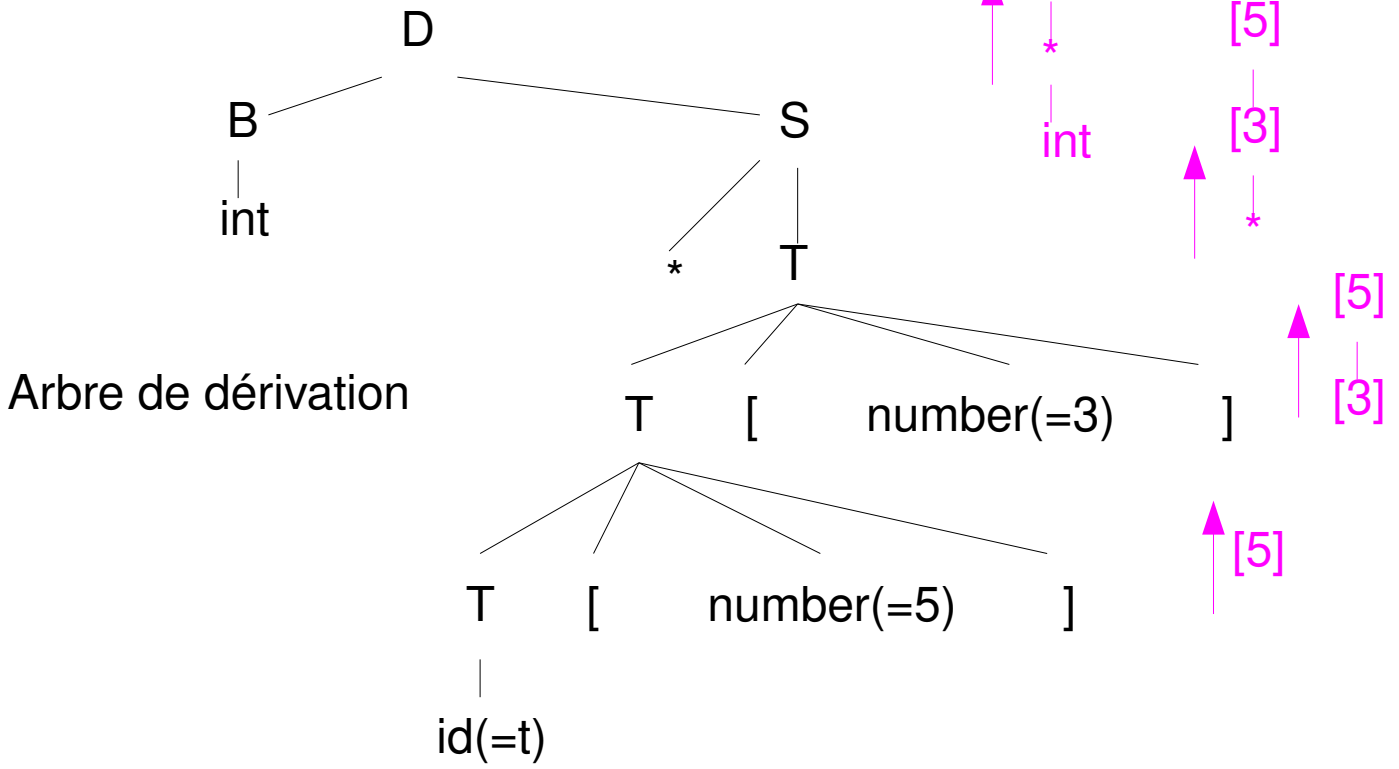
Une grammaire prenant en compte les priorités :

D -> B S

B -> int | float

S -> * S | T

T -> (S) | T [number] | id



Arbre de dérivation

Arbre pour le type

Contrôle de type
Un exemple simple

```
prog: decls instrs ;

decls: decls decl | decl ;

decl: type ID ';'  { add_symbol_table( $2, $1) ; }

type:
INT                                  { $$ = get_type_node(INT_TYPE) ; }
| BOOLEAN                            { $$ = get_type_node(BOOLEAN_TYPE) ; }
| ARRAY OF type '[' NUMBER ']'  { $$ = new_type_node(ARRAY_TYPE) ; $$->size = $5 ; $$->fg = $3 ; }
| '*' type                           { $$ = new_type_node(POINTER_TYPE) ; $$->fg = $2 ; }
| FUNCTION type TO type          { $$ = new_type_node(FUNCTION_TYPE) ; $$->fg = $2 ; $$->fd = $4 ; }
;

instrs: instrs instr | instr ;

instr:
IF expr THEN instr ELSE instr ';' { if (!equiv_type( $2, get_type_node(BOOLEAN_TYPE) )) erreur ; }
| IF expr THEN instr ';'              { if (!equiv_type( $2, get_type_node(BOOLEAN_TYPE) )) erreur ; }
| WHILE expr THEN instr ';'       { if (!equiv_type( $2, get_type_node(BOOLEAN_TYPE) )) erreur ; }
| ID AFF expr ';'                     { symbol_table_entry *ste ; if ( (ste = get_symbol_table( $1 )) == NULL ) erreur ;
                                        if (!equiv_type( ste->type, $3 )) erreur ; }
;

expr:
TRUE              { $$ = get_type_node(BOOLEAN_TYPE) ; }
| FALSE               { $$ = get_type_node(BOOLEAN_TYPE) ; }
| NUMBER        { $$ = get_type_node(INT_TYPE); }
| ID                { symbol_table_entry *ste ; if ( (ste = get_symbol_table( $1 )) == NULL ) erreur else $$ = ste->type ; }
| expr EQ expr    { if ( !equiv_type( $1, $3) ) erreur else $$ = get_type_node(BOOLEAN_TYPE) ; }
| expr '(' expr ')'   { if ( $1->type != FUNCTION_TYPE || !equiv_type($3, $1->fg) ) erreur else $$ = $1->fd ; }
| expr '[' expr ']'   { if ( $1->type != ARRAY_TYPE || ! equiv_type( $3, get_type_node(INT_TYPE) ) erreur else $$ = $1->fg ; }
| '*' expr            { if ( $2->type!= POINTER_TYPE) erreur else $$ = $2->fg ; }
;
```

Conversion de type

v :t -> v' :t'

Trouver, dans le type t', une valeur v' « correspondant » à v de type t

Implique souvent (mais pas toujours) un changement de représentation.
Peut impliquer aussi un changement de la valeur.

Promotion : conversion implicite.
But : normaliser la manipulation de certaines catégories de données par la machine
Ne devrait pas engendrer de perte de données.

Conversion implicite : réalisée automatiquement par le compilateur.
Ne devrait pas engendrer de perte de données
Ça n'est pas le cas en C et en Java, par exemple

Conversion explicite : le programmeur doit indiquer sa volonté de convertir
(et donc, qu'il sait qu'il peut y avoir perte de données)

Ne pas confondre conversion et sous-typage. En Java par exemple

```
// A est un super-type de B
A a = new B(); // Aucune conversion
B b = (B) a;     // Aucune conversion : le programmeur indique juste au compilateur qu'il sait
                    // que la référence a désigne un objet de type B.
```

Promotion :
Exemple de la promotion entière en C et en Java

Plusieurs types d'entiers : char, short, int (et pour chacun, un type *unsigned* en C)

En termes ensemblistes, char ⊆ int, short ⊆ int

Le type int étant le type « naturel » des entiers pour la machine, la somme int*int -> int est la plus efficace

Plutôt que d'avoir une somme pour chaque combinaison de types possibles :
char*char -> char, short*short -> short, int*int -> int
on en utilise une seule :
int*int -> int
Il faut donc convertir les arguments en int avant de réaliser la somme.
La conversion est automatique pour ne pas alourdir la syntaxe des programmes

L'exécution de la somme est efficace, mais il faut réaliser la conversion de ses opérandes avant

Promotion :
Exemple de la promotion entière en C et en Java

En C :

(signed+unsigned)(char+short int) —Promotion entière→ int+unsigned int

Si int contient le type d'origine, alors conversion en int, sinon en unsigned int

La promotion entière est donc machine dépendante

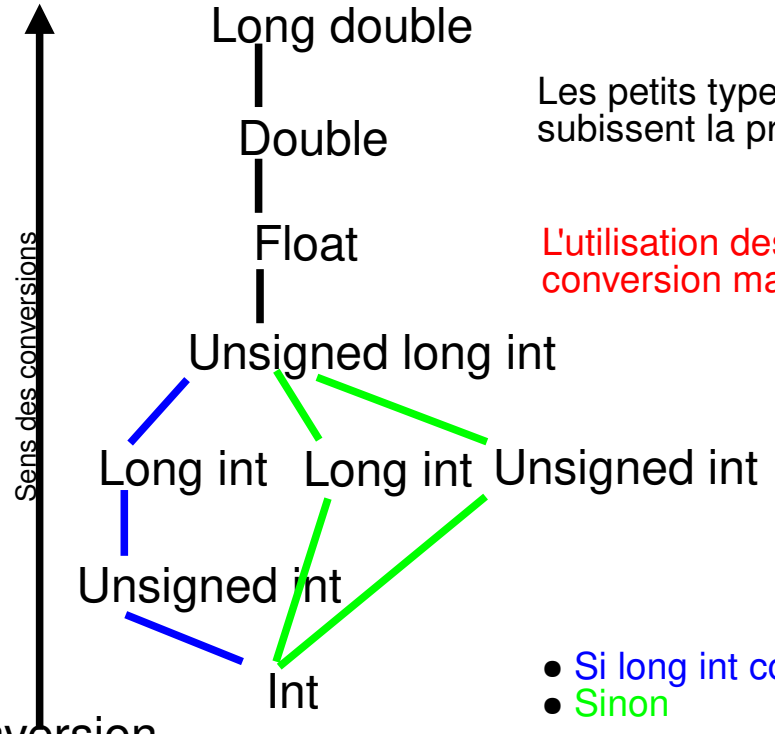
En Java :

```
short s = 5;
s = s + s;                  // Erreur à la compilation:
                            // Explicit cast needed to convert int to short.

System.out.println(s + s);  // Affiche 10. C'est System.out.println(int) qui est appelée
s += s;                     // Correct (affectation étendue)
```

Conversion implicite : exemple en C

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

unsigned f(unsigned int u) {
  return u;
}

int main(int argc, char *argv[]) {
  unsigned int u = argc; // 1
  printf("%u", argc);       // 2
  f(argc);                  //  3
  return EXIT_SUCCESS;
}
```

gcc -std=c11 -pedantic -Wall -> OK
gcc -std=c11 -pedantic -Wall -Wconversion
-> Warnings pour 1 et 3



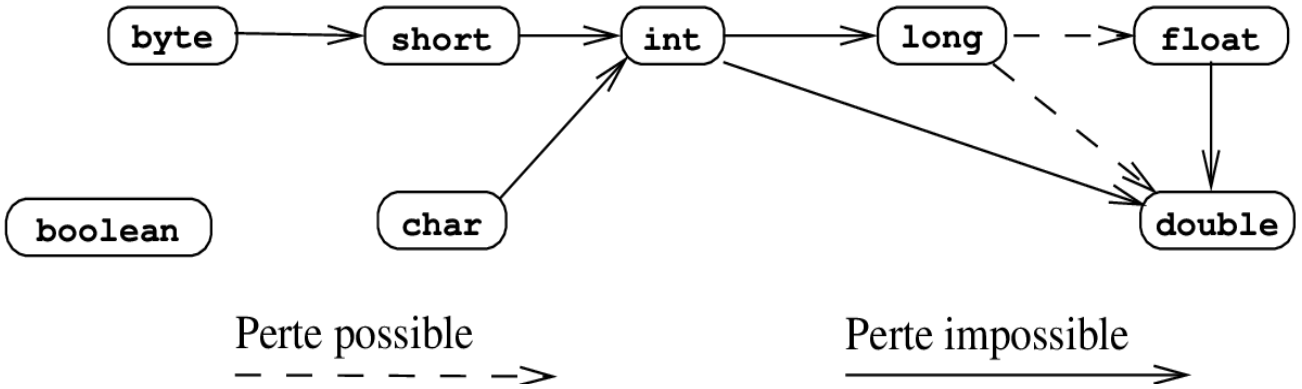
Les petits types d'entiers (char, short) subissent la promotion entière

L'utilisation des unsigned rend la conversion machine-dépendante

- Si long int contient unsigned int
- Sinon

Dangereux :
perte de données possible
trop difficile à bien maîtriser

Conversion implicite en Java



Les petits types d'entiers (char, byte, short) subissent la promotion entière

```
class Conversion {
   public static void main(String[] args) {
      float f1,f2;
      int i1 = 475435677;
      int i2 = i1 + 1;
      f1 = i1;
      f2 = i2;
      System.out.println(i1==i2); // false
      System.out.println(f1==f2); // true !
   }
}
```

Dangereux !

Surcharge

Consiste à avoir plusieurs opérateurs (resp. fonctions, méthodes) dénotés par le même symbole, mais avec des types différents.

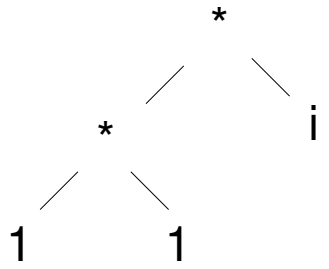
Le choix de l'opérateur (resp. de la fonction, méthode) dépend du contexte d'utilisation.

Exemples :

```
System.out.println(1) ; // C'est System.out.println(int) qui est appelée
System.out.println('c') ; // C'est System.out.println(char) qui est appelée

1+2 ;       // Somme de deux entiers : calcule un entier
1.0+2.0 ; // Somme de deux réels (opération différente de la première) : calcule un réel
```

Opérations de produit :

Deux types (entiers e, complexes c), trois produits : e*e->e, c*c->c, e*e->c



Une manière pour résoudre la surcharge :

1) Une remontée récursive pour calculer les types possibles
2) Si pas unicité du type possible à la racine : erreur
3) Descente récursive pour fixer les types
4) Si pas unicité erreur

Surcharge

Consiste à avoir plusieurs opérateurs (resp. fonctions, méthodes) dénotés par le même symbole, mais avec des types différents.

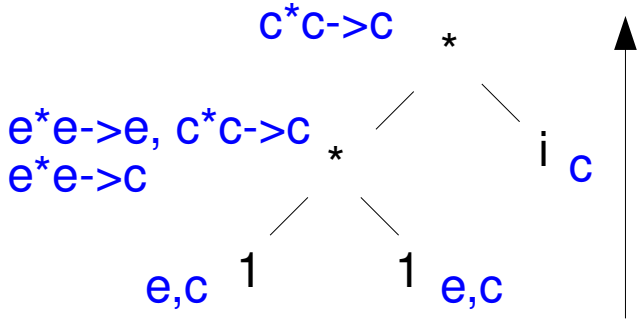
Le choix de l'opérateur (resp. de la fonction, méthode) dépend du contexte d'utilisation.

Exemples :

```
System.out.println(1) ; // C'est System.out.println(int) qui est appelée
System.out.println('c') ; // C'est System.out.println(char) qui est appelée

1+2 ;       // Somme de deux entiers : calcule un entier
1.0+2.0 ; // Somme de deux réels (opération différente de la première) : calcule un réel
```

Opérations de produit :

Deux types (entiers e, complexes c), trois produits : e*e->e, c*c->c, e*e->c



Une manière pour résoudre la surcharge :

1) Une remontée récursive pour calculer les types possibles
2) Si pas unicité du type possible à la racine : erreur
3) Descente récursive pour fixer les types
4) Si pas unicité erreur

Surcharge

Consiste à avoir plusieurs opérateurs (resp. fonctions, méthodes) dénotés par le même symbole, mais avec des types différents.

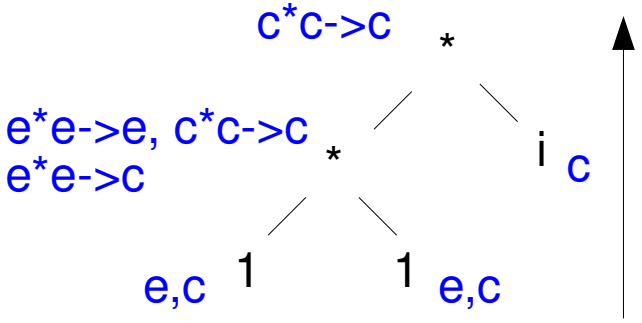
Le choix de l'opérateur (resp. de la fonction, méthode) dépend du contexte d'utilisation.

Exemples :

```
System.out.println(1) ; // C'est System.out.println(int) qui est appelée
System.out.println('c') ; // C'est System.out.println(char) qui est appelée

1+2 ;       // Somme de deux entiers : calcule un entier
1.0+2.0 ; // Somme de deux réels (opération différente de la première) : calcule un réel
```

Opérations de produit :

Deux types (entiers e, complexes c), trois produits : e*e->e, c*c->c, e*e->c



Une manière pour résoudre la surcharge :

1) Une remontée récursive pour calculer les types possibles
2) Si pas unicité du type possible à la racine : erreur
3) Descente récursive pour fixer les types
4) Si pas unicité erreur

Surcharge

Consiste à avoir plusieurs opérateurs (resp. fonctions, méthodes) dénotés par le même symbole, mais avec des types différents.

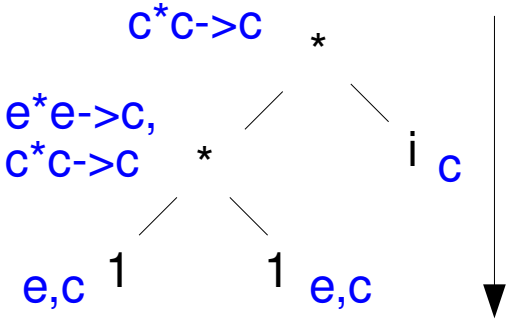
Le choix de l'opérateur (resp. de la fonction, méthode) dépend du contexte d'utilisation.

Exemples :

```
System.out.println(1) ; // C'est System.out.println(int) qui est appelée
System.out.println('c') ; // C'est System.out.println(char) qui est appelée

1+2 ;       // Somme de deux entiers : calcule un entier
1.0+2.0 ; // Somme de deux réels (opération différente de la première) : calcule un réel
```

Opérations de produit :

Deux types (entiers e, complexes c), trois produits : e*e->e, c*c->c, e*e->c



Une manière pour résoudre la surcharge :

1) Une remontée récursive pour calculer les types possibles
2) Si pas unicité du type possible à la racine : erreur
3) Descente récursive pour fixer les types
4) Si pas unicité erreur

Surcharge

Consiste à avoir plusieurs opérateurs (resp. fonctions, méthodes) dénotés par le même symbole, mais avec des types différents.

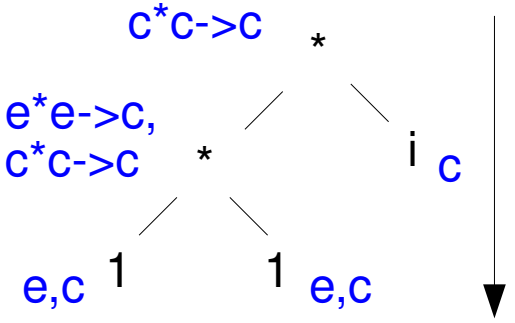
Le choix de l'opérateur (resp. de la fonction, méthode) dépend du contexte d'utilisation.

Exemples :

```
System.out.println(1) ; // C'est System.out.println(int) qui est appelée
System.out.println('c') ; // C'est System.out.println(char) qui est appelée

1+2 ;       // Somme de deux entiers : calcule un entier
1.0+2.0 ; // Somme de deux réels (opération différente de la première) : calcule un réel
```

Opérations de produit :

Deux types (entiers e, complexes c), trois produits : e*e->e, c*c->c, e*e->c



Une manière pour résoudre la surcharge :

1) Une remontée récursive pour calculer les types possibles
2) Si pas unicité du type possible à la racine : erreur
3) Descente récursive pour fixer les types
4) Si pas unicité erreur

Surcharge
Exemple de Java

On peut aussi décider, quand plusieurs types sont possibles, d'établir des priorités

```
class Surcharge {
    static interface I {}
    static class A implements I {}
    public static void m(I a, I b) {
        System.out.println("II");
    }
    public static void m(I a, A b) {
        System.out.println("IA");
    }
    public static void m(A a, I b) {
        System.out.println("AI");
    }
    public static void main(String[] args) {
        A a = new A();
        A b = new A();
        m(a,b);
    }
}
```

Quand la méthode avec les bons arguments n'est pas trouvée, on remonte l'arbre des types sur tous les arguments, en affectant un poids aux remontées

Il doit y avoir une remontée unique avec un poids minimal et amenant à une méthode.

```
 javac Surcharge.java
Surcharge.java:16: error: reference to m is
                                        ambiguous
        m(a,b);
        ^
  both method m(I,A) in Surcharge
        and method m(A,I) in Surcharge match
1 error
```

Surcharge
Exemple de Java

On peut aussi décider, quand plusieurs types sont possibles, d'établir des priorités

```
class SurchargeConversion {
    public static void m(long a, long b) {
        System.out.println("LL");
    }
    public static void m(long a, double b) {
        System.out.println("LD");
    }
    public static void m(double a, long b) {
        System.out.println("DL");
    }
    public static void m(double a, double b) {
        System.out.println("DD");
    }
    public static void main(String[] args) {
        m(1,2); // LL
    }
}
```

Polymorphisme

Les types des fonctions ne sont pas entièrement fixés à l'avance.

Faciliter la ré-utilisabilité

Présent dans les langages fonctionnels, mais aussi en Java 8, par exemple.

Exemple en ocaml :

```
let rec length = function
        [] -> 0
        | _::l -> 1+length l
;;
length : 'a list -> int
```

La liste en paramètre peut être contenir des éléments de n'importe quel type
(tous les éléments sont du même type).
Le type des éléments, inconnu, est noté 'a.

Inférence de type : trouver le type d'une expression

Unification

Trouver, à partir de deux expressions de type e et e', une substitution S qui à chaque variable de type associe un type de telle manière que S(t)=S(t')

Exemple : soient les deux expressions de type suivantes

e : (('a -> 'b)*'c list) -> 'b list

e' : (('c -> 'd)*'c list) -> 'e

Les deux substitutions S et S' suivantes définissent chacune une unification de e et e'

x	S(x)	S'(x)
'a	'c	'a
'b	'b	'a
'c	'c	'a
'd	'b	'a
'e	'b list	'a list

En effet

S(e) = (('c -> 'b)*'c list) -> 'b list et S'(e) = (('a -> 'a)*'a list) -> 'a list

S(e') = (('c -> 'b)*'c list) -> 'b list S'(e') = (('a -> 'a)*'a list) -> 'a list

S est plus générale que S', puisque

en substituant dans S 'c et 'b par 'a on obtient S'

il n'existe pas de substitution permettent d'obtenir S à partir de S'

L'**unificateur** de e et e' est la substitution *la plus générale* (aux noms de variables près) permettant d'unifier e et e'

Trouver un type général pour une fonction : l'inférence de type

let rec f = function [] -> (function x -> x) | x::m -> (function l -> f m x::l) ;;

- On pose f : 'a -> 'b
- Le type de 'a est une 'c list
- Unification de 'a et 'c list : 'a='c list
- Donc f : 'c list -> 'b
- Changement de variable : f : 'a list -> 'b
- Unification de 'b et 'd -> 'd
- Donc f : 'a list -> ('d -> 'd)
- Changement de variable : f : 'a list -> 'b -> 'b
- Unification de 'b->'b et 'c list->('a list->'b->'b) 'c list 'c list
  - 'b = 'c list
  - ('a list->'b->'b) 'c list 'c list : 'c list en prenant 'a='c et 'b='c list

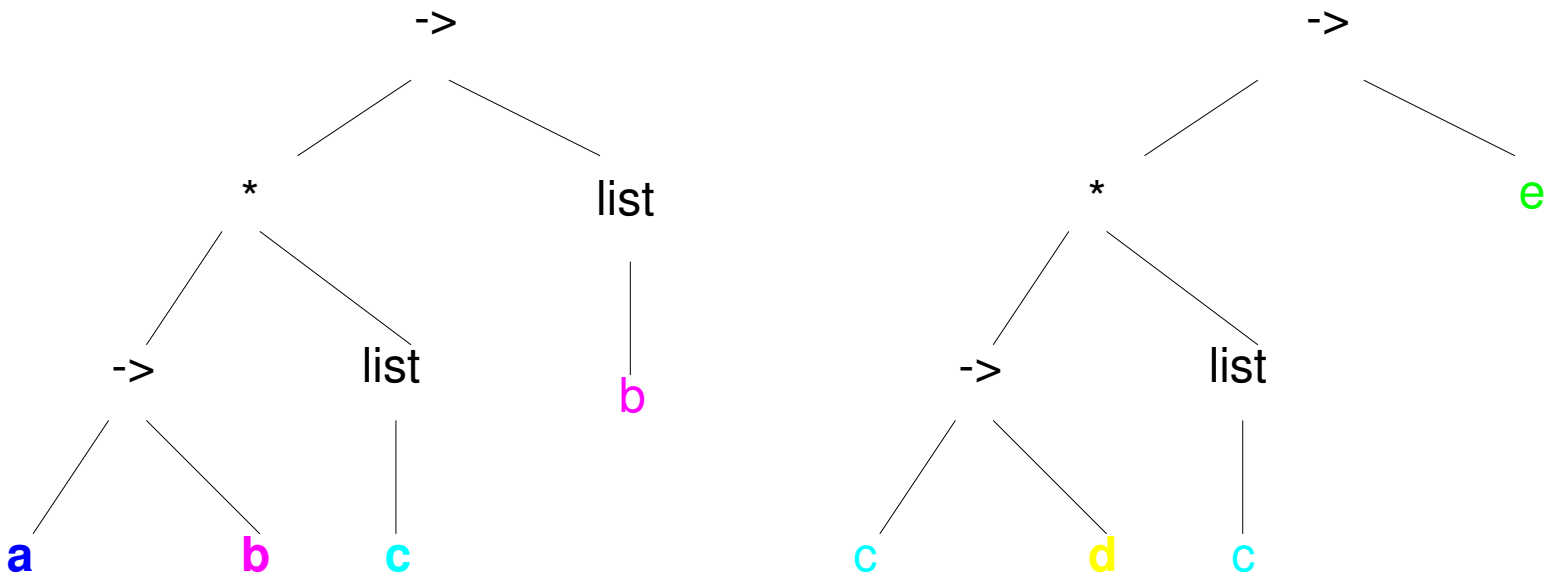
Donc f :'a list->'a list->'a list

Un algorithme d'unification

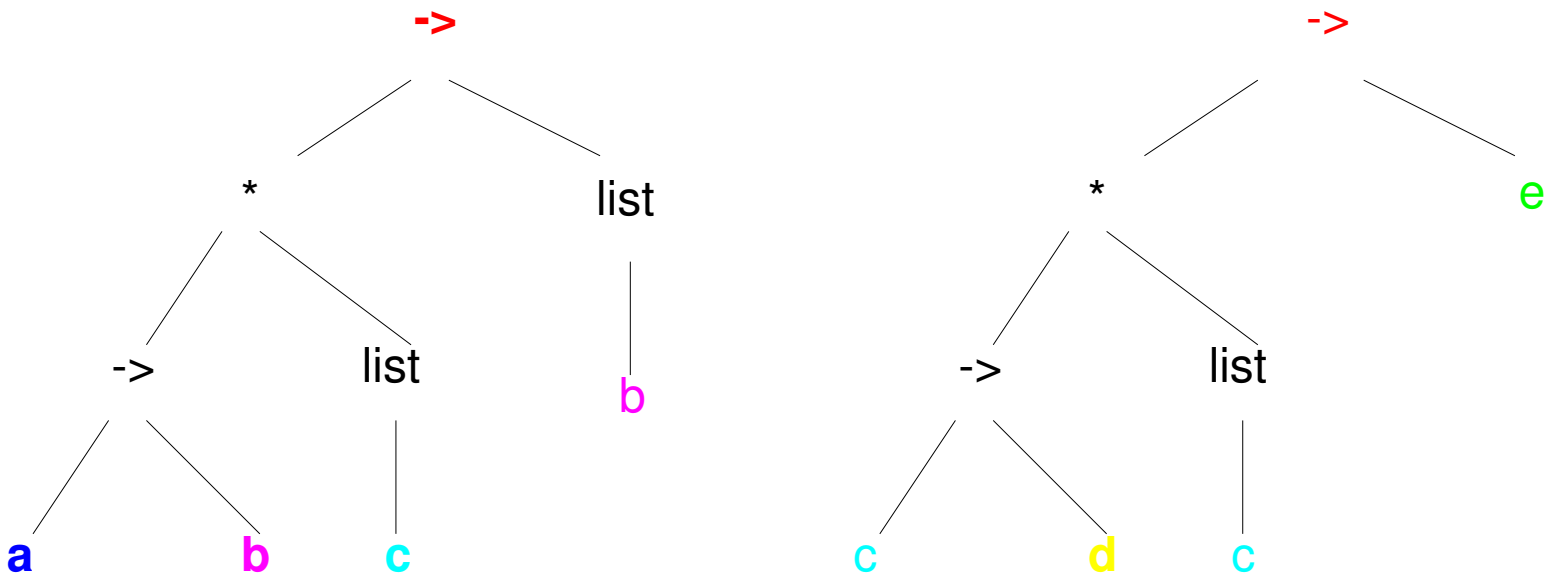
Entrée : deux graphes de représentation de type d'expression
Sortie : les deux graphes dont les nœuds sont regroupés en classes d'équivalence de types
    Chaque classe contient un nœud représentatif de la classe. Dans la mesure du possible,
        ce nœud représentatif n'est pas une variable.
Retour : vrai si l'unification est un succès, faux sinon

booléen unification(nœud n, nœud m)
    Soient s et t respectivement les nœuds représentatifs de la classe de n et de celle de m
    Si s = t
        Retourner vrai
    Sinon si s et t représentent le même type de base
        Retourner vrai
    Sinon si s et t sont des opérateurs de même type et de fils respectifs s_1, s_2, t_1, t_2
        Mettre s et t dans la même classe c.
        Si l'un des deux n'est pas une variable il devient représentant de c
        Sinon n'importe lequel des deux devient représentant de c
        Retourner unification(s_1,t_1) && unification(s_2,t_2)
    Sinon si s ou t est une variable
        Mettre s et t dans la même classe c.
        Si l'un des deux n'est pas une variable il devient représentant de c
        Sinon n'importe lequel des deux devient représentant de c
        Retourner vrai
    Sinon
        Retourner faux

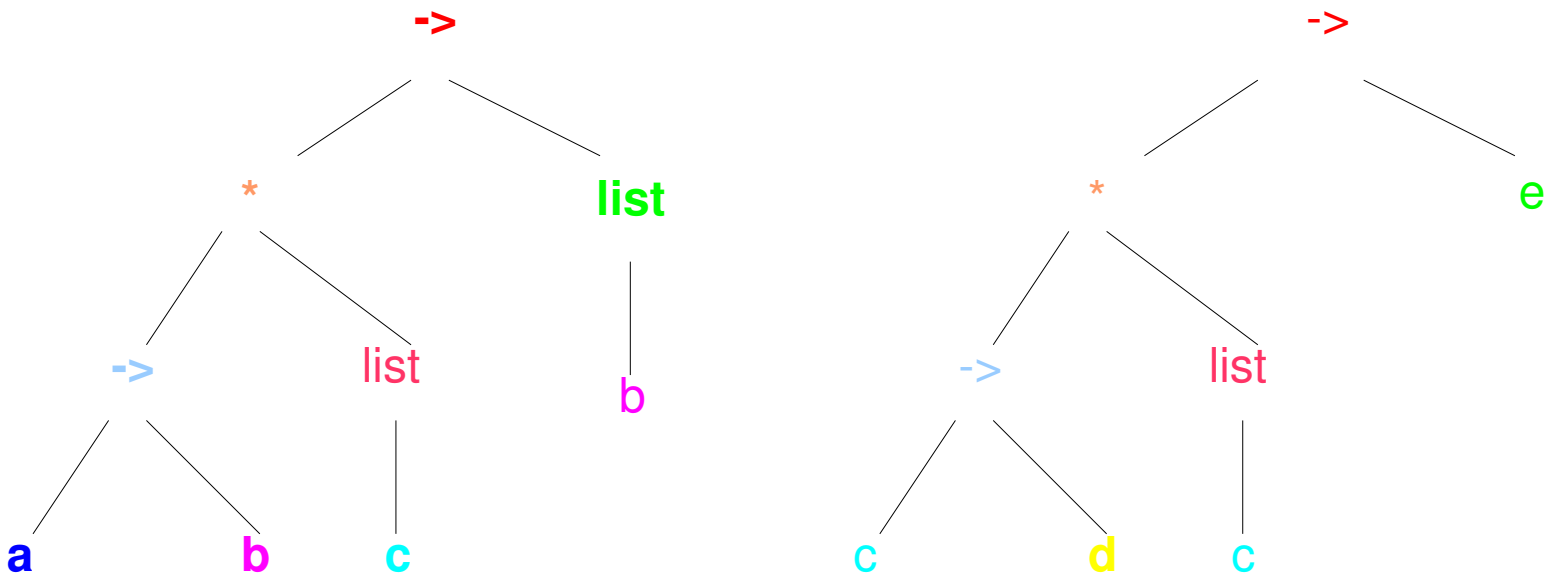
Exemple



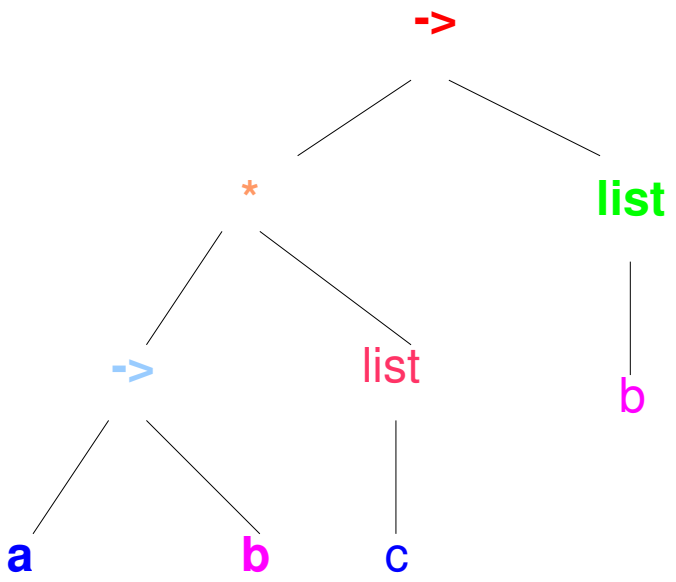
Exemple

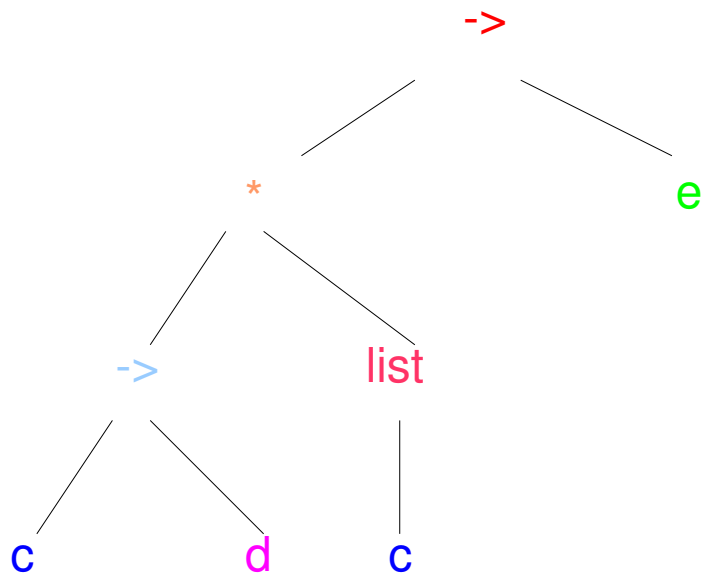


Exemple



Exemple



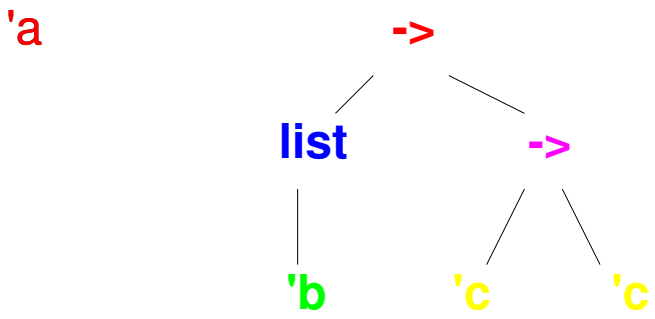


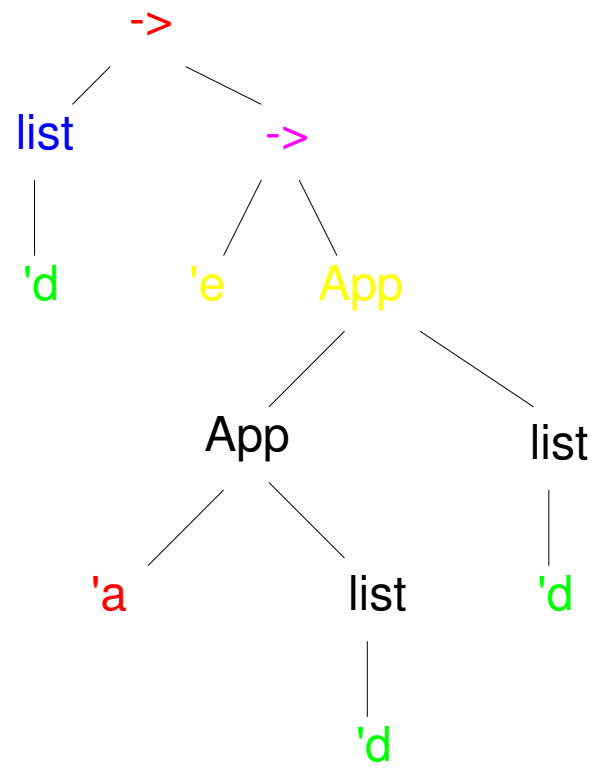
x	S(x)
a	a
b	b
c	a
d	b
e	list b

Exemple

let rec f = function [] -> (function x -> x) | x::m -> (function l -> f m x::l)

'a 'b 'c 'c 'd 'e 'd



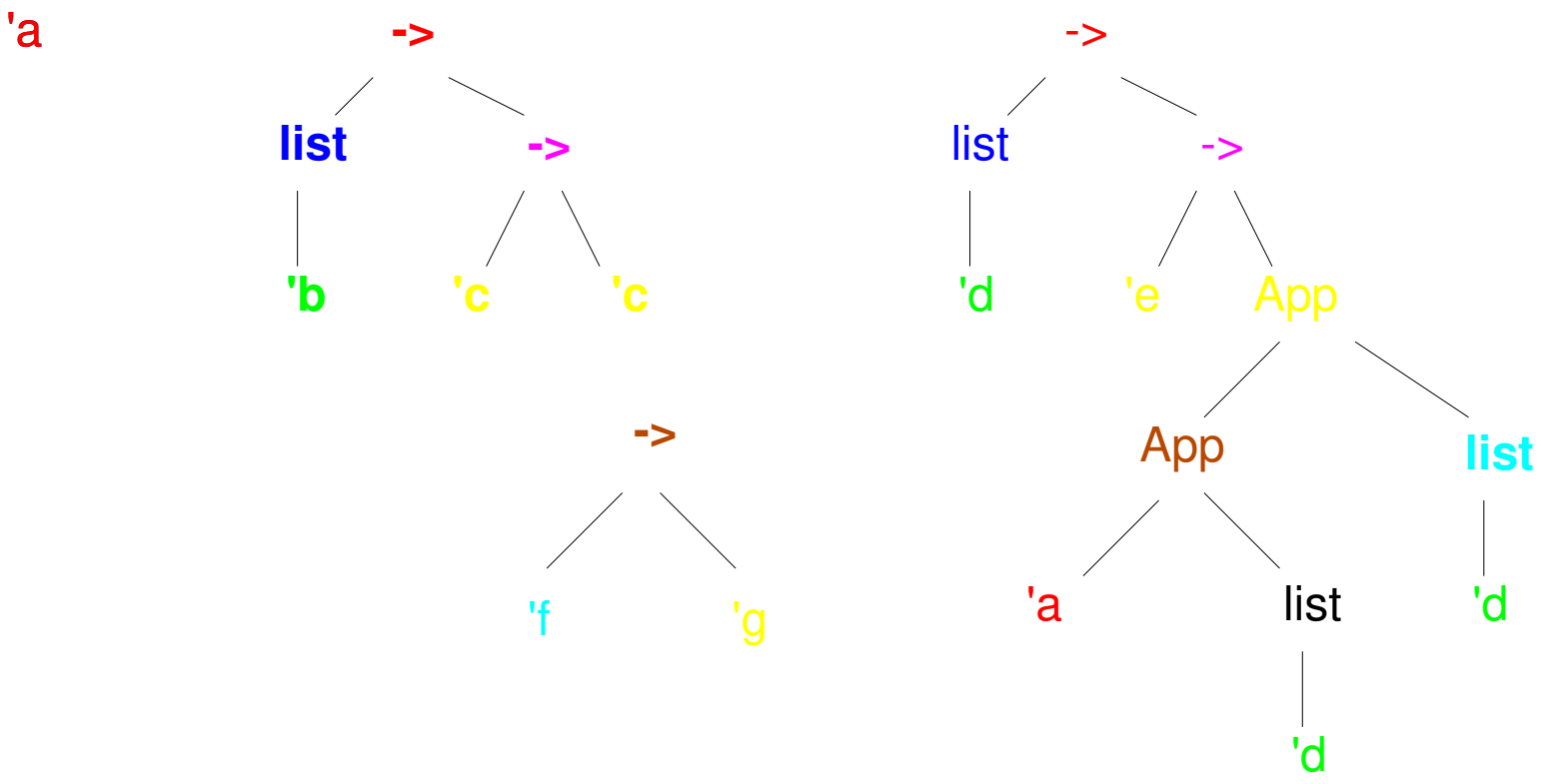


On ajoute à l'algorithme les règles suivantes :
Si s est de la forme App(f,g)
Soient 'a et 'b deux nouvelles inconnues
retourner unification(f, 'a->'b)
&& unification(g,'a)
&& unification(s,'b)

Exemple

let rec f = function [] -> (function x -> x) | x::m -> (function l -> f m x::l)

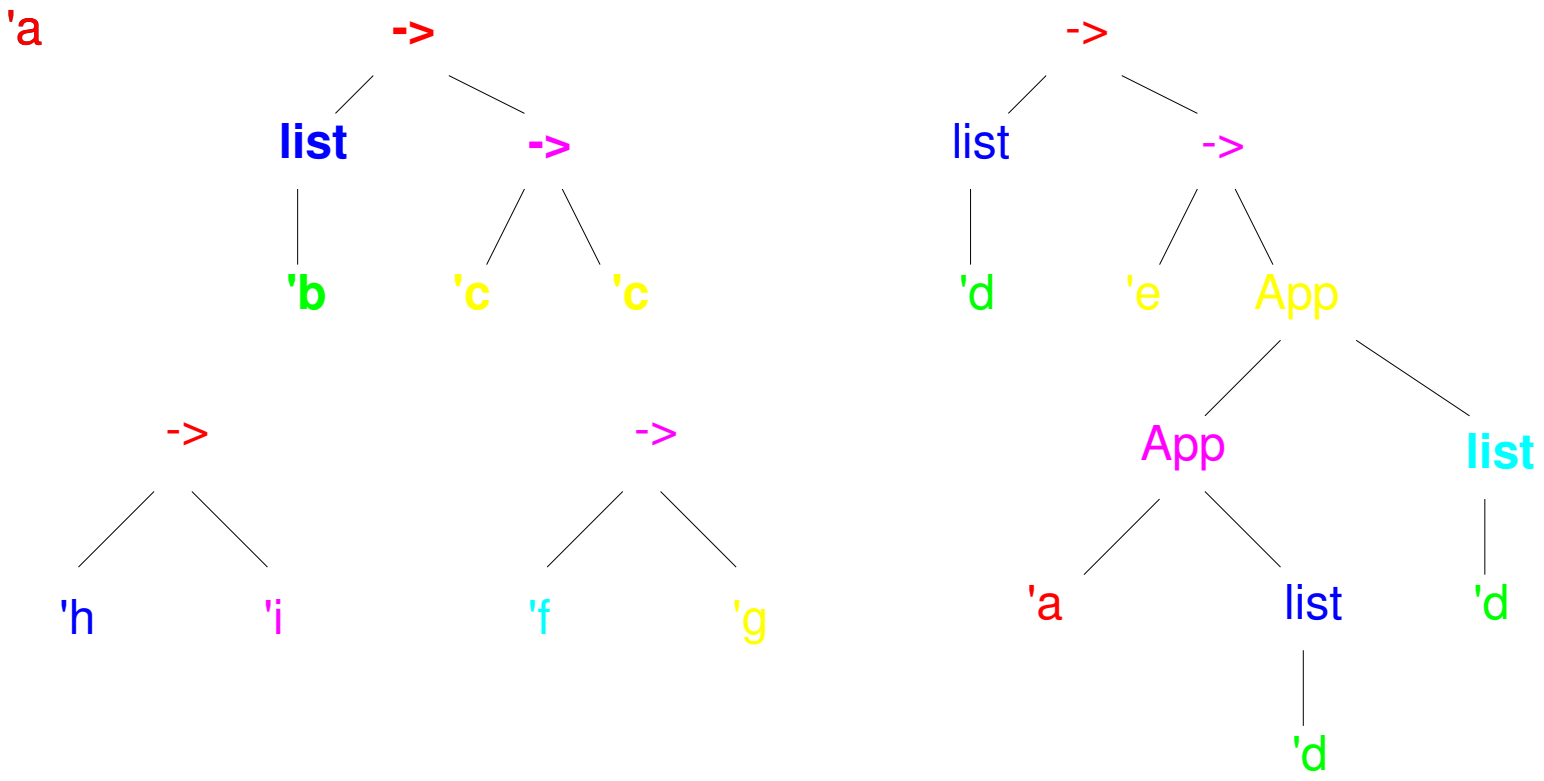
'a 'b 'c 'c 'd 'e 'd



Exemple

let rec f = function [] -> (function x -> x) | x::m -> (function l -> f m x::l)

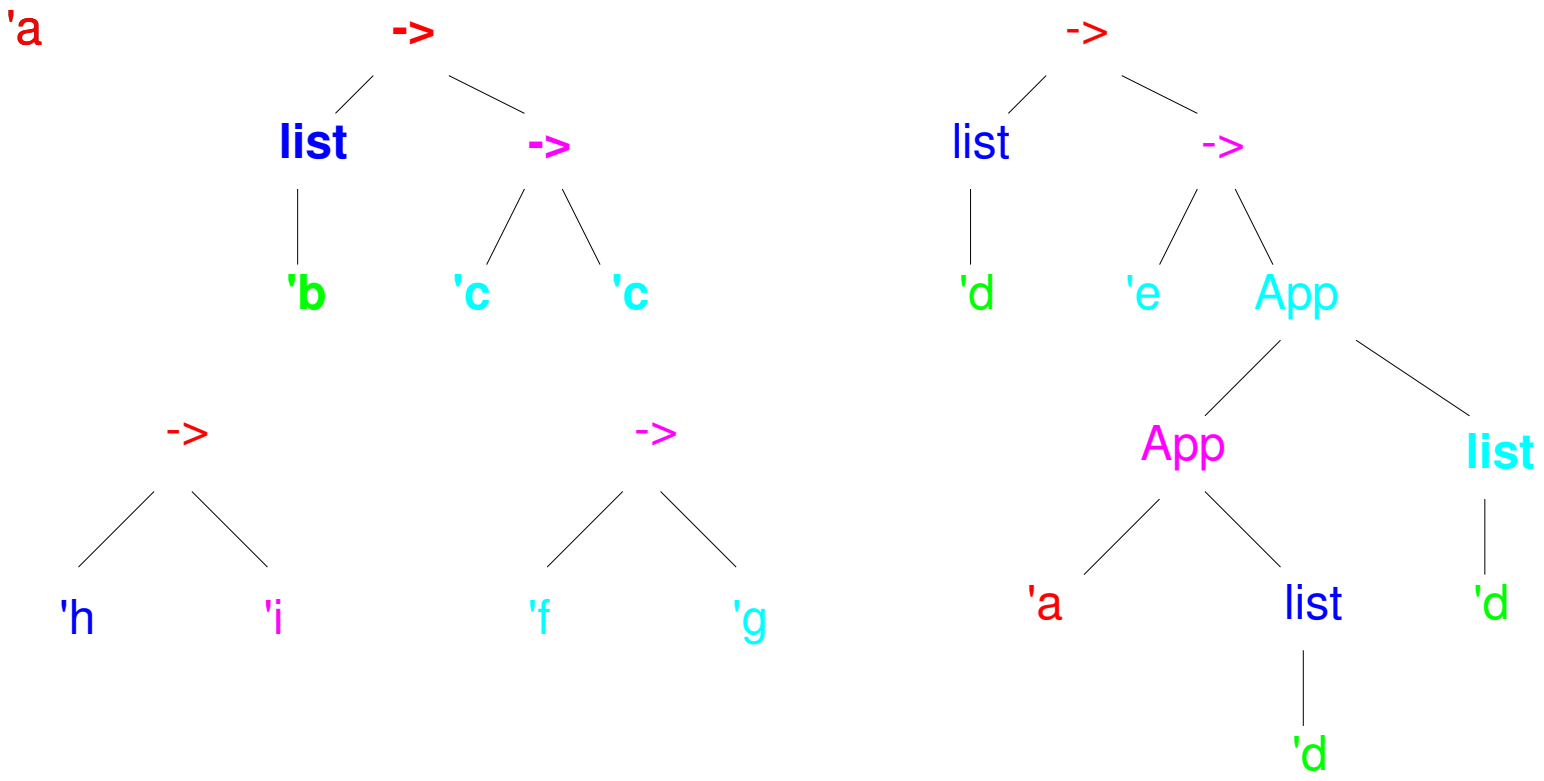
'a 'b 'c 'c 'd 'e 'd



Exemple

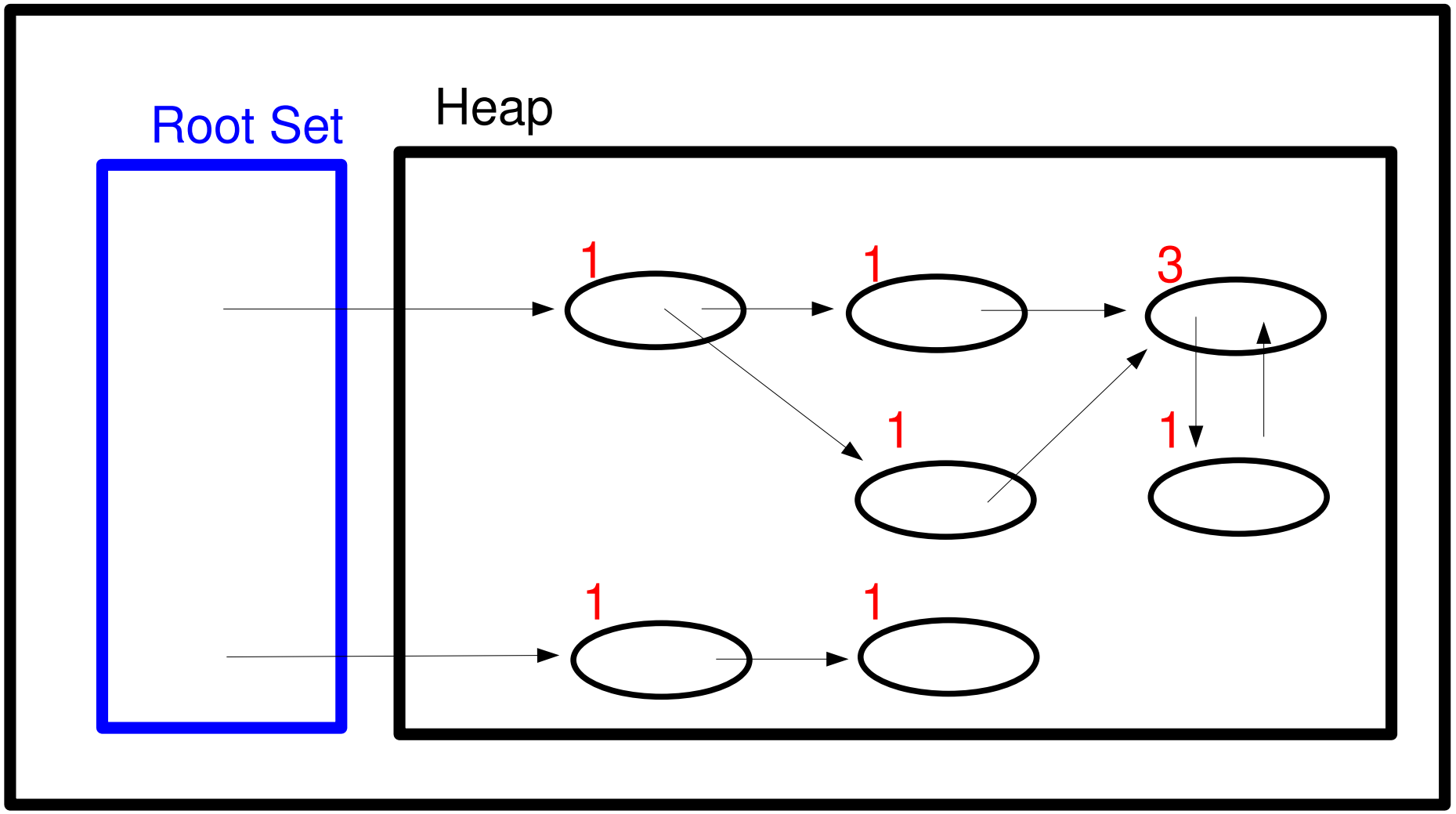
let rec f = function [] -> (function x -> x) | x::m -> (function l -> f m x::l)

'a 'b 'c 'c 'd 'e 'd

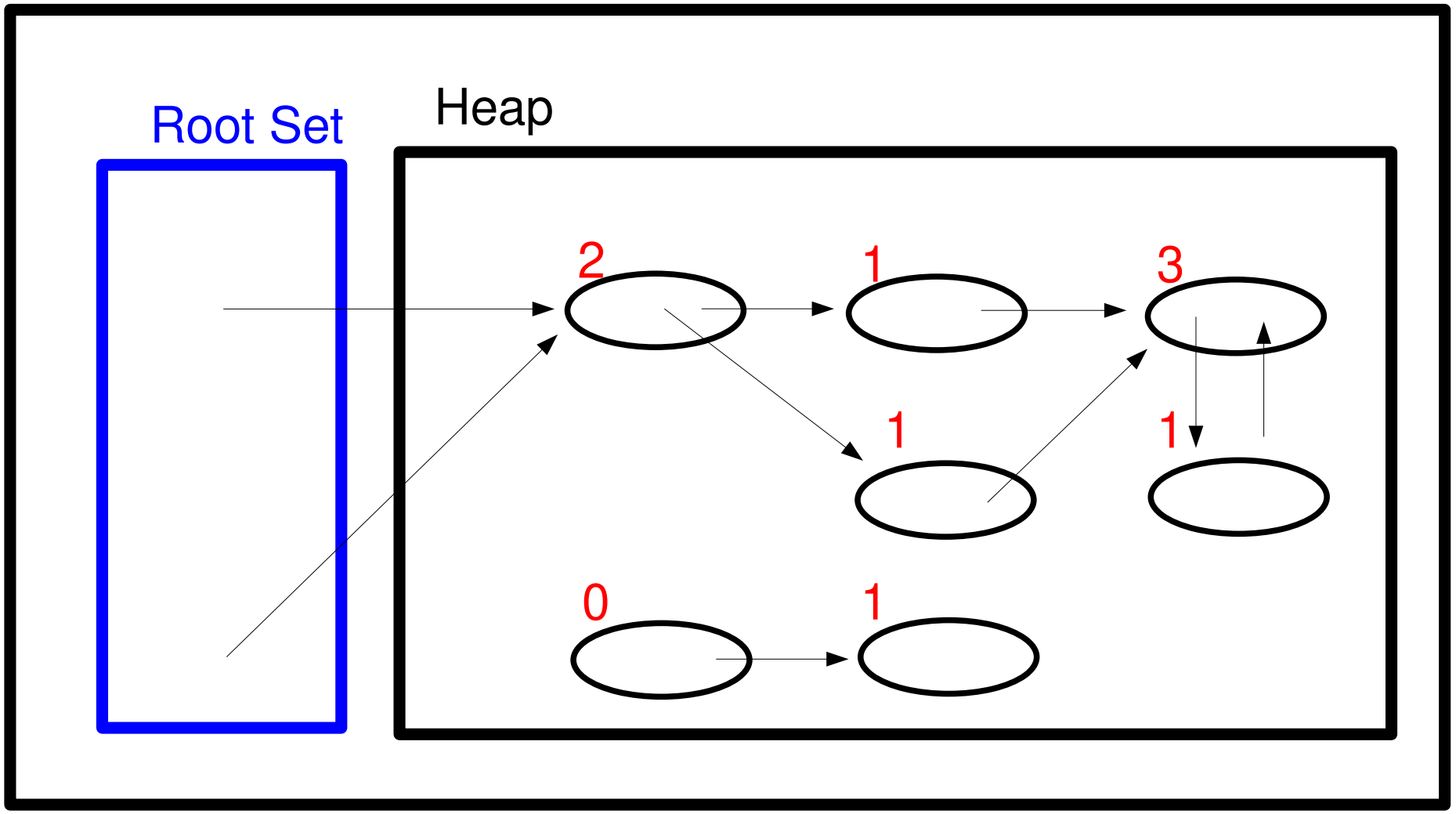


f : list 'b -> list 'b -> list 'b

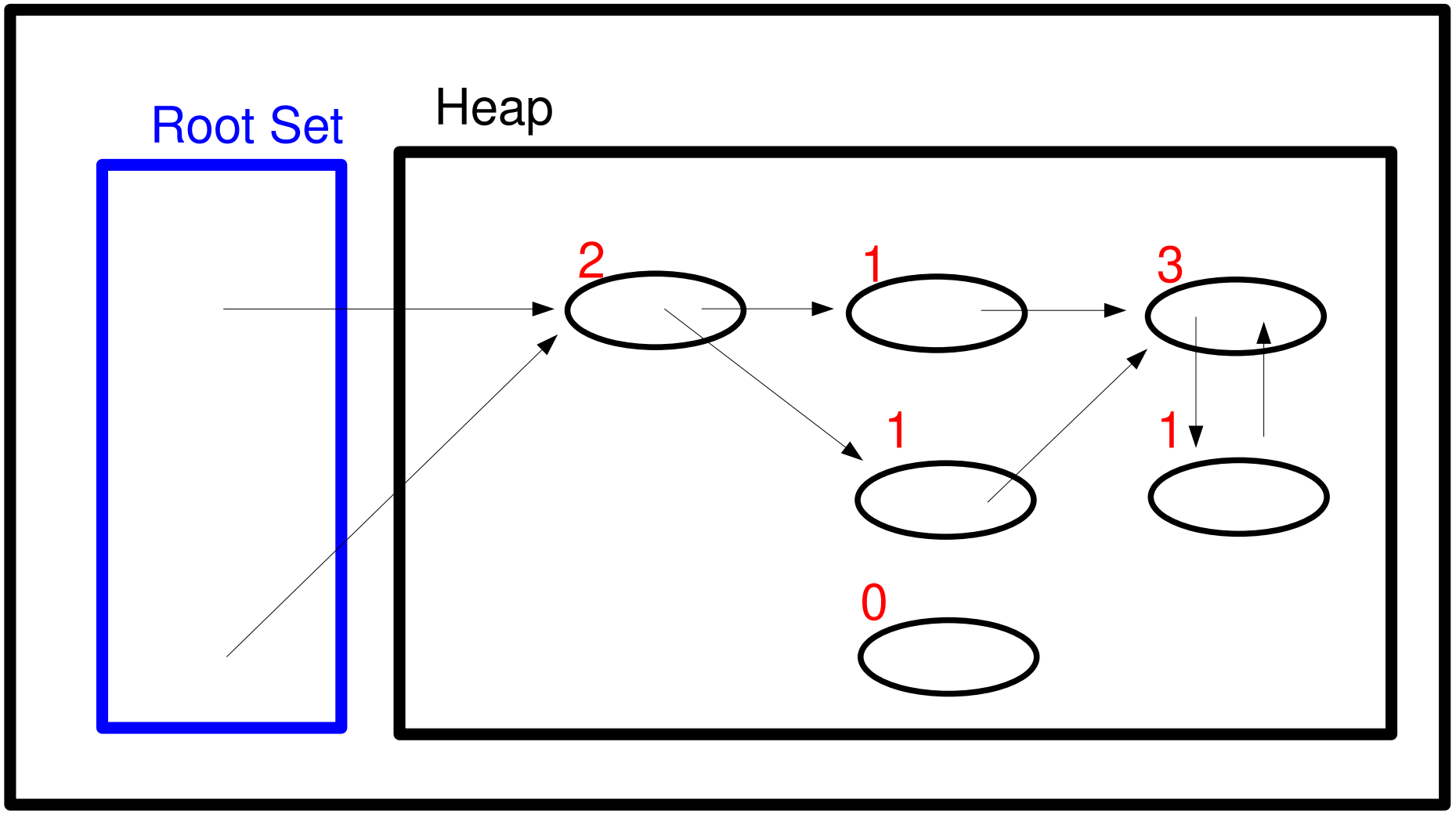
Ramasse-miettes : reference counting



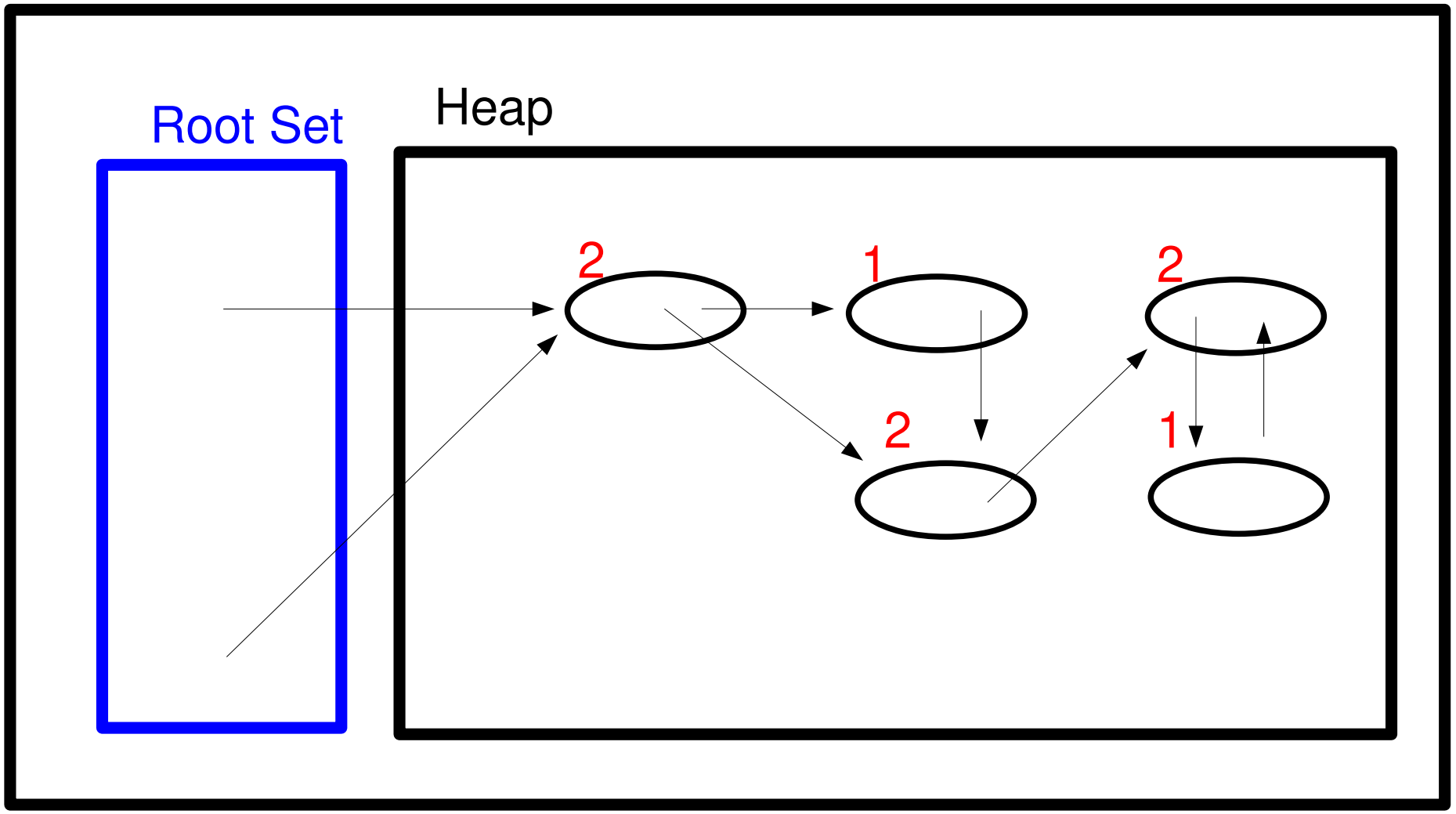
Ramasse-miettes : reference counting



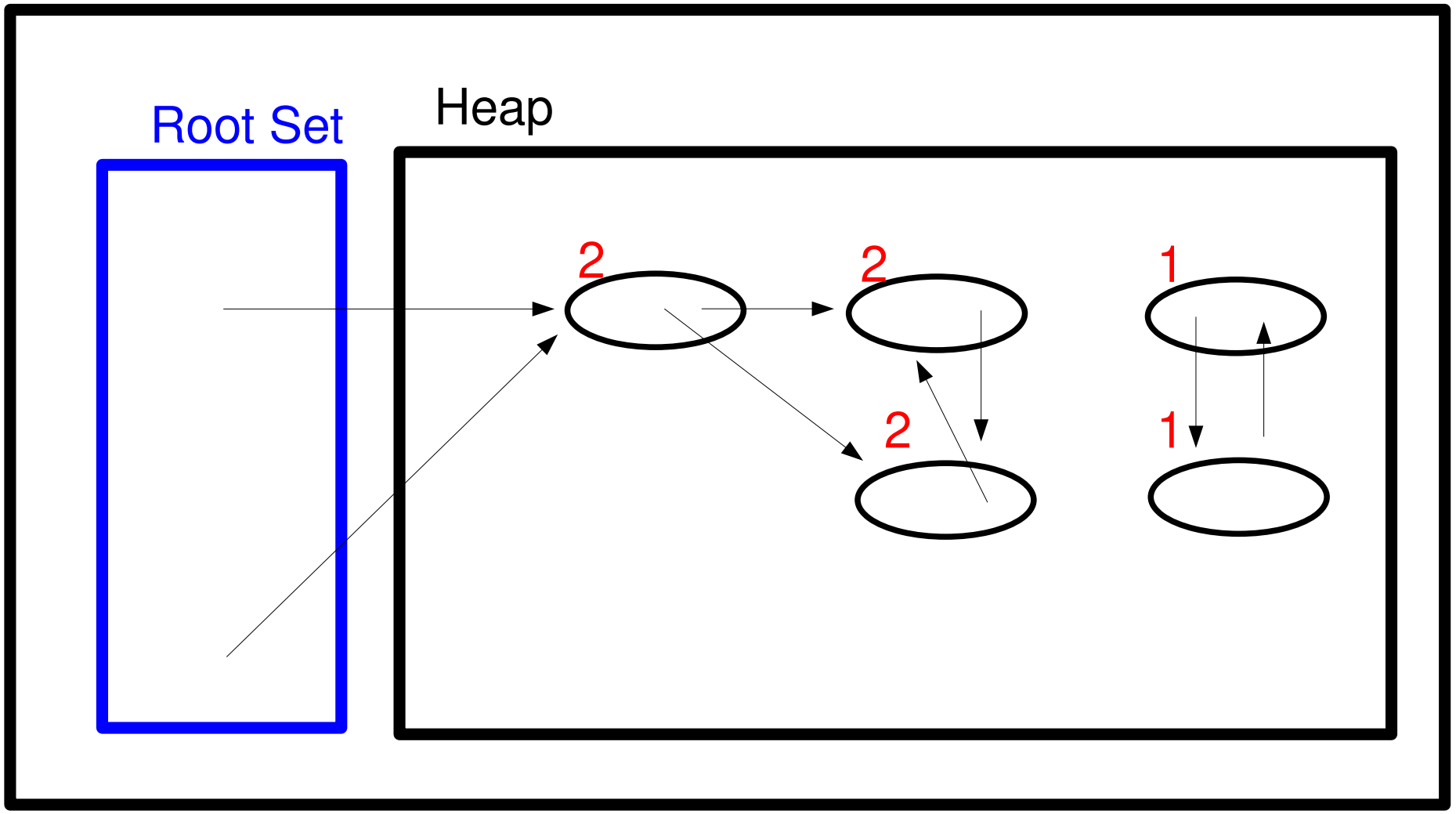
Ramasse-miettes : reference counting



Ramasse-miettes : reference counting



Ramasse-miettes : reference counting



Ramasse-miettes : reference counting

Algorithme utilisé par Perl 5

- Algorithme simple
- Données de gestion: un entier dans chaque objet
- Temps de calcul majoré par le nombre total d'objets
  (car libération transitive éventuelle)
- Temps de calcul important si les changements de références sont courants
  - Changement de l'entier
  - Test de l'entier
  - Autres opérations éventuelles
- Temps de calcul correct en moyenne, et sans arrêt intempestif de l'application
- Permet une récupération immédiate de l'espace mémoire
- Libération partielle pour les objets avec cycles !!
- Fragmentation ?

Représentation des types

Représentation en mémoire
- Taille
- Utilisation de l'espace

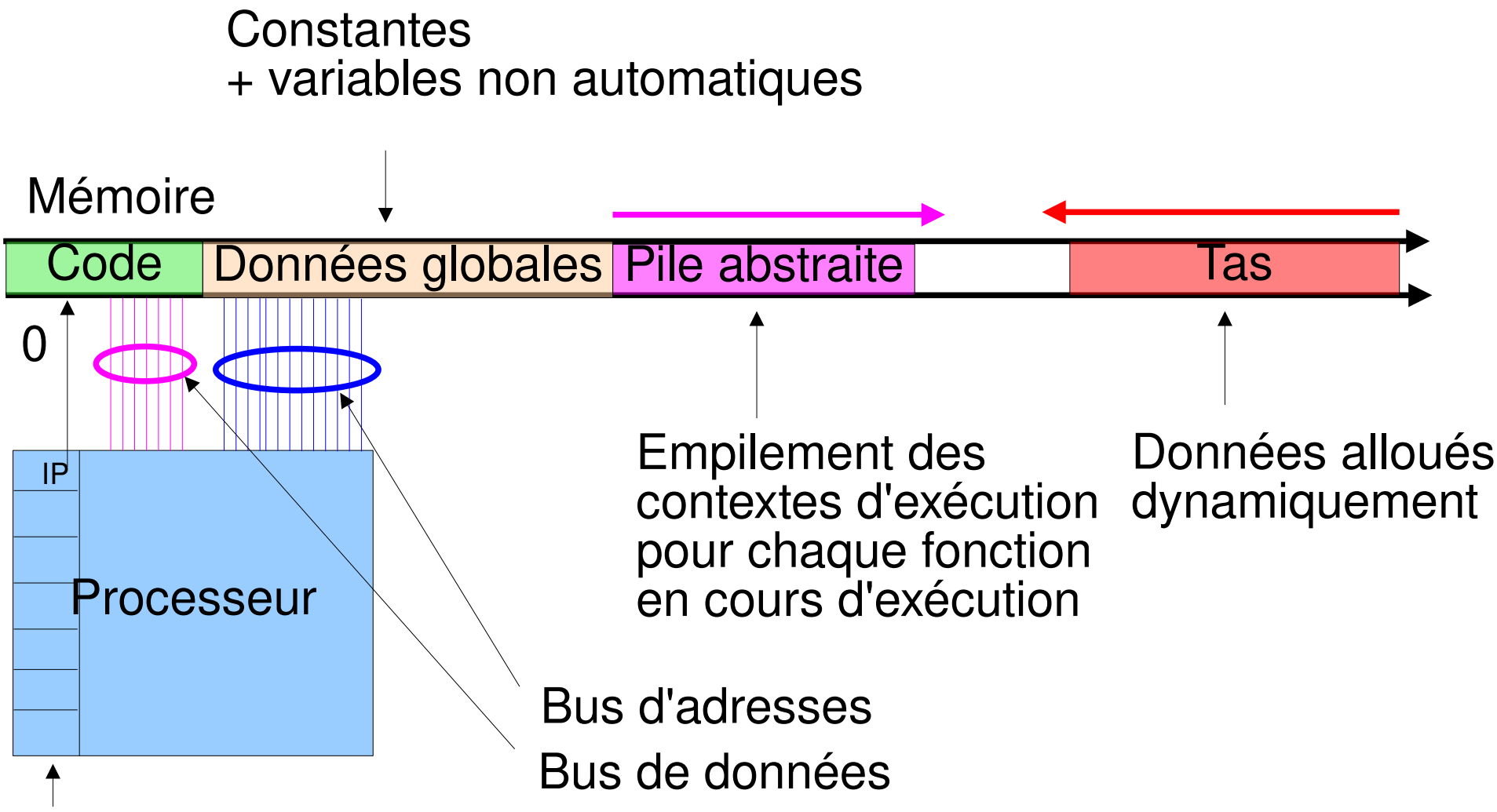
Caractéristiques de stockage
- Alignement
- Zones de stockage

Exemple : type « entier » de taille fixe
- Taille : 8 cellules de mémoire
- Représentation des valeurs positives ou nulles : base 2, little endian
- Représentation des valeurs négatives : complément à deux
- Alignement : les entiers doivent être stockés à des adresses multiples de 8
- Zone de stockage : toutes sont autorisées

Exemple : type « liste » de taille variable
- Taille : structure+taille de chacun des maillons
- Représentation : celle de la structure et celle des maillons
- Alignement : celui de la structure
- Zone de stockage : toutes pour la structure, tas pour les maillons

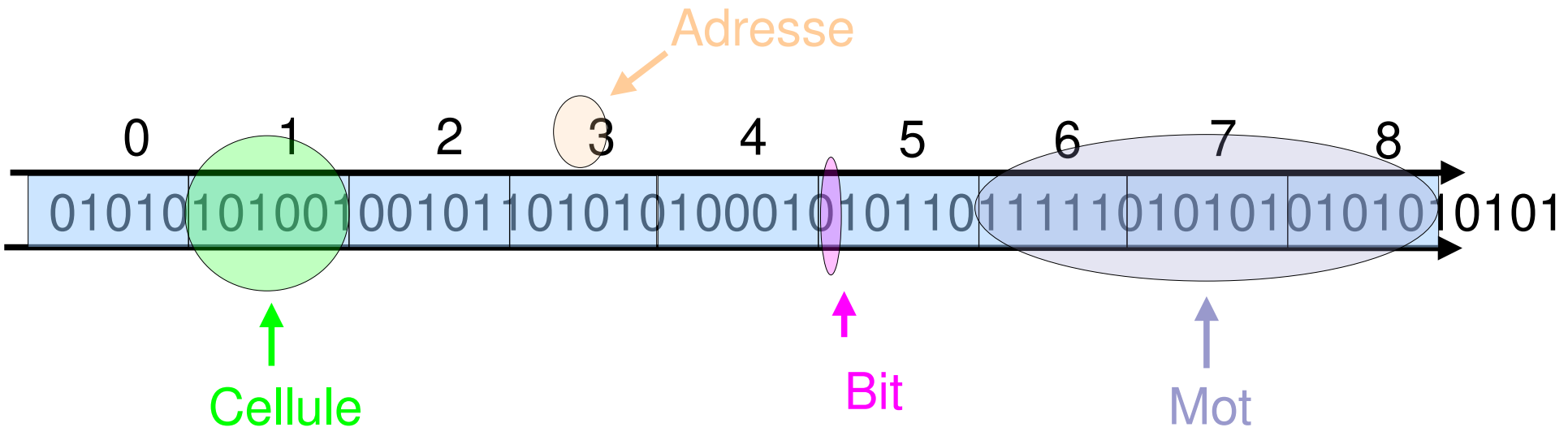
Quand ils existent, le compilateur peut utiliser les types implantés par le processeur.
Mais il peut aussi choisir de ne pas le faire : il faut alors qu'il les implante
- programmes plus lents et plus gros

Rappels sur la structure physique d'une machine



Registres (chaque registre contient un ***mot machine***)

La mémoire



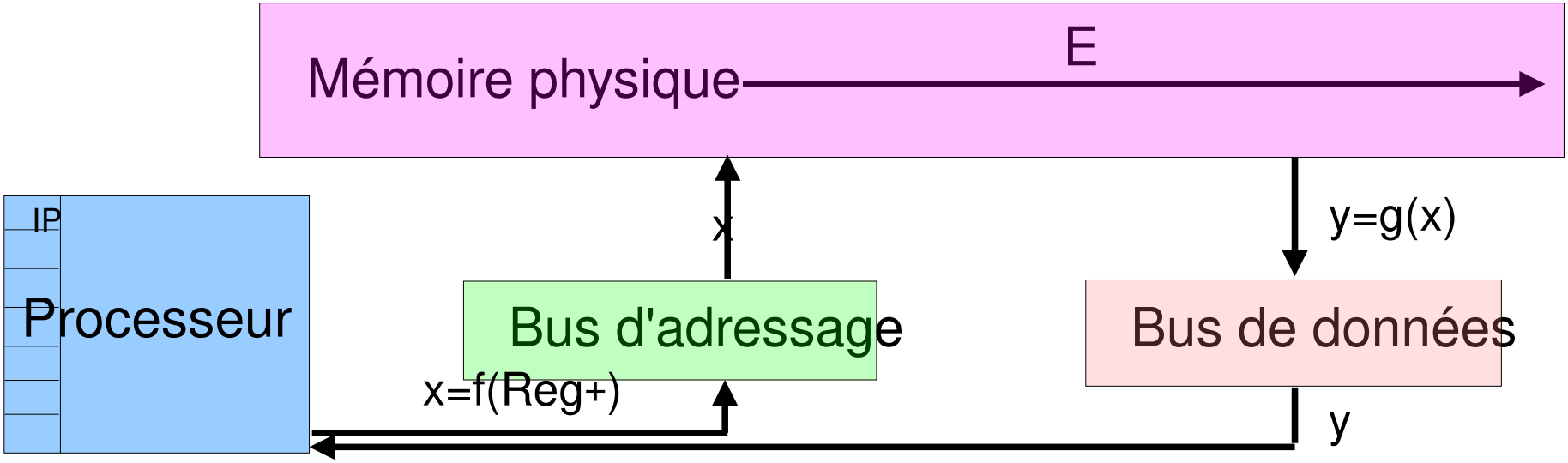
1 octet (octet)=cellule de 8 bits (bit)

byte = cellule

1 Ko=2^{10} octets (1024)
1 Mo=2^{10} Ko
1 Go=2^{10} Mo
1 To=2^{10} Go

Adressage de la mémoire par le processeur (modèle simplifié)

Deux espaces d'adressage:
- E: celui de la mémoire physique
- Reg+: celui du processeur

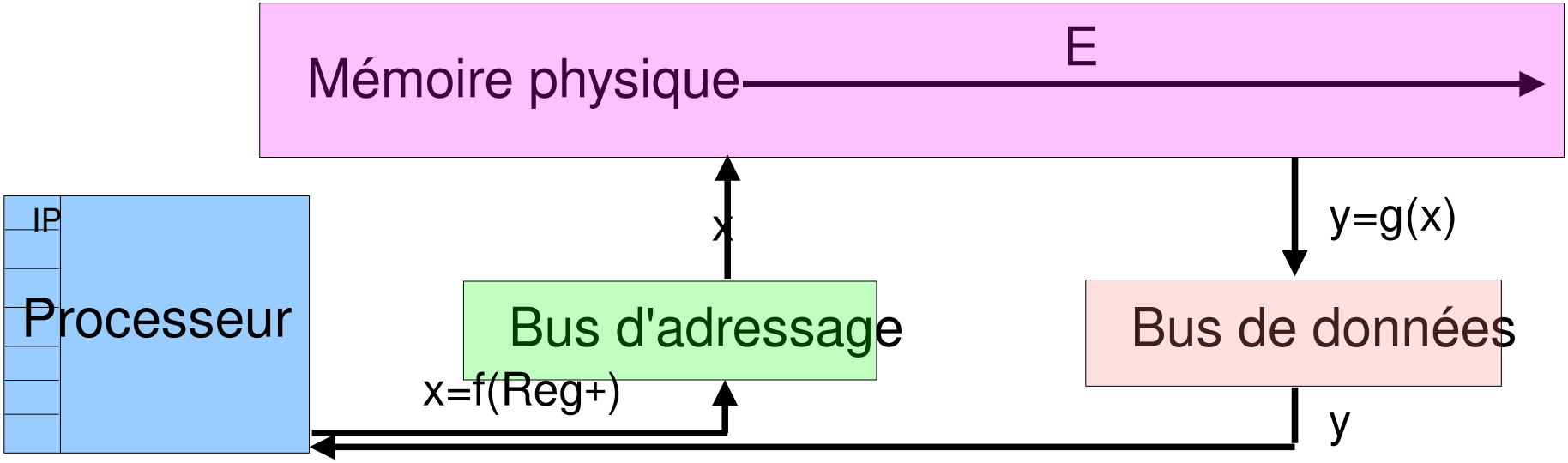


f: application de Reg+ dans E. Surjective, mais pas injective

g: fonction de E dans ensemble des cellules.
Ni injective ni surjective

Adressage de la mémoire par le processeur (modèle simplifié)

Deux espaces d'adressage:
- E: celui de la mémoire physique
- Reg+: celui du processeur



f: application de Reg+ dans E. Surjective, mais pas injective

g: fonction de E dans ensemble des cellules.
Ni injective ni surjective

Adressage de la mémoire par le processeur: un exemple réel

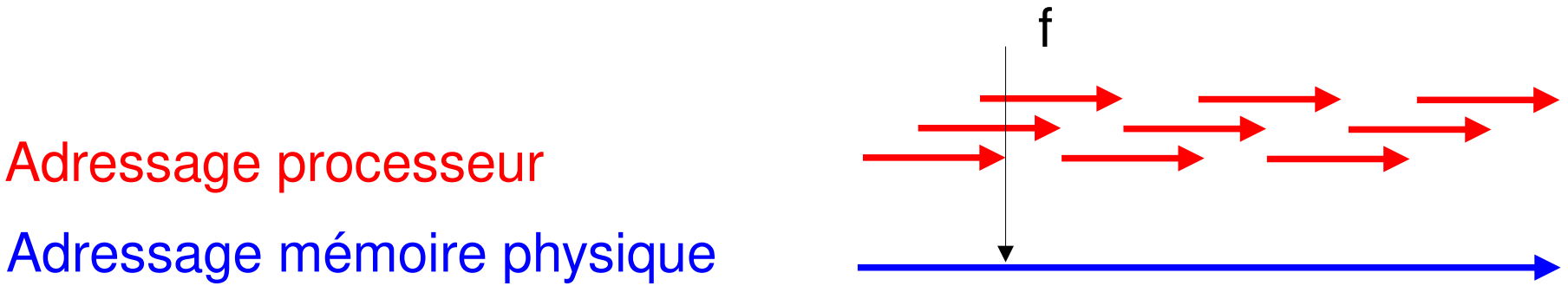
Processeur: i8086 (Registres 16 bits, cellule=octet, mémoire adressable: 1Mo,1978)

CS: 1010010101001010

IP: 1011101010100101 +

f((CS,IP)): 10110000111101000101

$f^{-1}f((CS,IP))$ pas unique: deux adresses processeurs différentes peuvent désigner la même cellule mémoire !!



Adressage de la mémoire par le processeur: un autre exemple

Processeur: ? (registres 16 bits, cellule=octet, mémoire adressable 32Ko)

Registre d'adresse: RA ⟷ instruction de transfert ⟷ Accumulateur: AX

Bit de poids faible de RA: toujours à 0 pour les adresses valides

1: génération d'une erreur (exception)

Le fait de calculer une mauvaise adresse (transfert de AX vers RA d'une valeur d'adresse invalide (terminant par 1)) peut provoquer une erreur au niveau du processeur, sans même avoir mis cette adresse sur le bus d'adresses.

Le mot machine

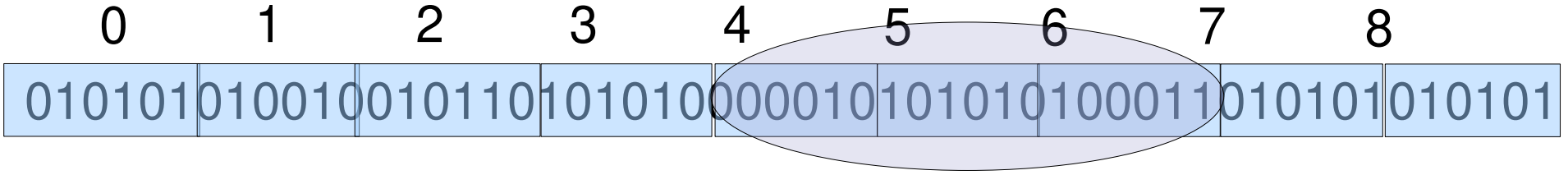
Groupe de cellules consécutives sur lequel le processeur fait la plupart de ses opérations.

Taille: machine dépendante (registre)

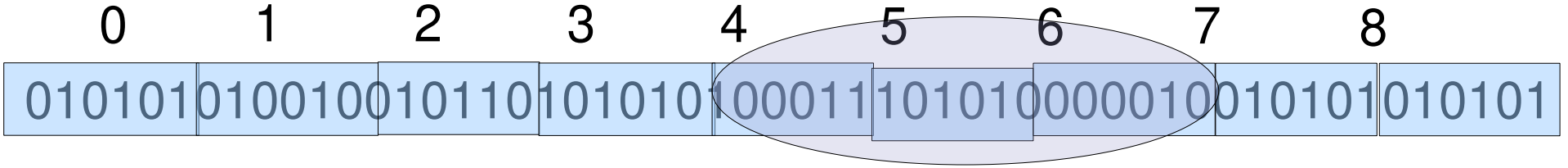
Ordre de placement des cellules dans le groupe: machine dépendant

Cellules de 6 bits, Mots de 3 cellules, $(10915)_{10}=(000010101010100011)_2$

Gros boutistes: on commence par le poids fort



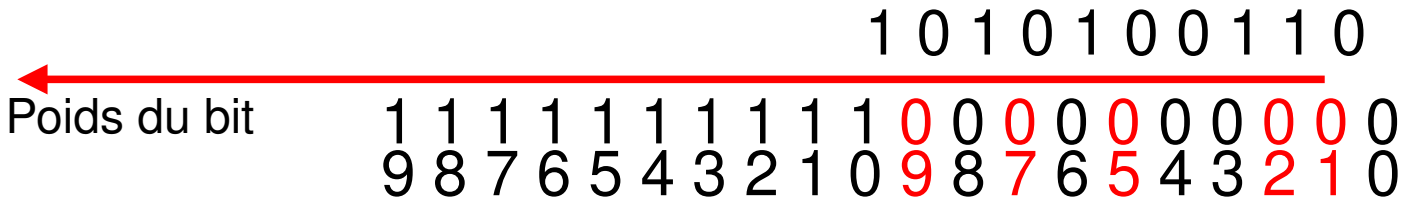
Petits boutistes: on commence par le poids faible



Représentation des types entiers

Valeurs positives ou nulles: codage en base 2

Ex: $678 = 2^9+2^7+2^5+2^2+2^1$



Valeurs négatives: codage machine dépendant

- codage avec bit de signe
- codage avec complément à deux
- ....

Intervalle des valeurs représentées

Codage sur n (machine dépendant) bits

Codage de 2^n valeurs

Valeurs entières naturelles: $[0,2^n[$

Valeurs entières relatives:
Le nombre de valeurs positives et négatives dépend du codage choisi (machine dépendant). En général, le codage est bon et le nombre de valeurs positives ou nulles est égal au nombre de valeurs négatives.
L'intervalle des valeurs représentées est alors $[-2^{n-1},2^{n-1}[$

Un codage avec bit de signe

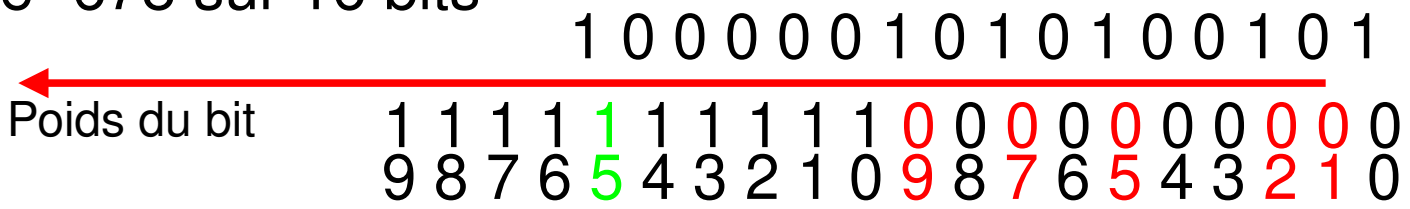
Codage de x

Un bit (par exemple le plus à gauche) est réservé pour le signe

Une valeur de ce bit (par exemple 1) indique que la valeur est négative, l'autre valeur qu'elle est positive ou nulle.

Le reste des bits sert à coder:
- x si x>=0
- abs(x)-1 si x<0

Ex: codage de -678 sur 16 bits

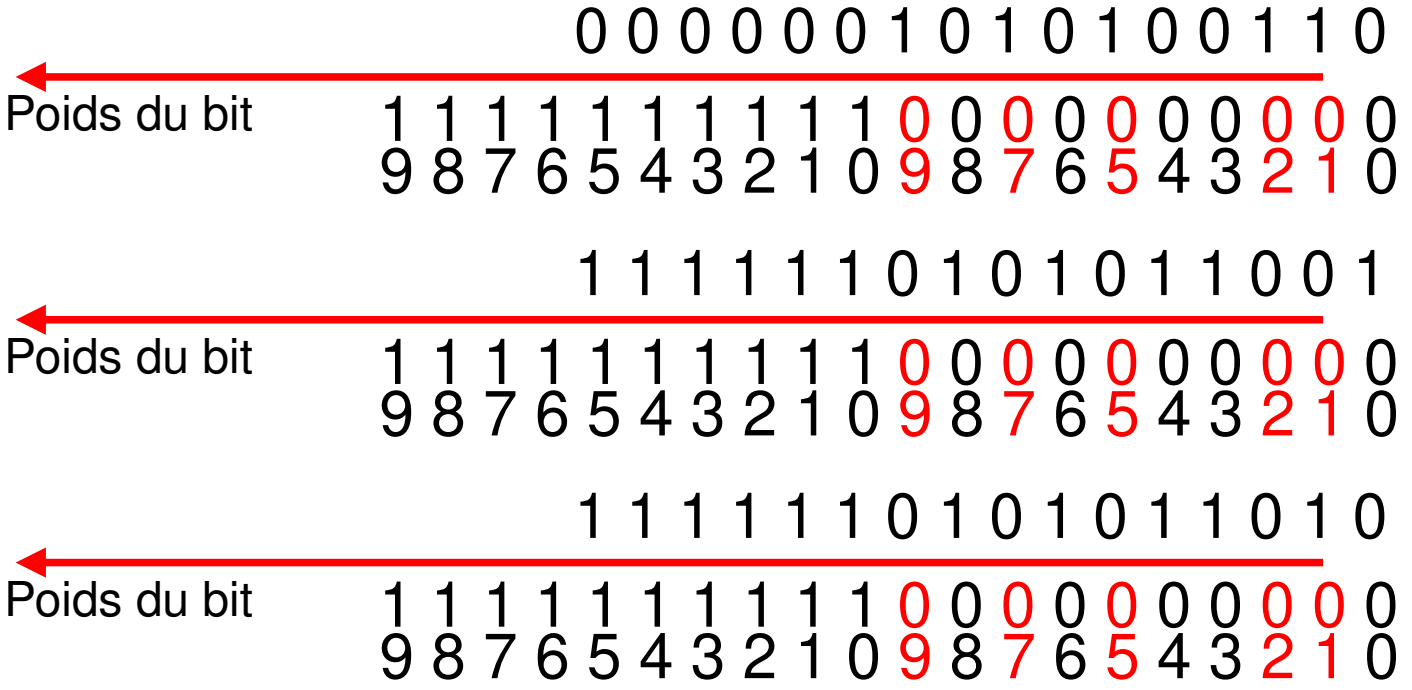


Le codage par complément à deux

De loin le plus utilisé

Complément à deux = complément à un + 1

Ex: codage de – 678 sur 16 bits



Rem: le bit de poids fort donne le signe

Justification du complément à deux

- L'algorithme d'addition est le même (bit à bit) que les valeurs soient positives ou négatives
- Le calcul du dépassement de capacité est simple
  - si les deux opérandes ne sont pas de même signe, débordement impossible
  - si le signe du résultat n'est pas celui des deux opérandes, débordement
  - si le bit de retenue entrante dans le bit de poids fort est différent du bit de retenue sortante, débordement (équivalent au principe précédent)

Exemples d'addition en complément à deux avec dépassement

8 bits, valeurs entières relatives, valeurs négatives: complément à deux

116+96=212

```
  0 1 1 1 0 1 0 0
+ 0 1 0 0 1 0 0 0
-----------------
  1 0 1 1 1 1 0 0
```

212 en codage non signé

-68 en codage par complément à deux

-12+-56=-68

```
  1 1 1 1 0 1 0 0
+ 1 1 0 0 1 0 0 0
-----------------
1 1 0 1 1 1 1 0 0
```

Bit perdu

Opérations sur les valeurs arithmétiques

Opérateurs arithmétiques:

Opérateurs unaires: + -

Opérateurs binaires: + - * / %

Opérateurs de manipulation de bits: ~ << >> & | ^

~x: négation bit à bit
x << y: décalage des bits de x de y crans vers la gauche
x >> y: décalage des bits de x de y crans vers la droite
x & y: ET bit à bit
x | y: OU bit à bit
x ^ y: OU exclusif bit à bit

entiers uniquement

Sémantique des opérations sur les valeurs arithmétiques

Opérateurs arithmétiques (-+-+*/%): usuelles

Opérateurs de manipulation de bits

~a: complément à un (négation bit à bit)
$\sim (1\ 0\ 0\ 0\ 1\ 1\ 1\ 0)_2 = (0\ 1\ 1\ 1\ 0\ 0\ 0\ 1)_2$

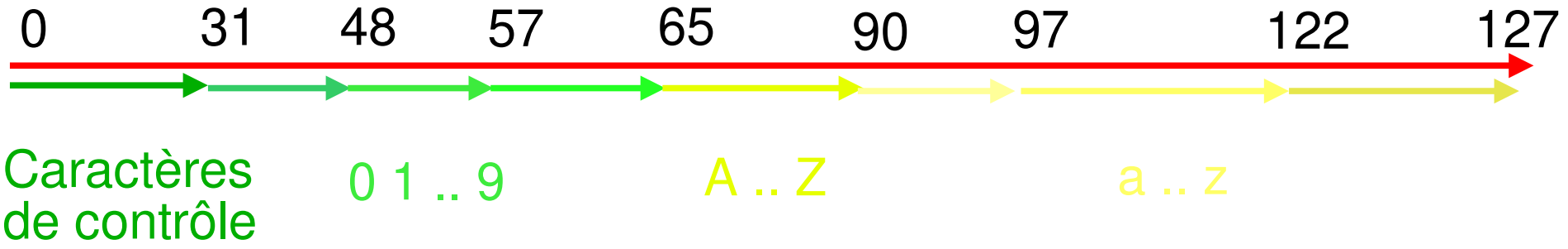
a & b: ET bit à bit
$(1\ 0\ 0\ 0\ 1\ 1\ 1\ 0)_2 \ \&\ (1\ 0\ 0\ 1\ 1\ 1\ 1\ 1)_2 = (1\ 0\ 0\ 0\ 1\ 1\ 1\ 0)_2$

a | b: OU bit à bit
$(1\ 0\ 0\ 0\ 1\ 1\ 1\ 0)_2 \ |\ (1\ 0\ 0\ 1\ 1\ 1\ 1\ 1)_2 = (1\ 0\ 0\ 1\ 1\ 1\ 1\ 1)_2$

a ^ b: OU exclusif bit à bit
$(1\ 0\ 0\ 0\ 1\ 1\ 1\ 0)_2 \ \hat{}\ (1\ 0\ 0\ 1\ 1\ 1\ 1\ 1)_2 = (0\ 0\ 0\ 1\ 0\ 0\ 0\ 1)_2$

Le code ASCII

Code ASCII originel: 7 bits



Code ASCII étendu: 8 bits

>=128: caractères semi-graphiques

Table ASCII

0 NUL	16 DLE	32 ESP	48 0	64 @	80 P	96 `	112 p
1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
4 EOT	20 DC4	36 $	52 4	68 D	84 T	100 d	116 t
5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
9 TAB	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124 |
13 RC	29 GS	45 -	61 =	77 M	93]	109 m	125 }
14 SO	30 RS	46 .	62 >	78 N	94 _	110 n	126 ~
15 SI	31 US	47 /	63 ?	79 O	95 ^	111 o	127 DEL

Le code EBCDIC

Code EBCDIC (IBM): 8 bits

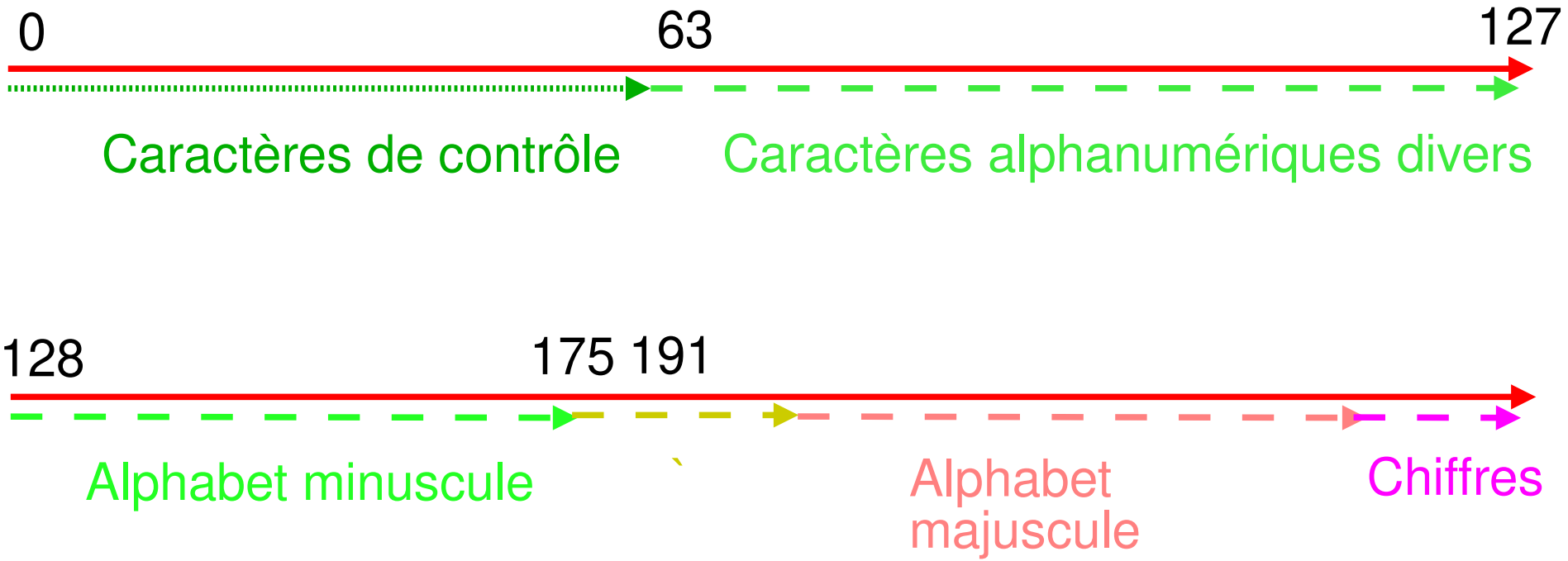


Table EBCDIC

0	NUL	16	DLE	32	DS	48		64	SP	80	&	96	-	112	
1	SOH	17	DC1	33	SOS	49		65		81		97	/	113	
2	STX	18	DC2	34	FS	50	SYN	66		82		98		114	
3	ETX	19	TM	35		51		67		83		99		115	
4	PF	20	RES	36	BYP	52	PN	68		84		100		116	
5	HT	21	NL	37	LF	53	RS	69		85		101		117	
6	LC	22	BS	38	ETB	54	UC	70		86		102		118	
7	DEL	23	IL	39	ESC	55	EOT	71		87		103		119	
8		24	CAN	40		56		72		88		104		120	
9		25	EM	41		57		73		89		105		121	
10	SMM	26	CC	42	SM	58		74	¢	90	!	106		122	:
11	VT	27	CU1	43	CU2	59	CU3	75	.	91	$	107	,	123	#
12	FF	28	IFS	44		60	DC4	76	<	92	*	108	%	124	@
13	CR	29	IGS	45	ENQ	61	NAK	77	(	93	)	109	_	125	'
14	SO	30	IRS	46	ACK	62		78	+	94	;	110	>	126	=
15	SI	31	IUS	47	BEL	63	SUB	79	|	95	¬	111	?	127	"

Table EBCDIC

128	144	160	176	192	208	224	240 0
129 a	145 j	161	177	193 A	209 J	225	241 1
130 b	146 k	162 s	178	194 B	210 K	226 S	242 2
131 c	147 l	163 t	179	195 C	211 L	227 T	243 3
132 d	148 m	164 u	180	196 D	212 M	228 U	244 4
133 e	149 n	165 v	181	197 E	213 N	229 V	245 5
134 f	150 o	166 w	182	198 F	214 O	230 W	246 6
135 g	151 p	167 x	183	199 G	215 P	231 X	247 7
136 h	152 q	168 y	184	200 H	216 Q	232 Y	248 8
137 i	153 r	169 z	185 `	201 I	217 R	233 Z	249 9
138	154	170	186	202	218	234	250
139	155	171	187	203	219	235	251
140	156	172	188	204	220	236	252
141	157	173	189	205	221	237	253
142	158	174	190	206	222	238	254
143	159	175	191	207	223	239	255

Norme IEEE 754-1985

- 4 formats de représentation
  - Simple précision (32 bits) :
    - 1 bit de signe,
    - 8 bits d'exposant,
    - 23 bits de mantisse
    - 1 bit implicite
  - Simple précision étendue (plus de 43 bits : obsolète)
  - Double précision (64 bits : 1 sgn, 11 exp, 52 mantisse, 1 implicite)
  - Double précision étendue (plus de 79 bits)
- Valeurs spéciales
- Modes d'arrondi (pris en charge dans gcc par des options)
  - Au plus près
  - Vers 0
  - Vers +Infty
  - Vers -Infty
- Opérations arithmétiques : +-*/ sqrt arrondi reste
- Cinq exceptions (pour détecter les erreurs, prises en charge dans gcc par options)

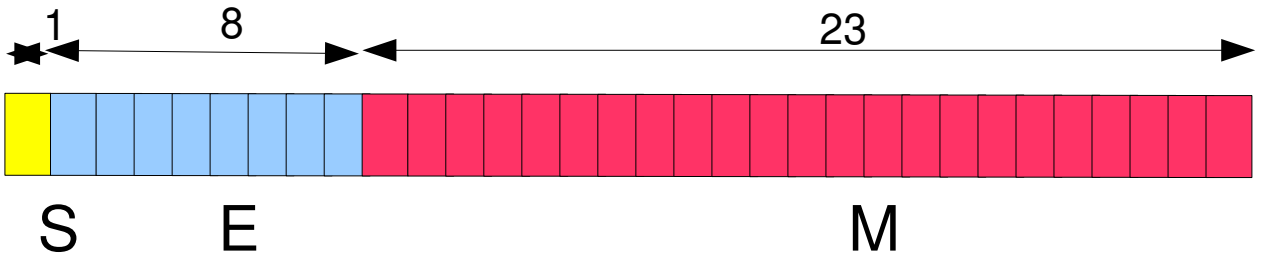
Norme IEEE 754-1985
Valeurs spéciales

- Not A Number (NaN)
- +0.0
- -0.0
- +Infinity
- -Infinity

+	NaN	-Infinity	+Infinity	y	0
NaN	NaN	NaN	NaN	NaN	NaN
-Infinity	NaN	-Infinity	NaN	-Infinity	-Infinity
+Infinity	NaN	NaN	+Infinity	+Infinity	+Infinity
x	NaN	-Infinity	+Infinity	x+y	x
0	NaN	-Infinity	+Infinity	y	0

```
float x=0.0/0.0 ; /* Correct en IEEE 754 !! */
printf(« %s », x==x ? « yes » : « no ») ; /* no en IEEE 754 !! */
```

Norme IEEE 754-1985
Représentation de la précision simple



- Si E différent de 0

$$x=(-1)^S*2^{E-127}*(1+M)$$

- Si E = 0

$$x=(-1)^S*2^{-127}*M$$

Ex : $14.4=2^3*(1+0.8)=2^3*(1+(1.6*2^{-1}))=2^3*(1+((1+0.6)*2^{-1}))$

$2^3*(1+((1+(1+0.2)*2^{-1})*2^{-1}))$

$2^3*(1+((1+(1+(1.6*2^{-3}))*2^{-1})*2^{-1}))$

S=1, E=130=10000010, M=11001100...1100...

Norme IEEE 754-1985
Représentation de la précision simple

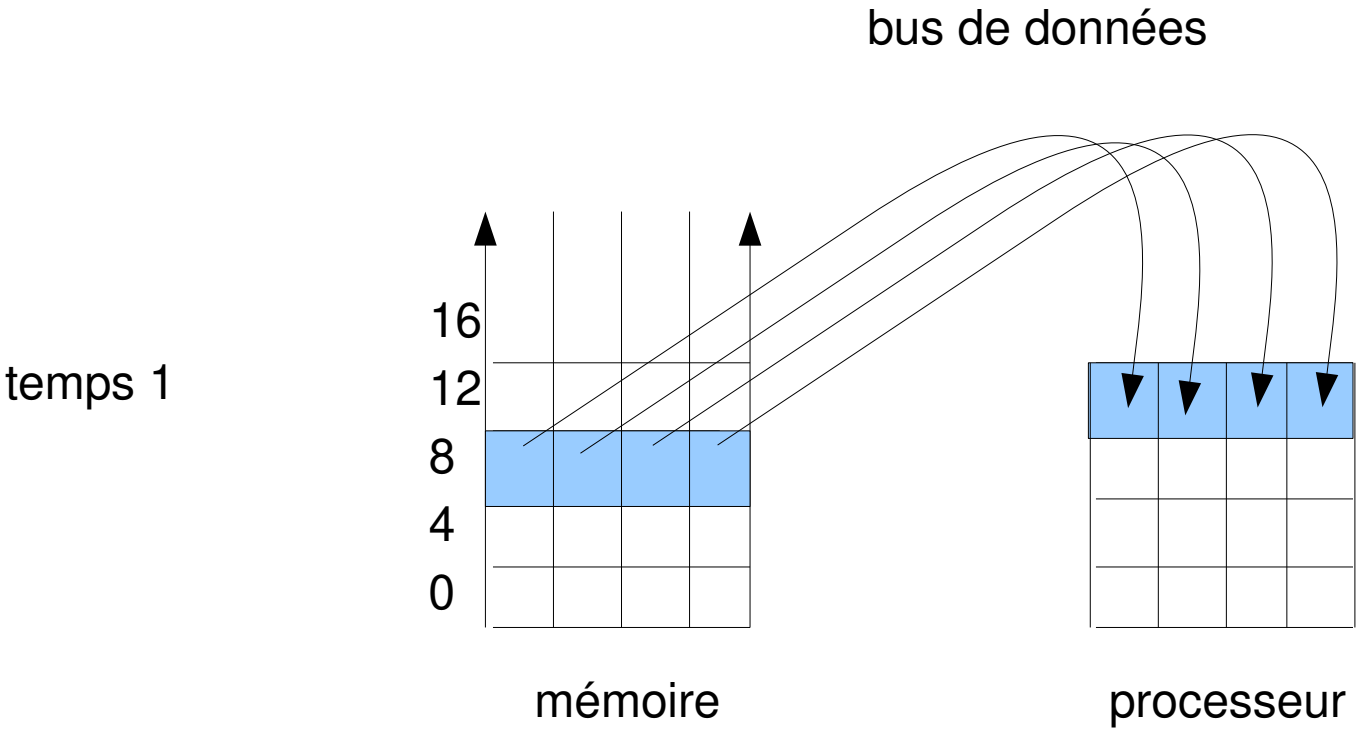
- Tous les nombres ne sont pas représentables (Ex:14.4)
- Pas associatif :

```
#include <stdio.h>

int main(void) { /* Hypothèse: calcul IEEE 754-1985, float=simple précision */
  float x=1.0/(2.0*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2); /* 2^-24 */
  printf("%.30f\n", (1.0+x)+x ); /* 1.000000000000000000000000000000 */
  printf("%.30f\n", 1.0+(x+x) ); /* 1.000000119209289550781250000000 */
  return 0;
}
```

Alignement

Le processeur n'accède qu'à un seul mot mémoire à un temps donné

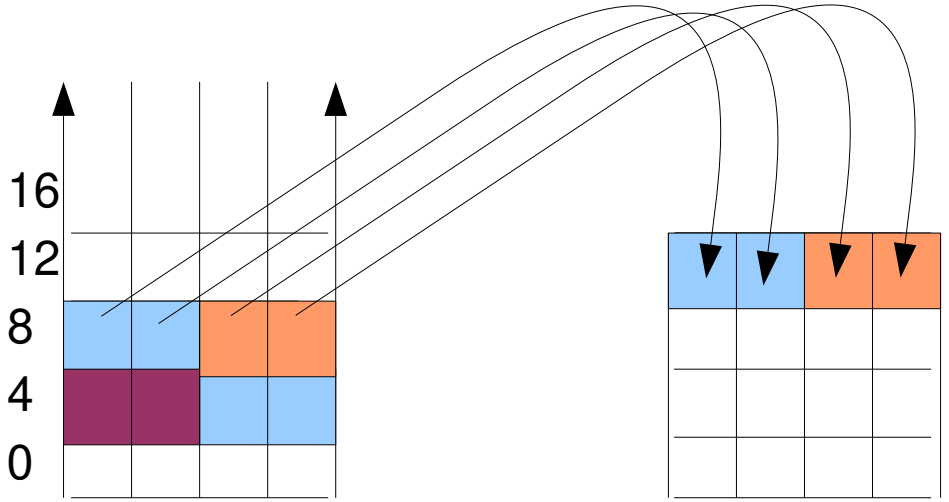
Il charge naturellement (et donc rapidement) les données dont l'adresse est un multiple de la taille d'un mot mémoire.



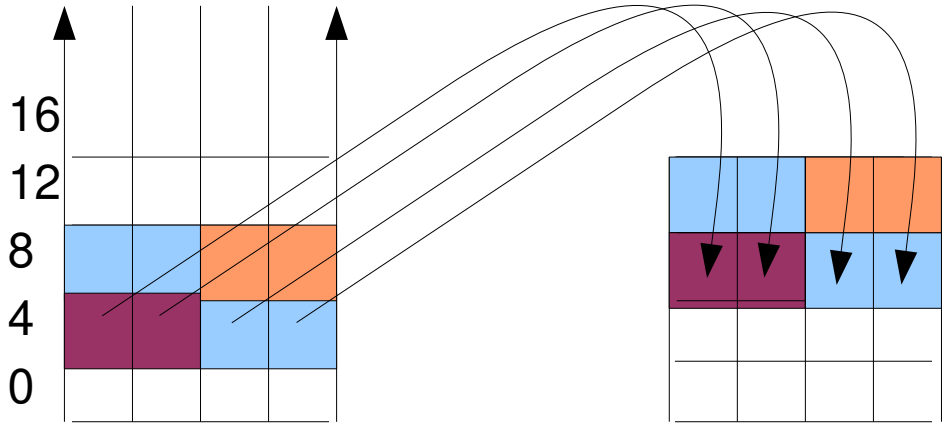
Alignement

Plusieurs accès mémoire peuvent être nécessaires pour transférer des données non alignées :

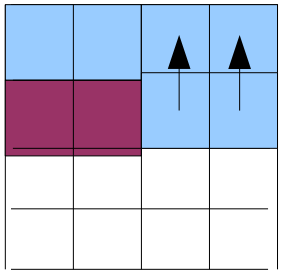
temps 1



temps 2



temps 3



processeur

De plus :
un défaut de page peut survenir si la donnée
à charger est à cheval sur deux pages mémoire

Alignement

Le compilateur peut arranger les données pour qu'elles soient alignées correctement :
- le programme s'exécutera plus rapidement
- les données prendront plus de place en mémoire

Il peut aussi ne pas le faire :
- le programme s'exécutera plus lentement (dans certains cas, très significativement)
- les données prendront moins de place mémoire (dans certains cas, très significativement)

Très souvent, on préfère aligner.

L'alignement peut avoir des conséquences sur les tailles des pointeurs.
- Exemple :
  - si les adresses des entiers sont des multiples de 4, les deux derniers bits sont inutiles
  - si les adresses des réels sont des multiples de 8, les trois derniers bits sont inutiles
  - les adresses « génériques » utilisent tous les bits.

Le compilateur peut décider de tailles différentes pour les pointeurs (ou pas)

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

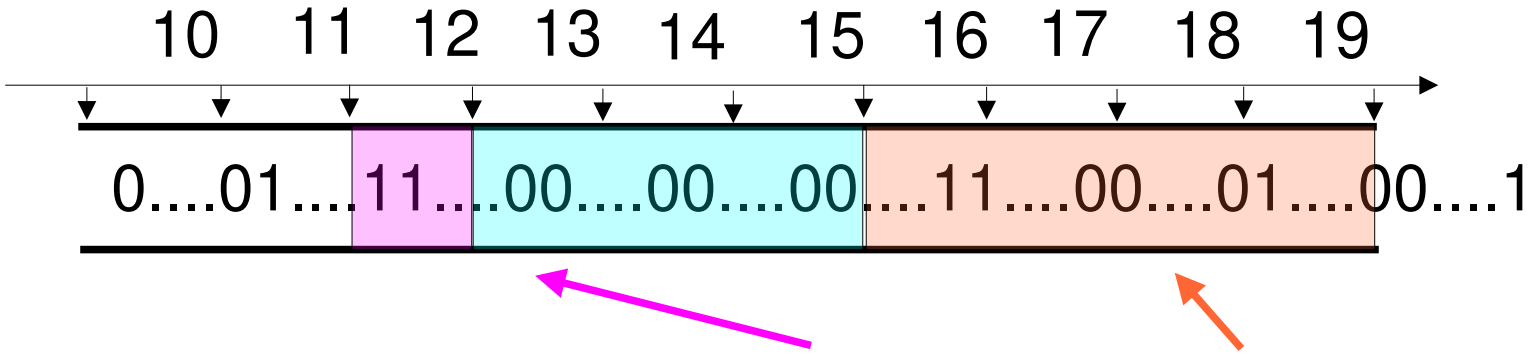
Si par ailleurs on souhaite que taille(T[n])=n*taille(T) alors il faut ajouter du *padding* en fin de type

$$sizeof\left(struct\ nom\right) \geqslant \sum_{i=1}^{i=n} sizeof\left(type_i\right)$$

Suivants leurs types, les adresses d'objets peuvent avoir des propriétés différentes (machine dépendantes). Sur chaque machine, il existe une classe d'adresses regroupant les contraintes de toutes les autres.

Exemples:

- Adresses de char: quelconques
- Adresses d'int: multiples (en cellules) de la taille d'un int
- Adresses de float: multiples (en cellules) de la taille d'un int
- .....



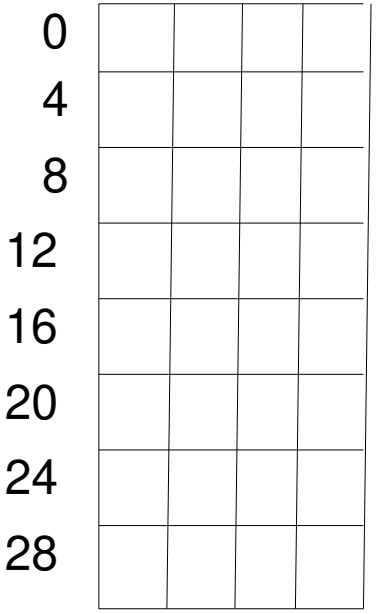
struct complexe { signed char reel; int imaginaire; };

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :

type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }

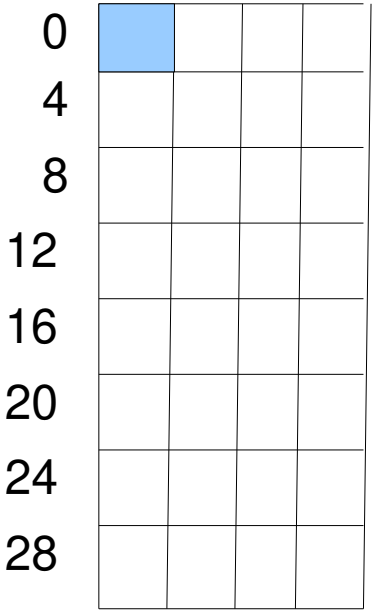


Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :

type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }

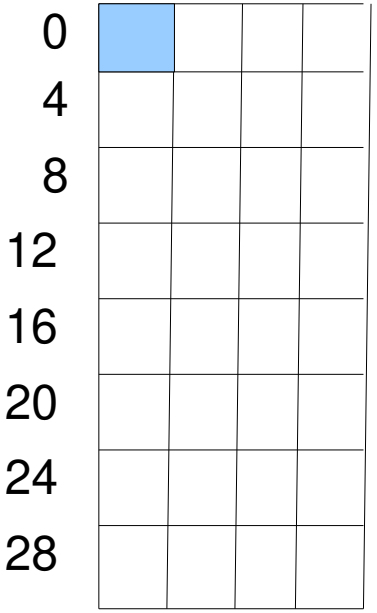


car a

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :
type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }



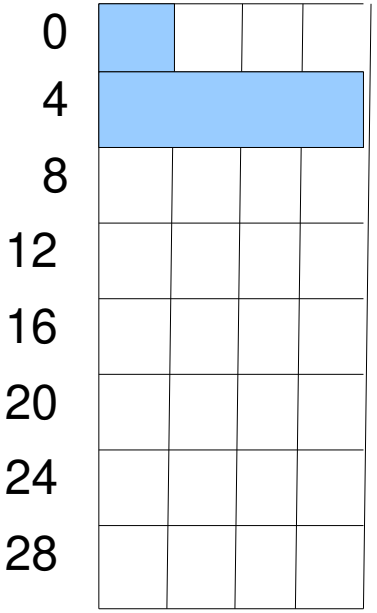
car a

placement de b
offset = 1
align = 4
padding = (4-(1 mod 4))mod 4 = 3

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :
type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }



car a

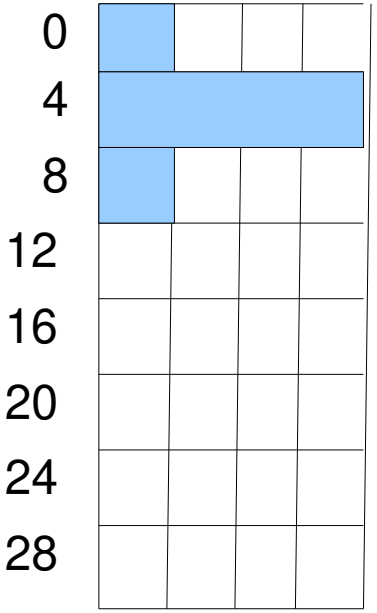
entier b

placement de b
offset = 1
align = 4
padding = (4-(1 mod 4))mod 4 = 3
new offset = 1+3

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :
type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }



car a

entier b

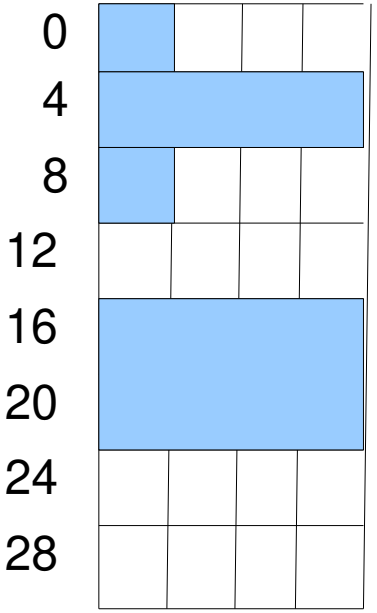
car c

placement de c
offset = 8
align = 1
padding = (1-(8 mod 1))mod 1 = 0
new offset = 8+0

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :
type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }



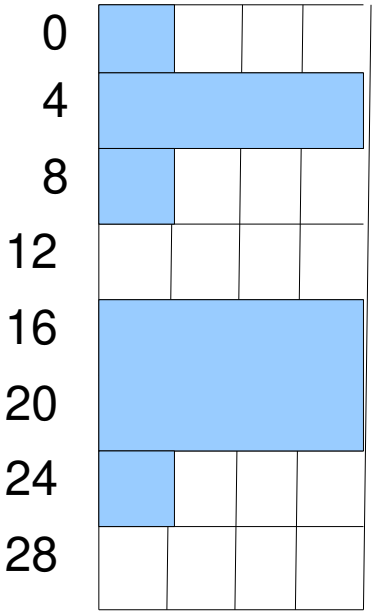
car a
entier b
car c
réel d

placement de d
offset = 9
align = 8
padding = (8-(9 mod 8))mod 8 = 7
new offset = 9+7=16

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :
type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }



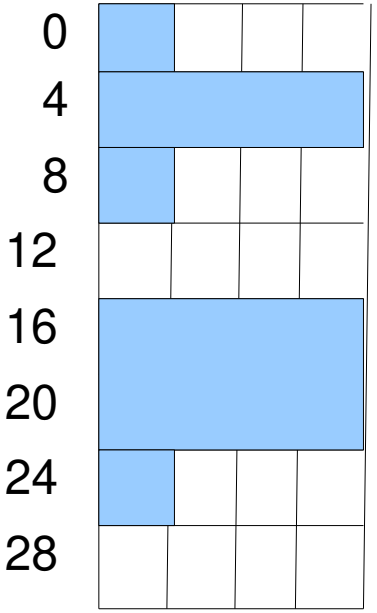
placement de e
offset = 24
align = 1
padding = (1-(24 mod 1))mod 1 = 0
new offset = 24+0

Alignement dans les structures
Taille des structures

padding = (align - (offset mod align)) mod align
new offset = offset + padding = offset + (align - (offset mod align)) mod align

Exemple :

type car : taille 1, pas de contrainte d'alignement
tpe entier : taille 4, contrainte d'alignement : multiple de 4
type reel : taille 8, contrainte d'alignement : multiple de 8
struct { car a ; entier b ; car c ; reel d ; car e ; }



Si on veut garantir que
sizeof(struct[n])=n*sizeof(struct)
align = maximum des alignements des membres = 8
offset = 25
padding = (8-(25 mod 8)) mod 8 = 7

sizeof(struct)=25+7=32

Taille des unions

Alignement : tous les membres sont alignés par rapport au début
les espaces mémoire se recouvrent

Contrainte d'alignement = max des contraintes d'alignement des membres

Pas de *padding* entre les membres

Taille = max des tailles des membres

Exemple : union { car a ; entier b ; car c ; réel d ; car e ; }
Contrainte d'alignement : celle des réels = 8
Taille : celle des réels = 8

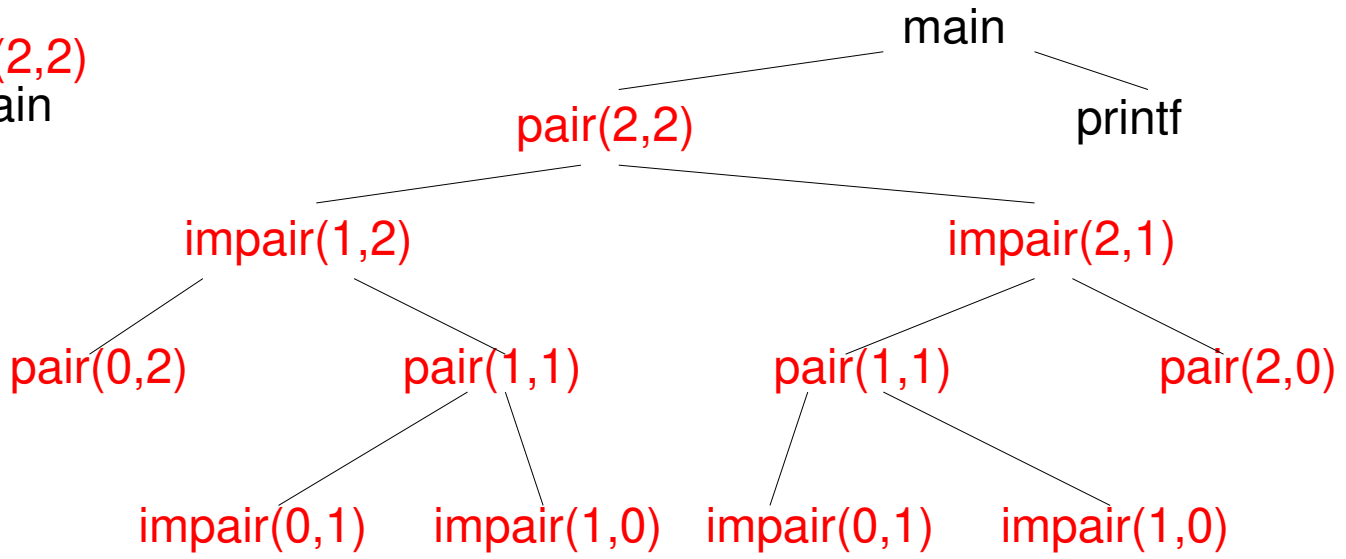
Appels de fonctions : activation

Activation d'un appel : entrée dans la fonction -> sortie de la fonction

```
unsigned pair(unsigned a, unsigned b) {
  if ( a == 0 )
    return b;
  if ( b == 0 )
    return a;
  return impair(a-1, b) + impair(a, b-1);
}
```

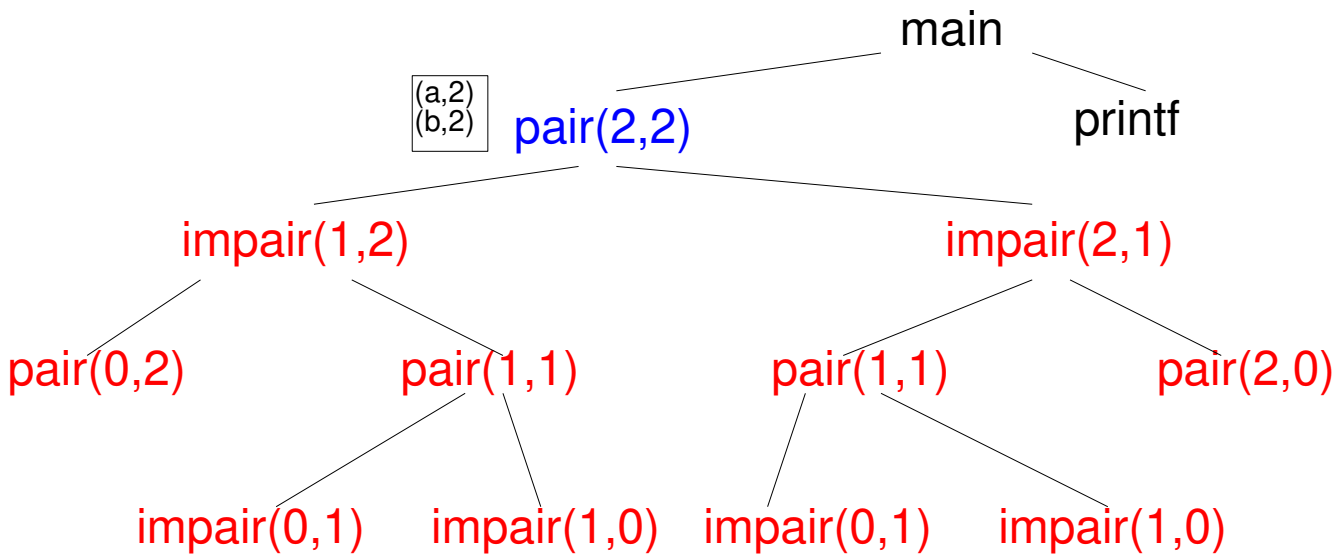
```
unsigned impair(unsigned a, unsigned b) {
  if ( a == 0 )
    return b;
  if ( b == 0 )
    return a;
  return pair(a-1, b) + pair(a, b-1);
}
```

Activation de pair(2,2)
à partir de main



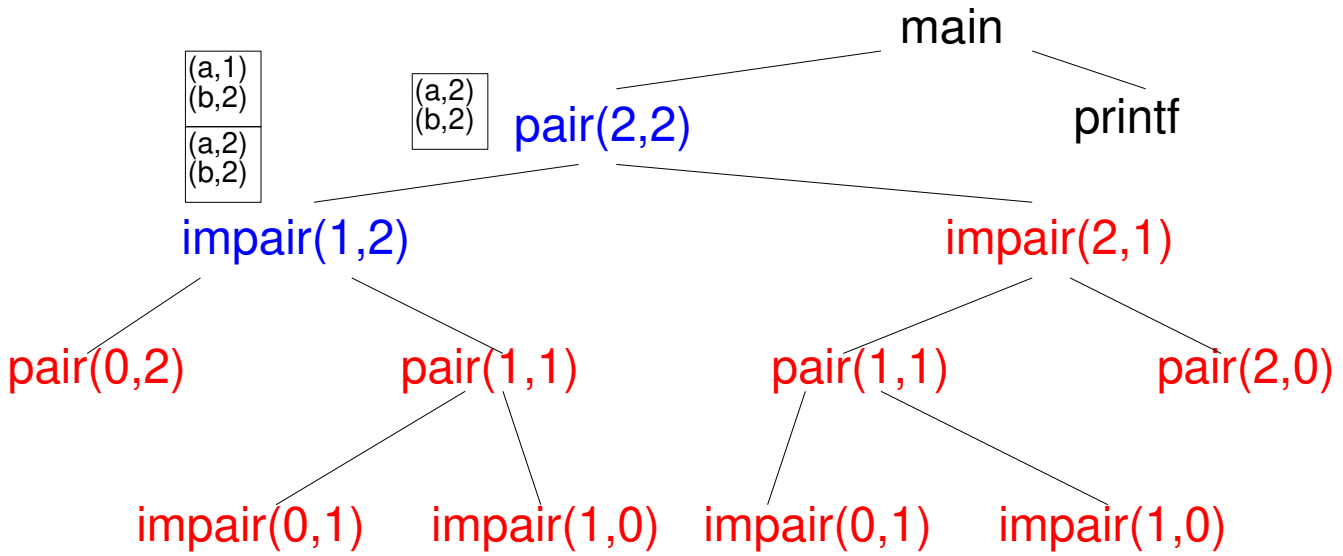
Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



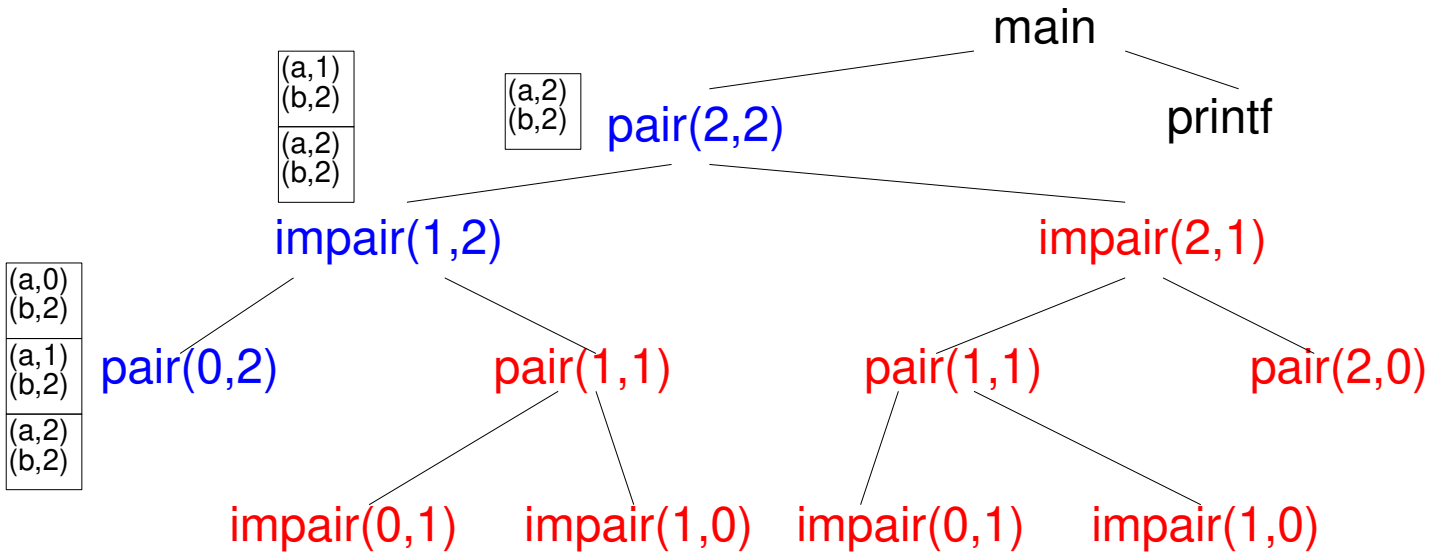
Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



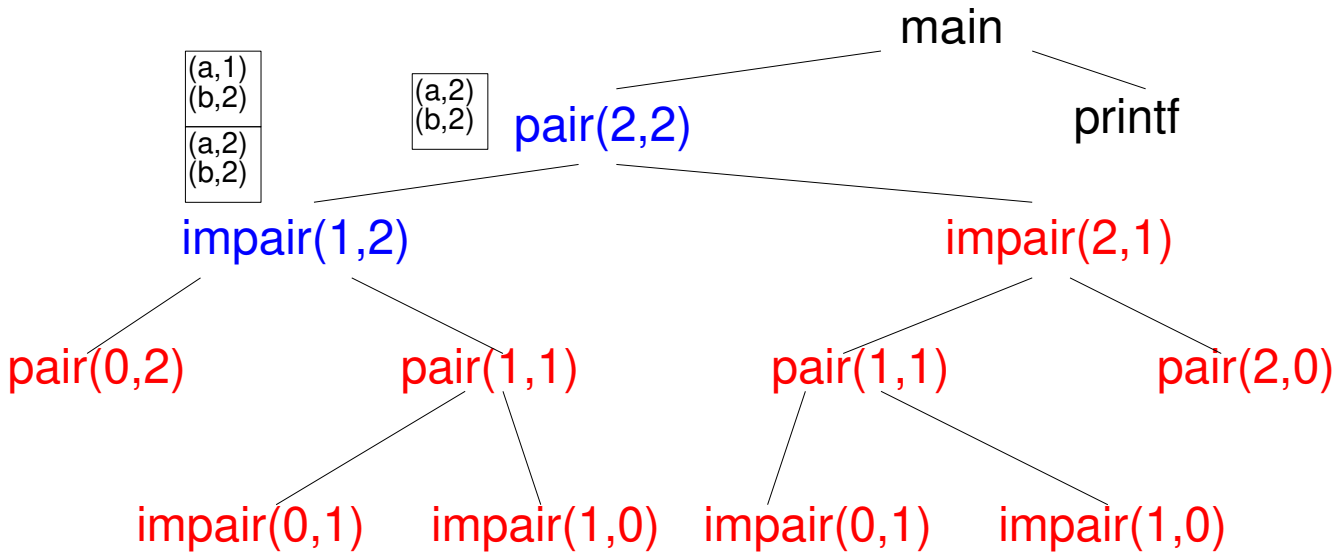
Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



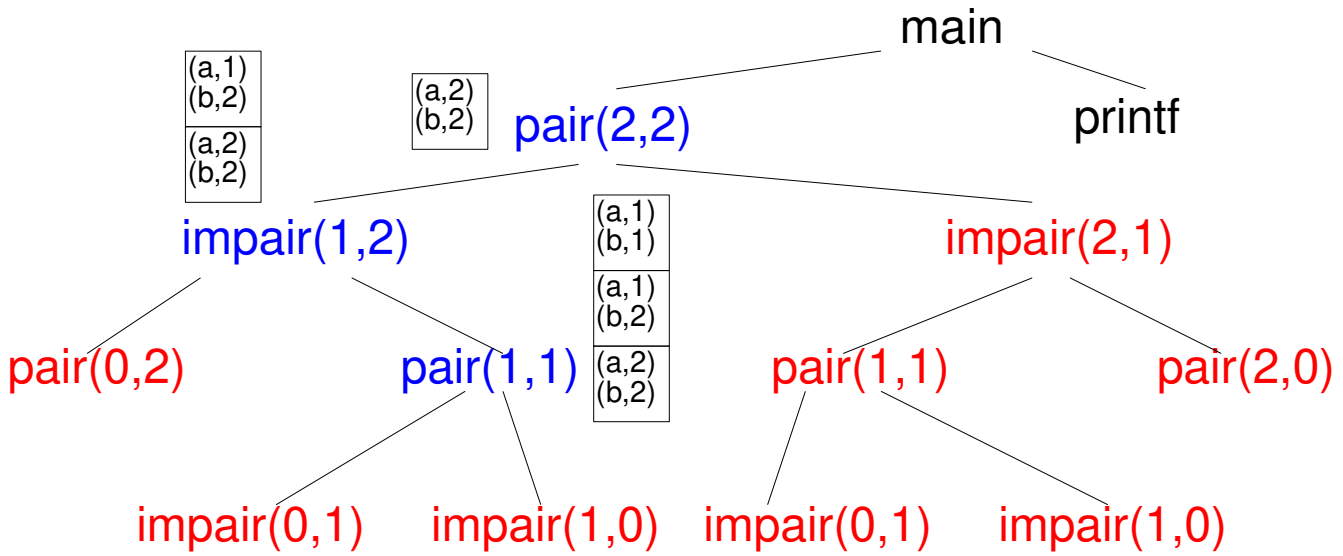
Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



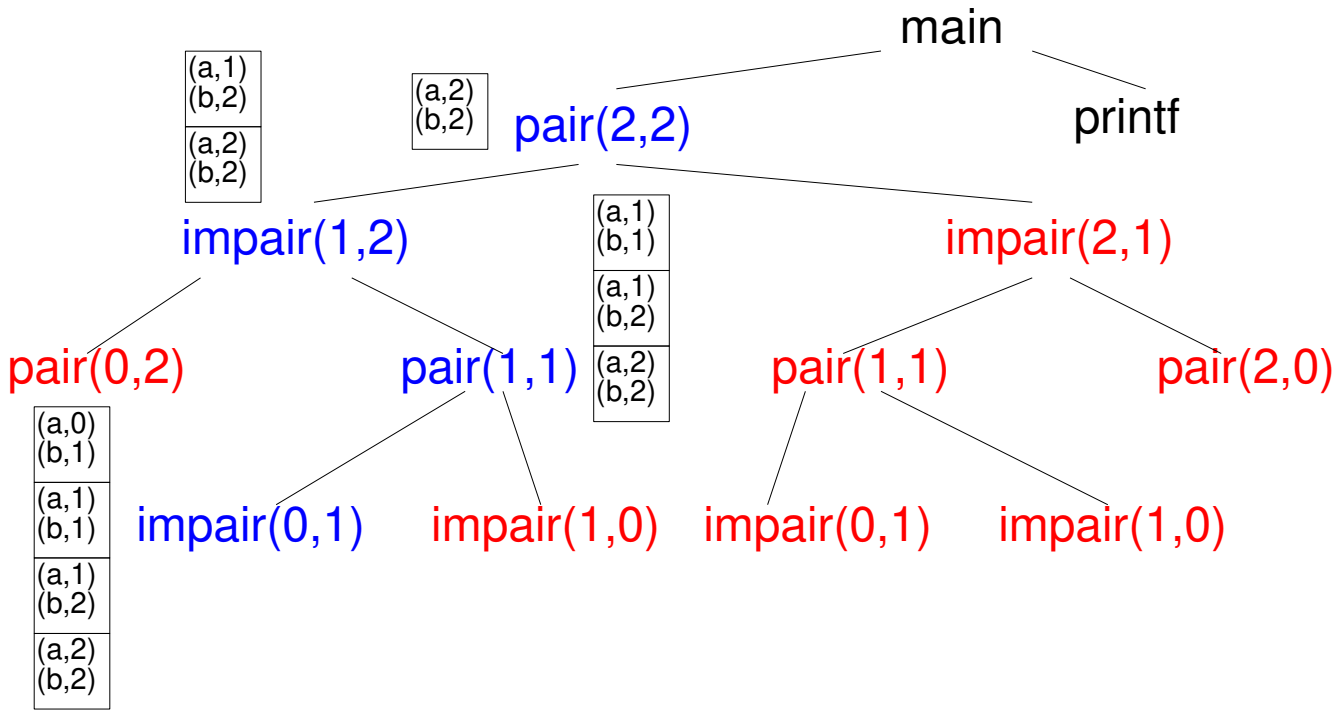
Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



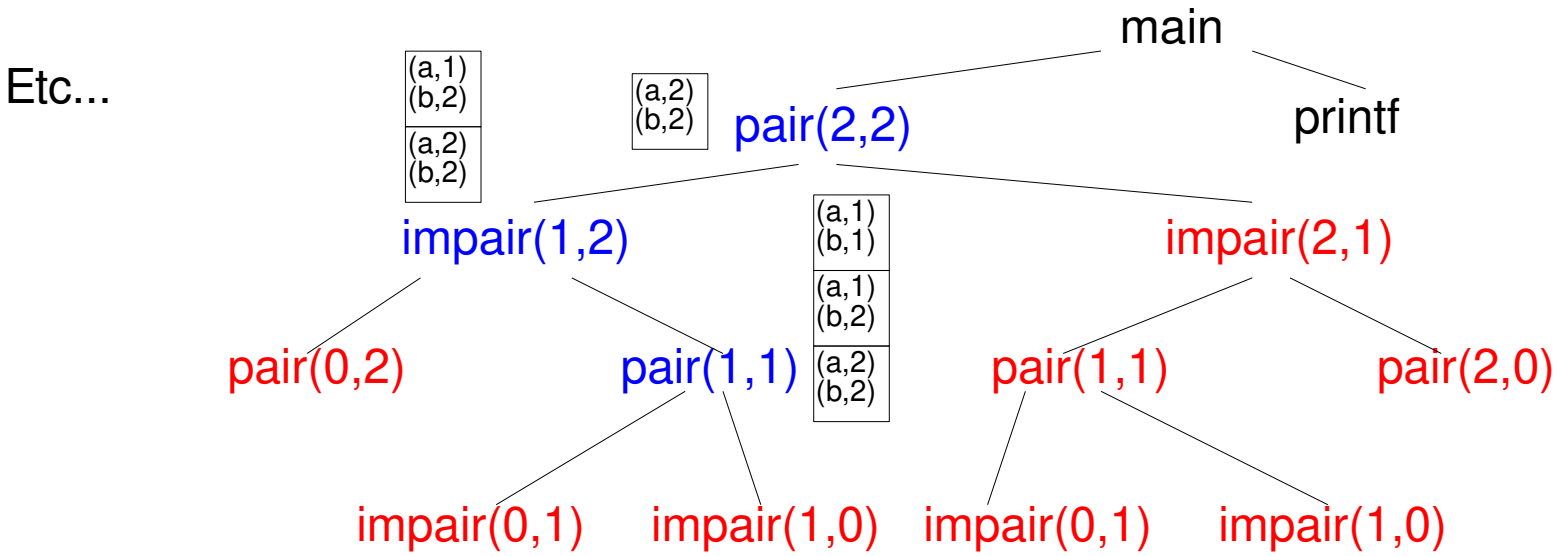
Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



Appels de fonctions : activation

Sauvegarder/restaurer les valeurs des paramètres, variables locales et autres informations sur l'exécution des fonctions



Les fonctions actives et leurs données sont conservées dans la pile.
La liste des fonctions actives est le chemin de la racine vers la dernière fonction activée.
La gestion en mémoire de la pile de contrôle permet d'implanter les langages procéduraux.

La solution Java

Chaque processus mémorise les variables locales et les paramètres des méthodes en cours d'exécution dans pile de *stack-frames* (cadres d'exécution)

Une *stack-frame* pour la méthode *m* contient en particulier

- l'espace nécessaire pour stocker chaque paramètre et chaque variable locale de *m*.
- une pile dont la taille maximale est calculée par le compilateur

Les instructions *invokeX* qui permettent d'appeler une méthode

- réalisent la résolution de la méthode *m* à appeler
- créent une stack-frame *s* pour préparer l'appel à *m* et placent *s* dans la pile de *stack-frames*
- recopient les valeurs des paramètres dans les espaces de *s* qui leur sont réservés
- passent le contrôle d'exécution à *m*

En fin d'exécution de *m*

- le sommet de la pile de la *stack-frame s* contient la valeur de retour *r*
- elle est mémorisée par la VM
- la VM détruit *s*
- la VM mets *r* en sommet de la pile d'exécution de la stack-frame de la fonction appelante
- le flot d'exécution de la fonction appelante reprends après *invokeX*

La solution Java

Dans la stack-frame, les variables locales et paramètres de *m* sont numérotés

Le *byte-code* permet de manipuler les variables locales et les paramètres uniquement par leur numéro

- Avantages :
  - la taille des *stack-frames* est constante, et calculée par le compilateur
  - comme les variables/paramètres ne sont accessibles que par leur numéro, pas de problème de sécurité

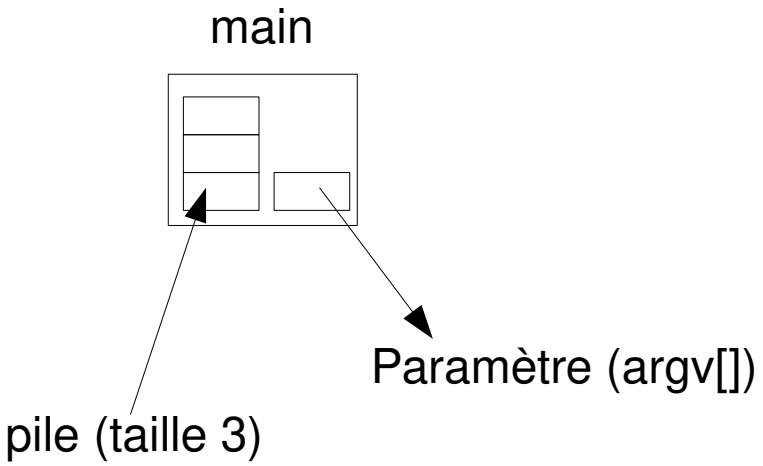
```
static int pair(int a, int b) {
  if ( a == 0 )
    return b;
  if ( b == 0 )
    return a;
  return impair(a-1, b) + impair(a, b-1);
}
```

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2                  // Method impair:(II)I
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2                  // Method impair:(II)I
      26: iadd
      27: ireturn
```

La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

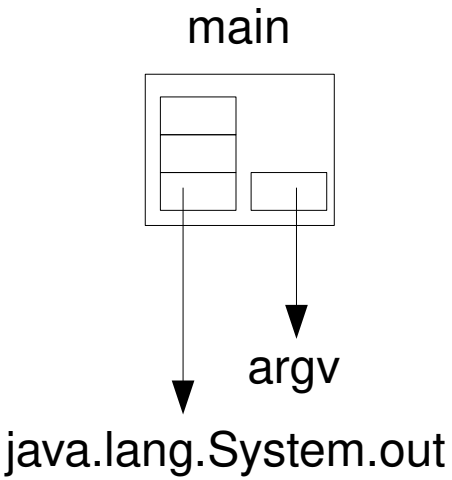
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

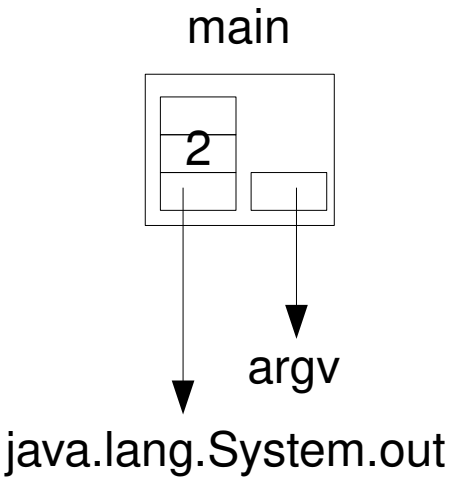
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

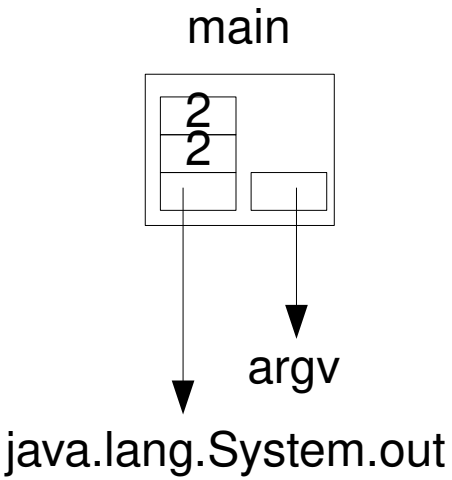
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

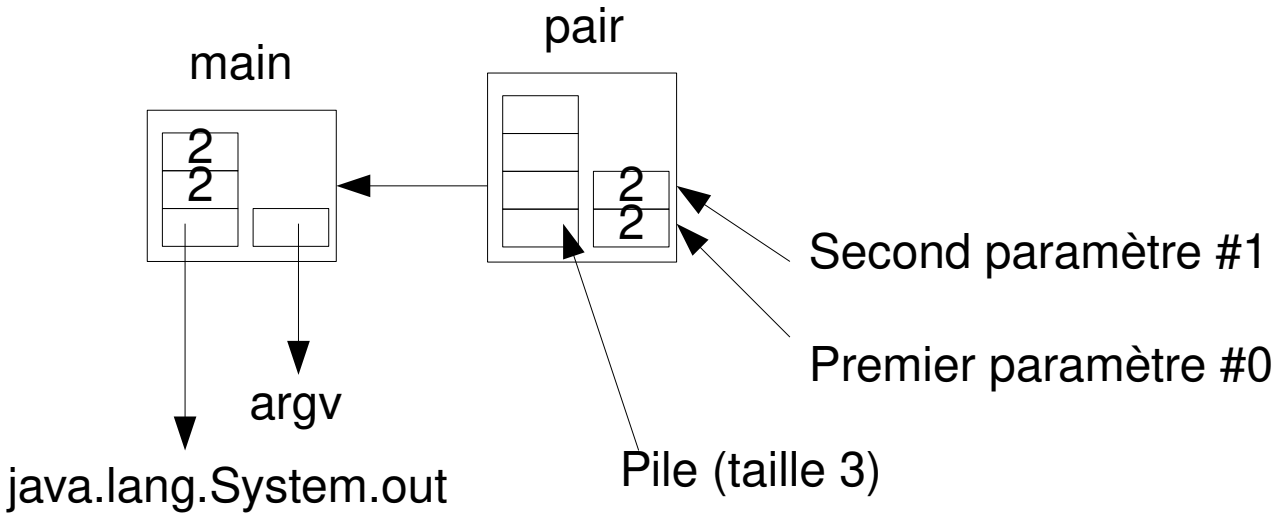
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

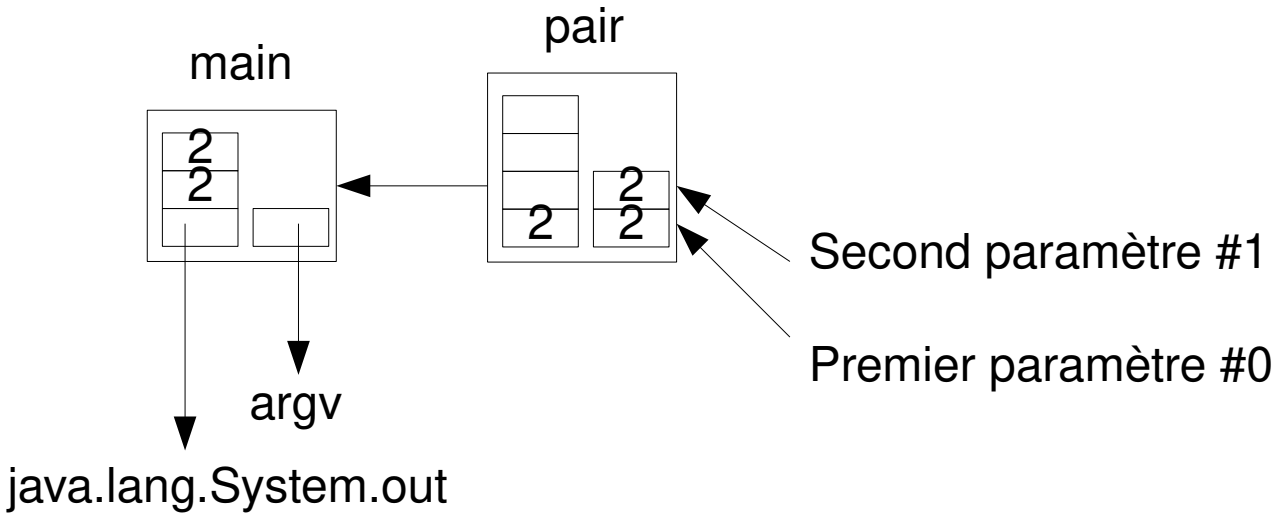
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
    0: iload_0
    1: ifne          6
    4: iload_1
    5: ireturn
    6: iload_1
    7: ifne          12
   10: iload_0
   11: ireturn
   12: iload_0
   13: iconst_1
   14: isub
   15: iload_1
   16: invokestatic  #2
   19: iload_0
   20: iload_1
   21: iconst_1
   22: isub
   23: invokestatic  #2
   26: iadd
   27: ireturn
```

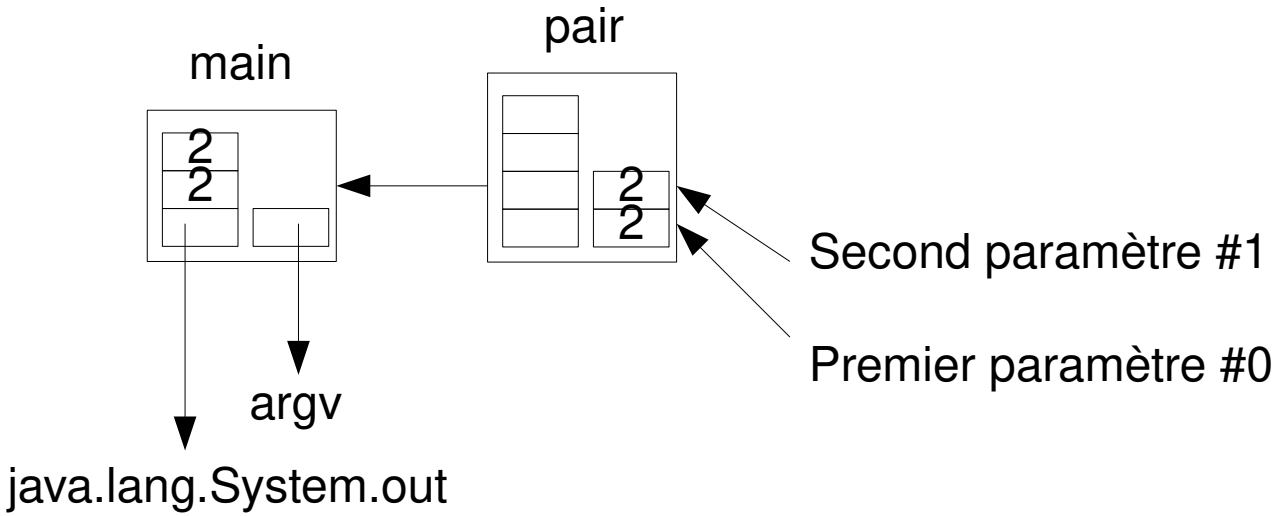
```
public static void main(java.lang.String[]);
  Code:
    0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
    3: iconst_2
    4: iconst_2
    5: invokestatic  #3                  // Method pair:(II)I
    8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
   11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

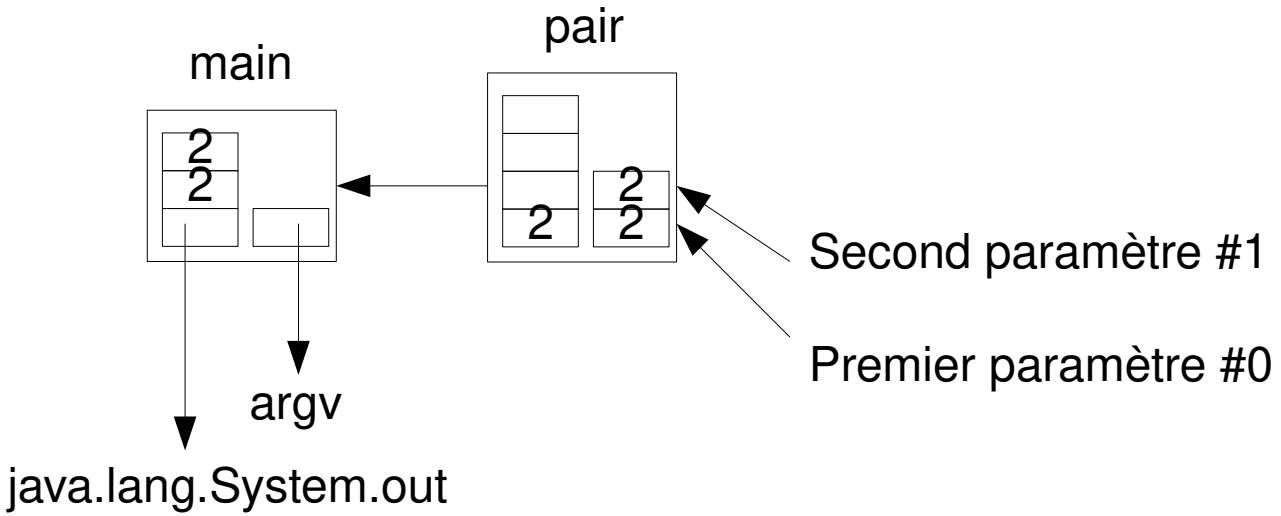
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

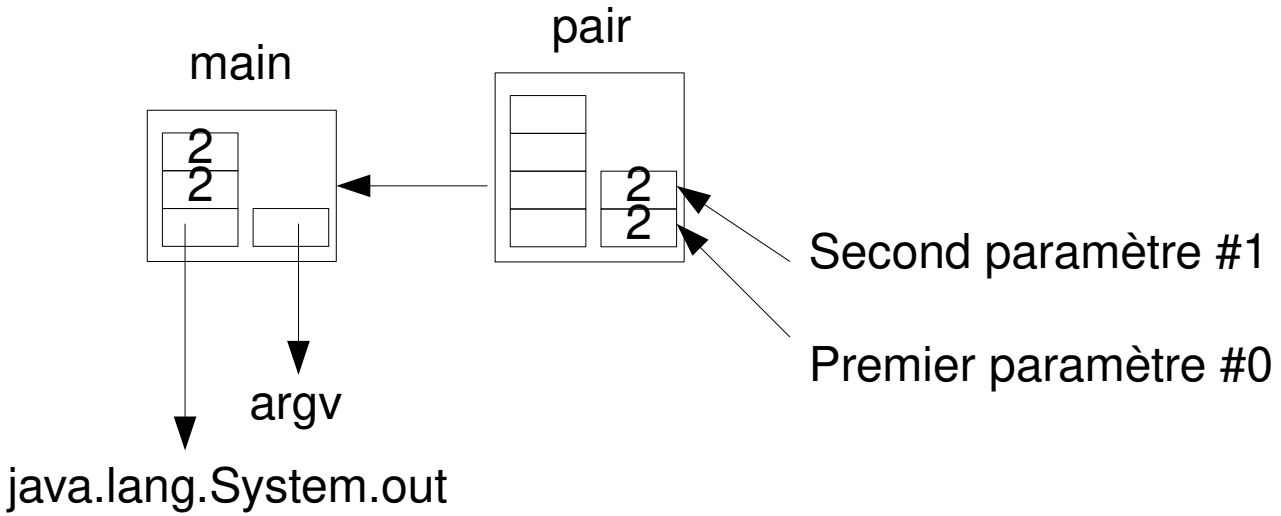
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

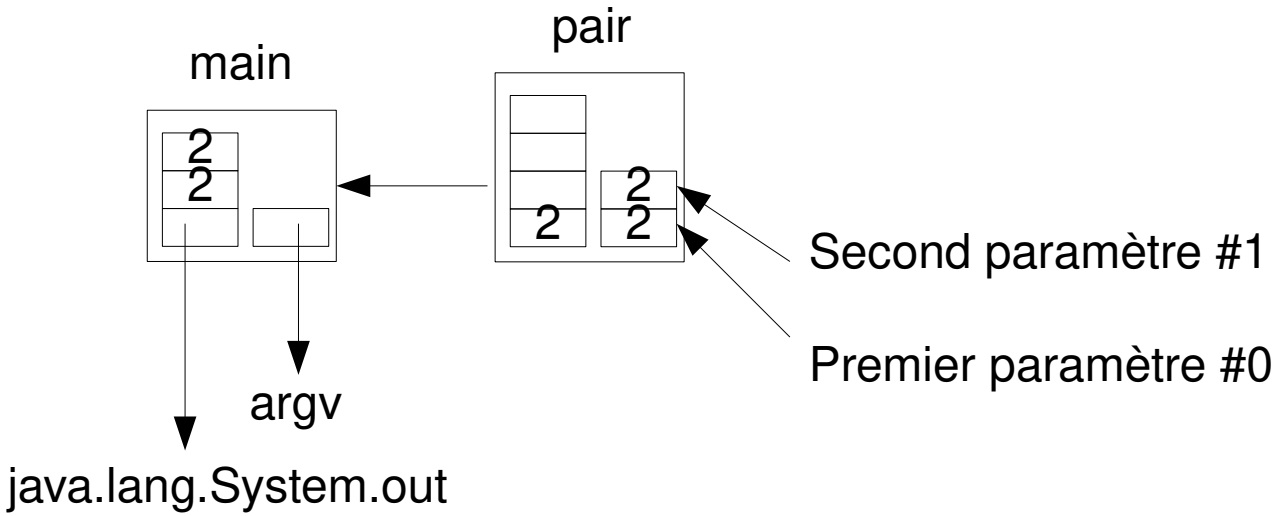
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

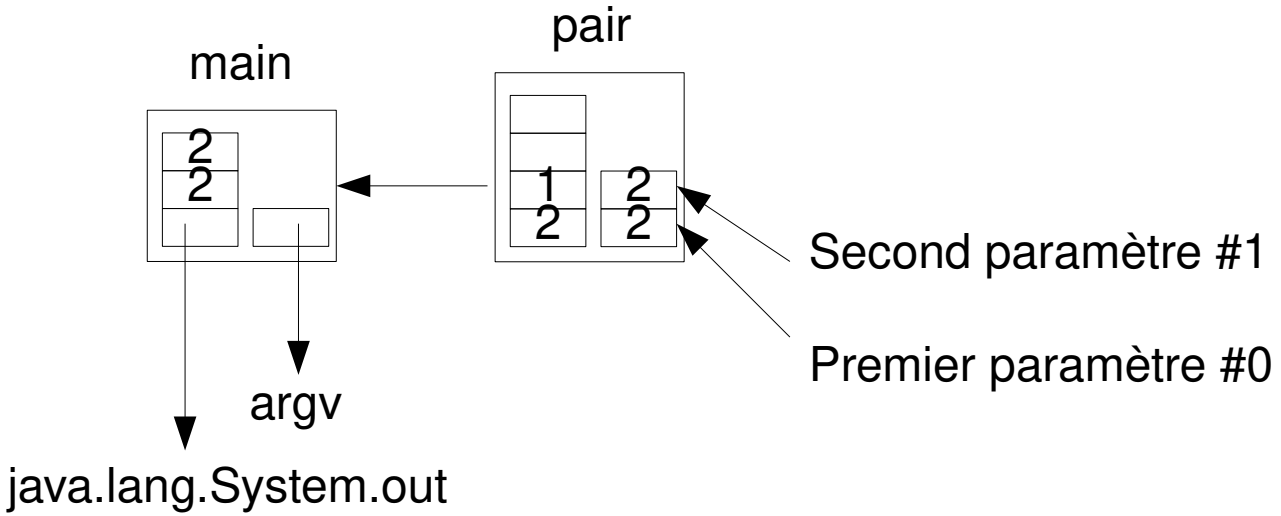
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

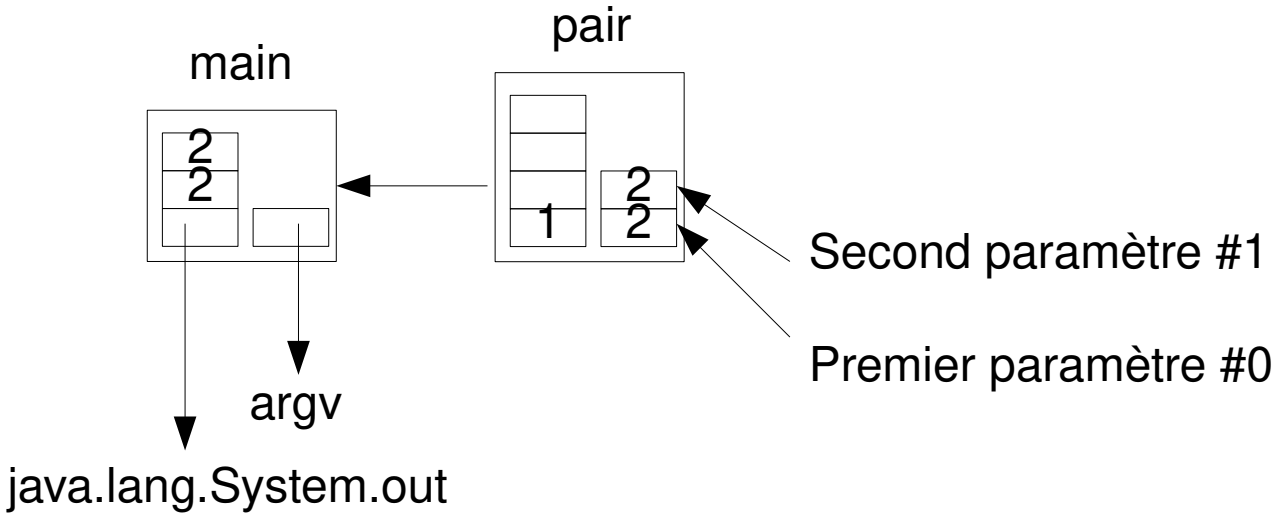
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

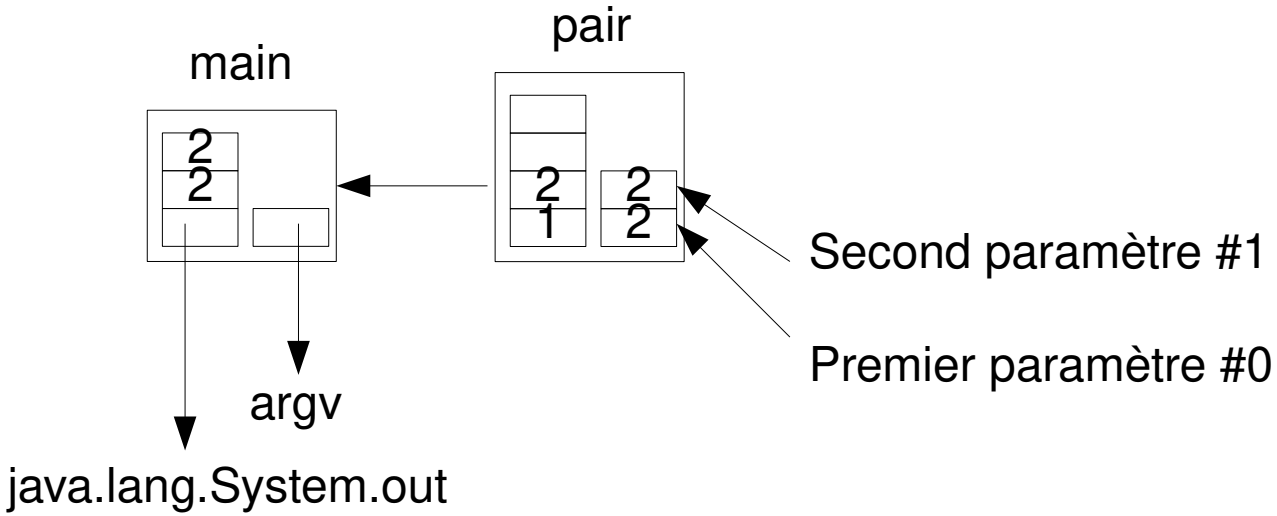
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

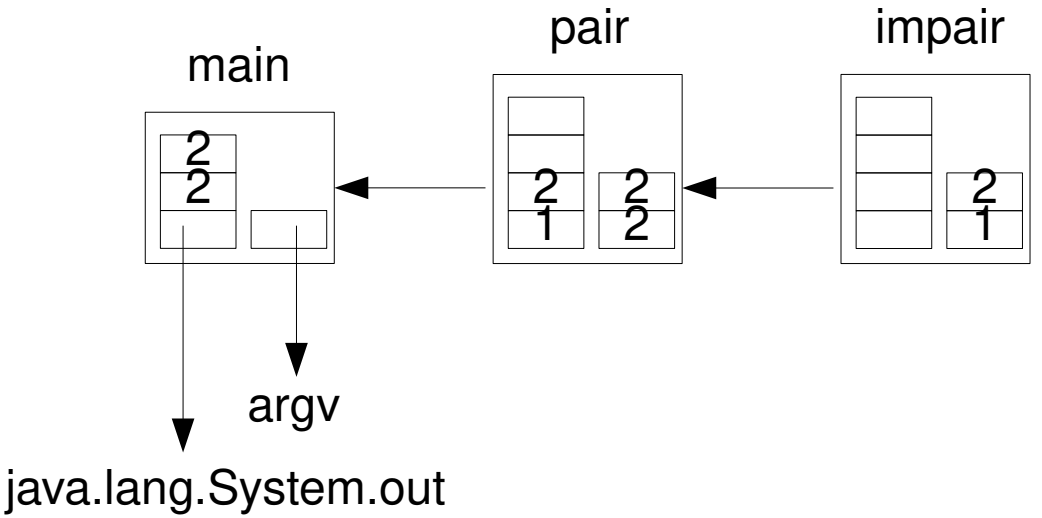
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

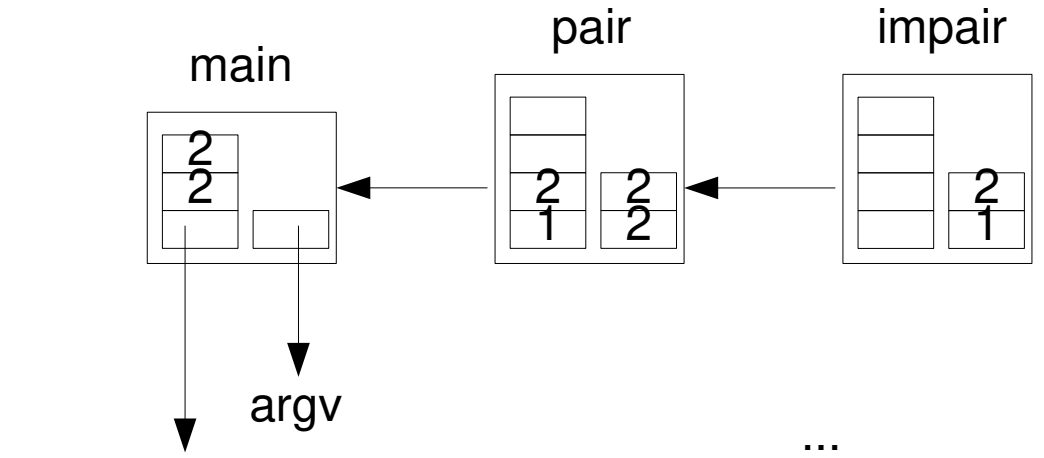
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```

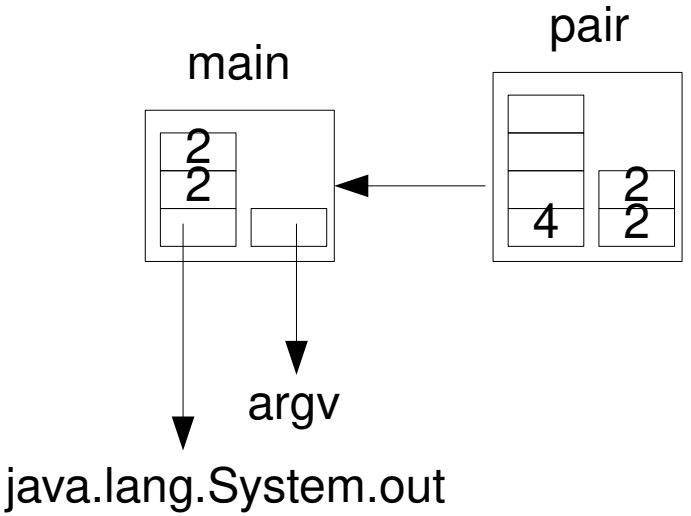


...
impair(1,2) s'exécute et termine
en retournant 4
...

La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

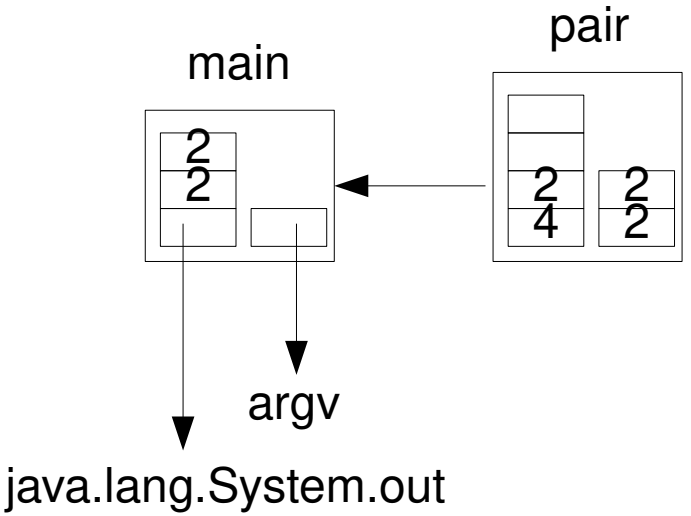
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

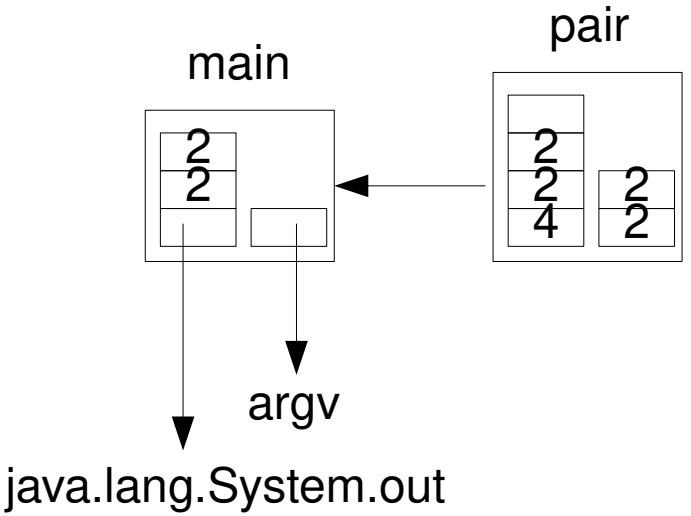
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

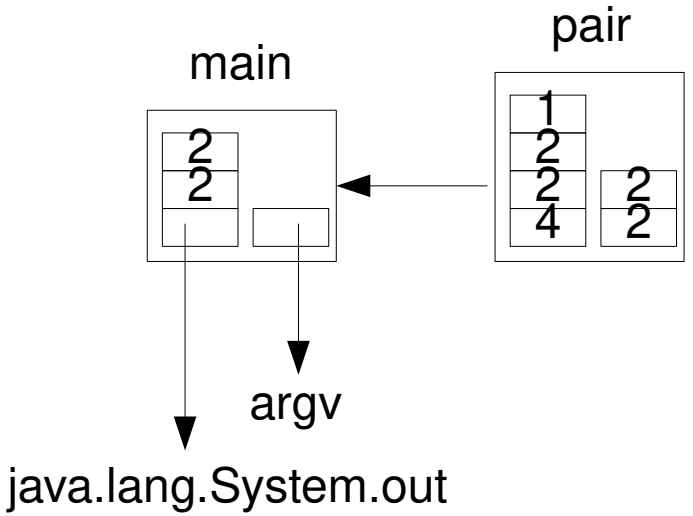
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

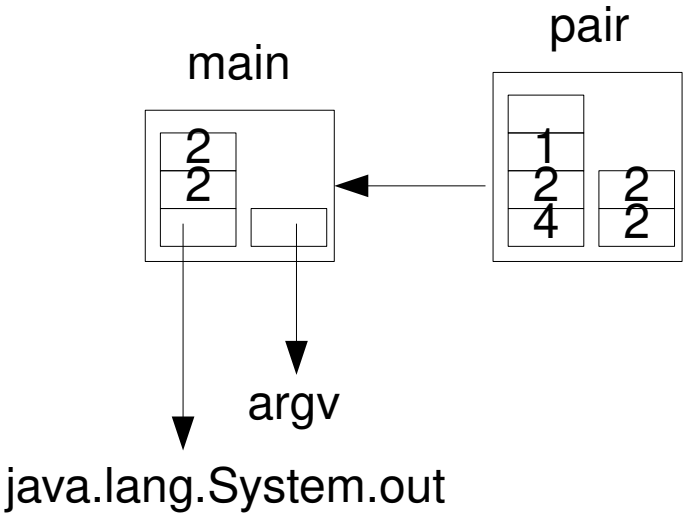
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

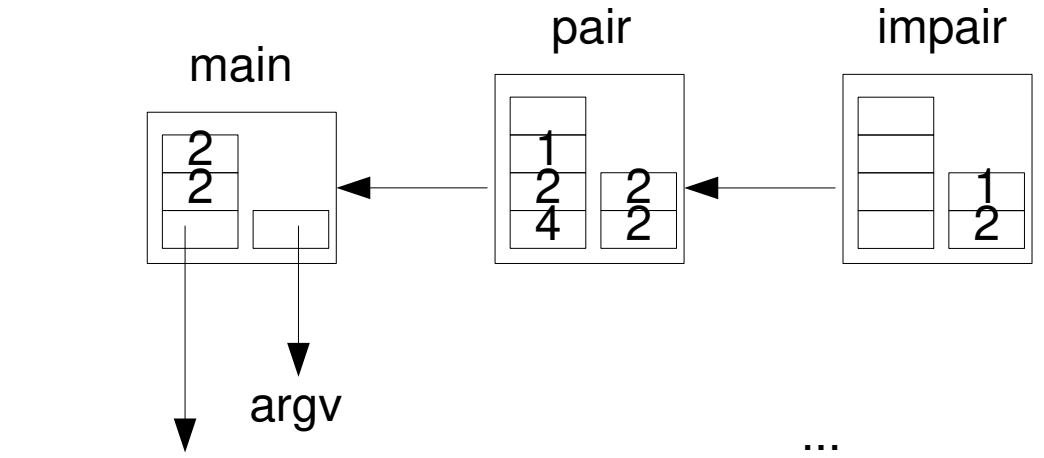
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```

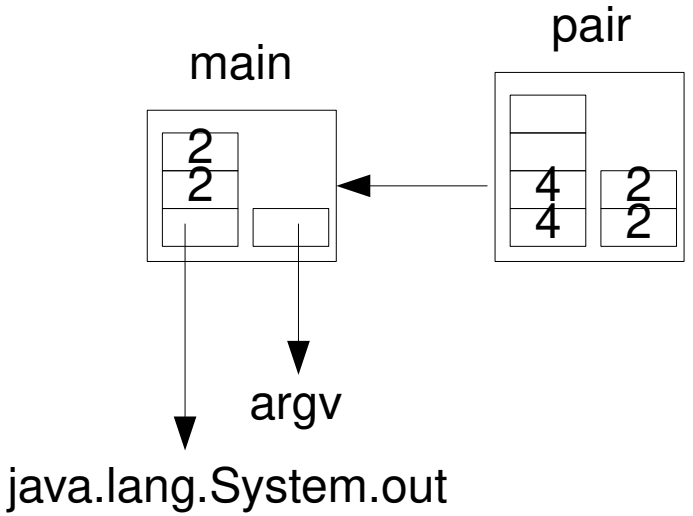


...
impair(2,1) s'exécute et termine
en retournant 4
...

La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

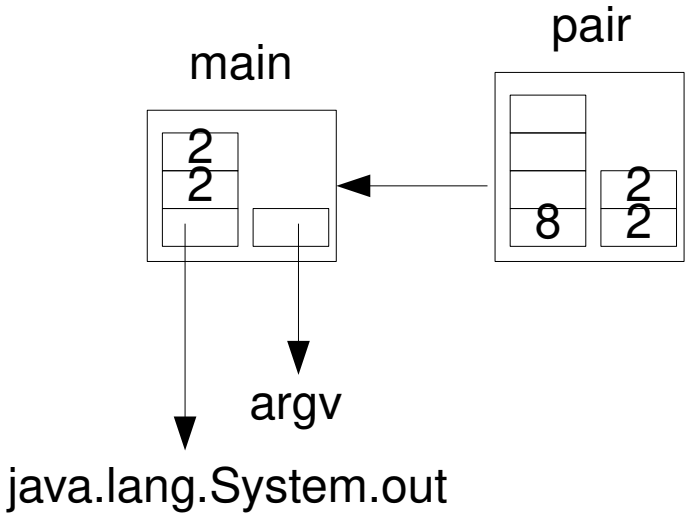
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

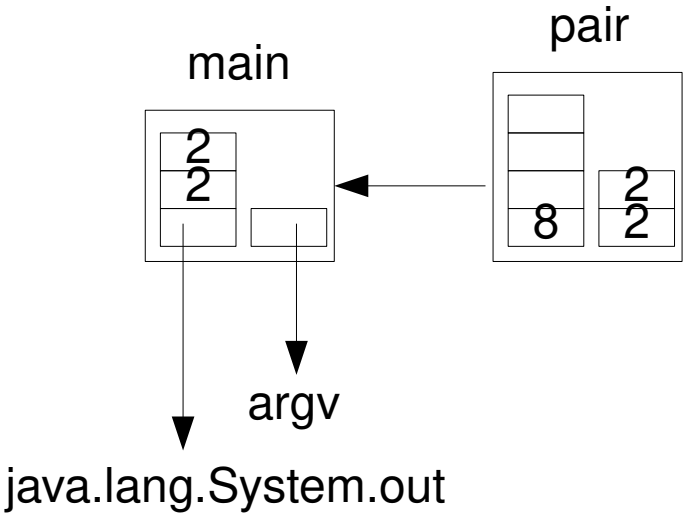
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

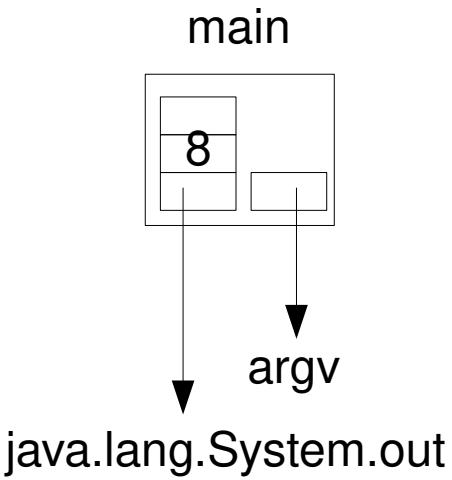
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

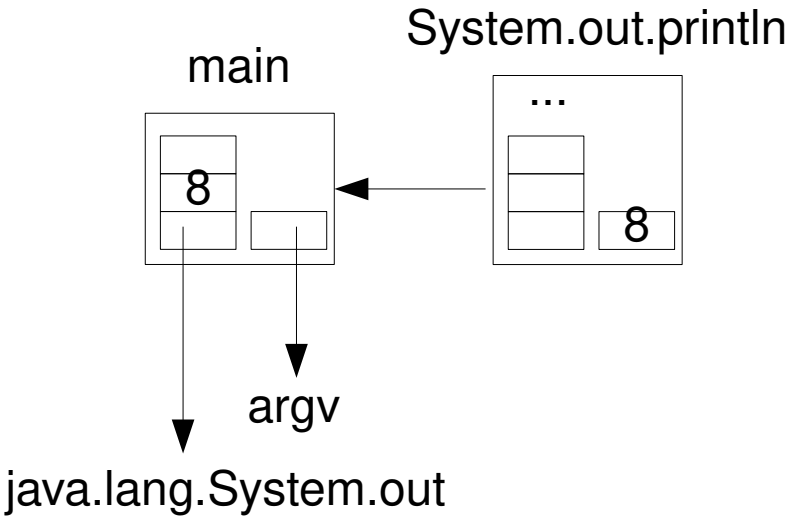
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

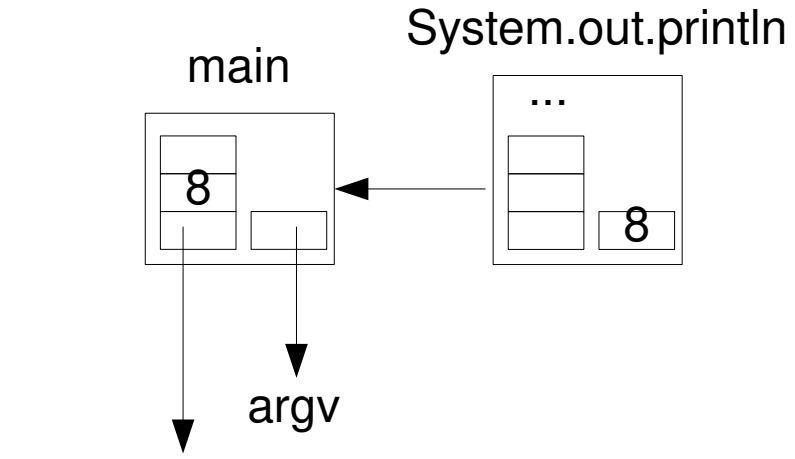
```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```



La solution Java

```
static int pair(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iconst_1
    14: isub
    15: iload_1
    16: invokestatic  #2
    19: iload_0
    20: iload_1
    21: iconst_1
    22: isub
    23: invokestatic  #2
    26: iadd
    27: ireturn
```

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: iconst_2
     4: iconst_2
     5: invokestatic  #3                  // Method pair:(II)I
     8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
    11: return
```

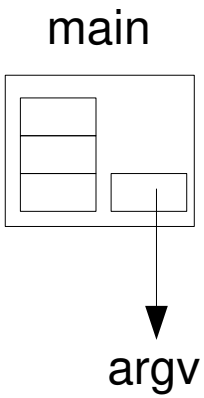


...
System.out.println(8) s'exécute
et termine
...

La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

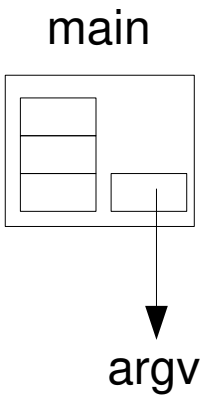
```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



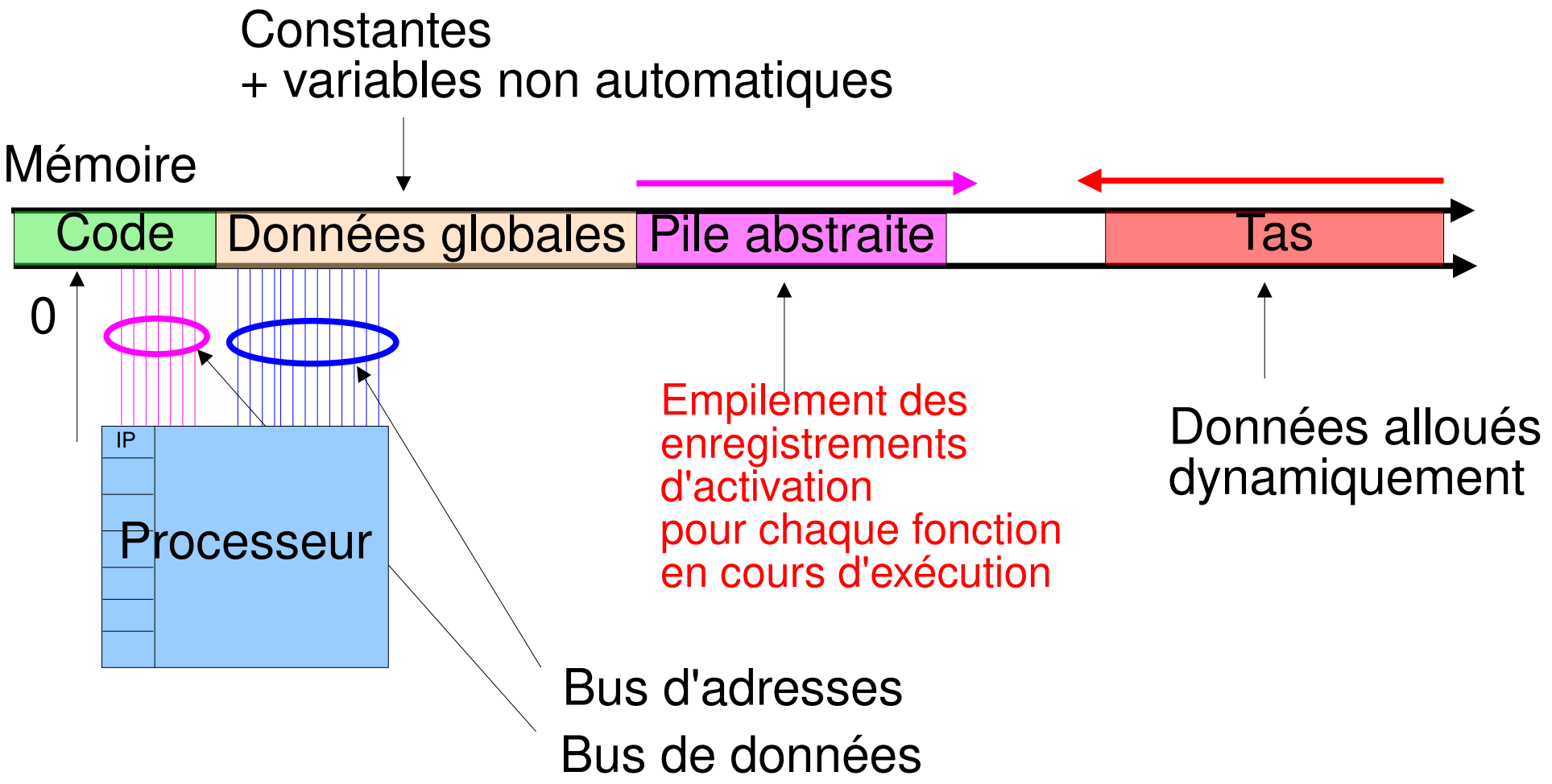
La solution Java

```
static int pair(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_1
       7: ifne          12
      10: iload_0
      11: ireturn
      12: iload_0
      13: iconst_1
      14: isub
      15: iload_1
      16: invokestatic  #2
      19: iload_0
      20: iload_1
      21: iconst_1
      22: isub
      23: invokestatic  #2
      26: iadd
      27: ireturn
```

```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iconst_2
       4: iconst_2
       5: invokestatic  #3                  // Method pair:(II)I
       8: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      11: return
```



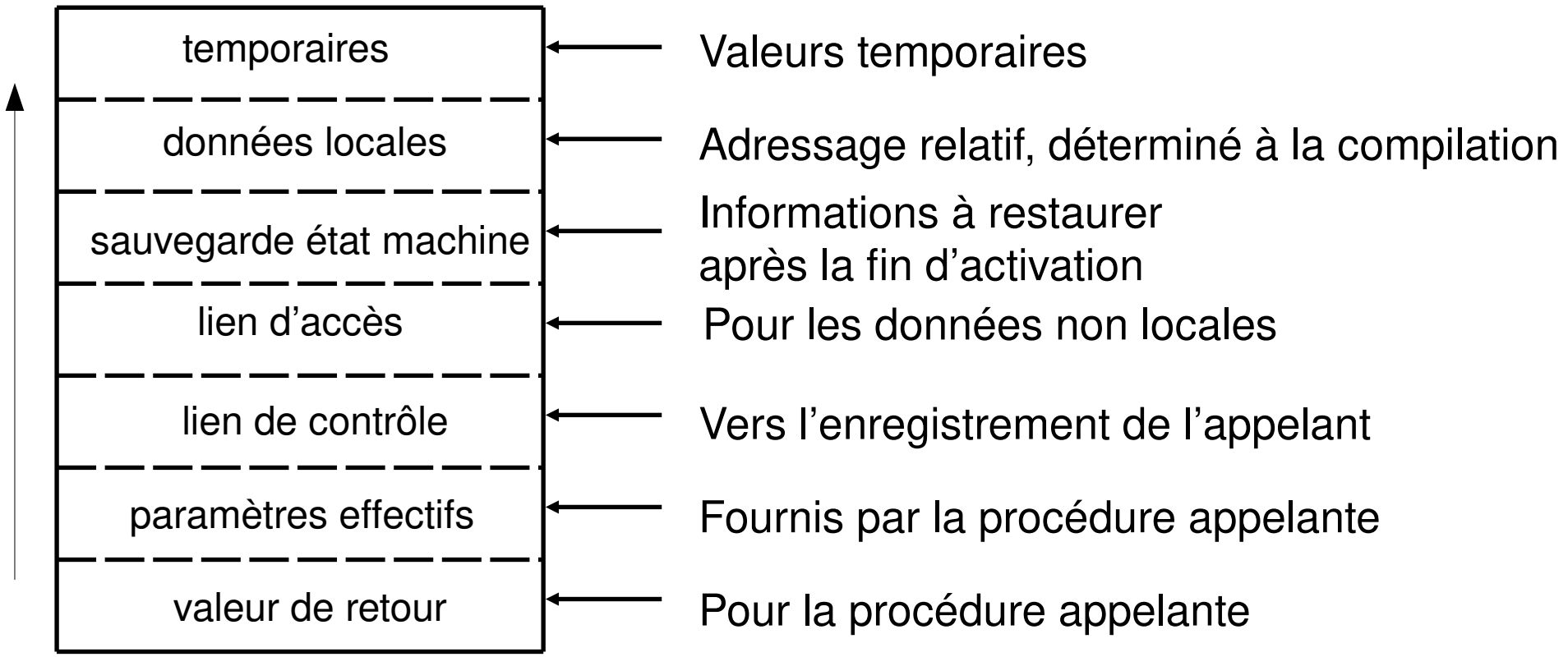
Solution pour une architecture classique



Registres (chaque registre contient un ***mot machine***)

Solution pour une architecture classique

Exemple d'enregistrement d'activation



Les tailles sont fixes, connues à la compilation, sauf pour des tableaux dont la taille dépend d'un argument.

Solution pour une architecture classique

Stratégies d'allocation des enregistrements d'activation :allocation statique

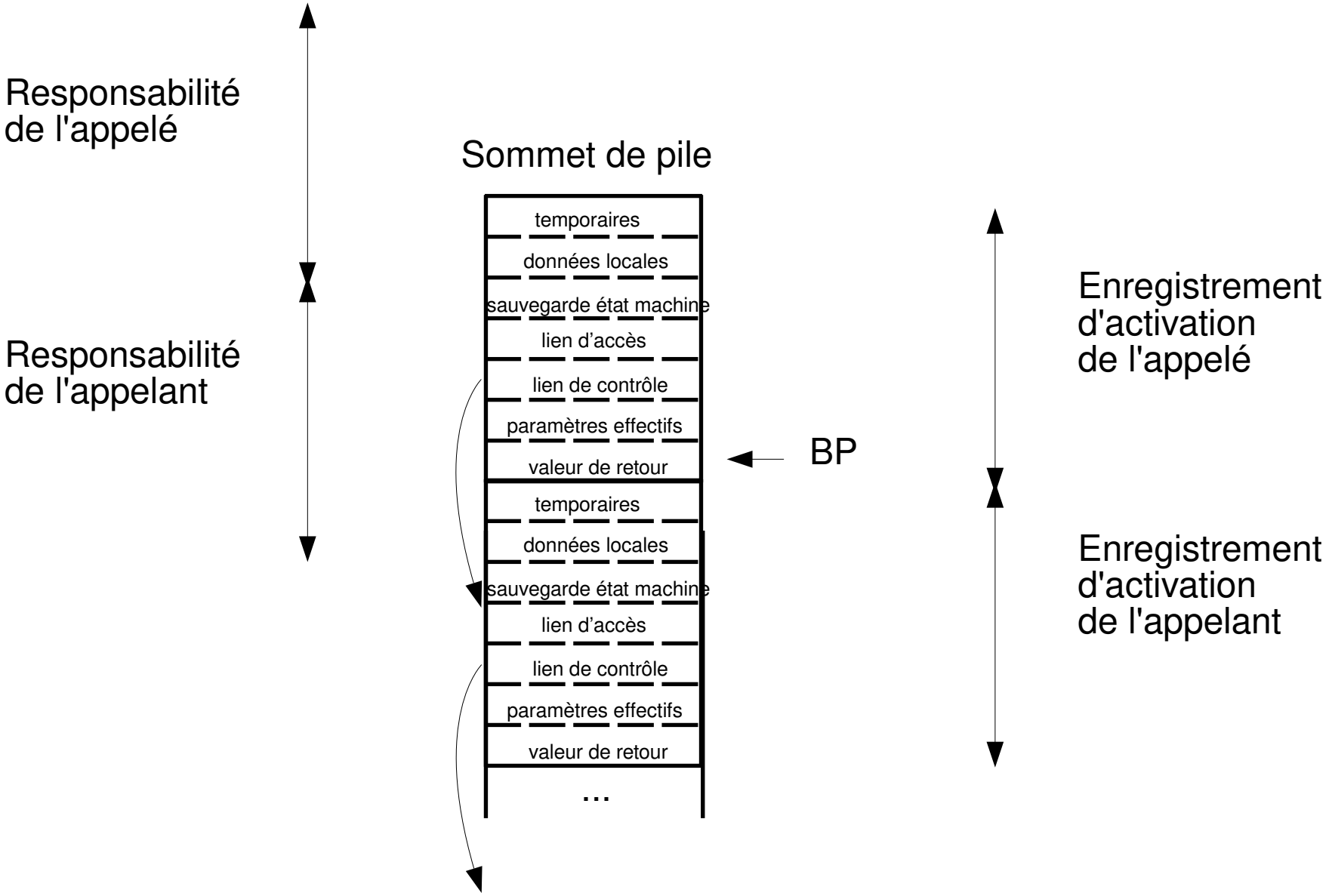
Tout doit être connu à la compilation
taille totale des données
Une place est réservée par le compilateur pour chaque donnée
En fin de compilation toute l'allocation est réalisée
A l'exécution, plus aucun travail d'allocation n'est à faire
A l'exécution, chaque donnée est directement accédée
le programme n'a pas de calcul d'adresse à réaliser pour accéder aux données
+ facile à mettre en œuvre, exécution rapide, données compactes
- le compilateur doit tout connaître sur les données, pas de structures dynamiques, récursivité très limitée
Remarque : rien n'interdit d'allouer statiquement les enregistrements d'activation si la récursivité est interdite

Solution pour une architecture classique

Stratégies d'allocation des enregistrements d'activation : allocation dans la pile

- Toute exécution de fonction :
  - commence par la construction d'un enregistrement d'activation dans la pile
  - termine par la destruction de cet enregistrement
    - les variables locales sont perdues en sortant de la fonction
    - la durée de vie des variables locales interne à un bloc est celle de l'enregistrement d'activation
- Nécessite deux registres de manipulation de pile :
  - *Stack Pointer* : désigne le sommet de pile
  - *Base Pointer* : désigne dans la pile l'enregistrement d'activation de la fonction en cours d'exécution
- Le code de la fonction appelant *f*
  - crée l'enregistrement d'activation de *f* en haut de la pile avant l'appel
  - détruit l'enregistrement après l'exécution de *f*
- Il faut établir un *protocole d'appel* entre l'appelant et l'appelé
  - il définit le rôle de chacun dans la manipulation de l'enregistrement d'activation
  - le compilateur implante ce protocole
    - il génère le code associé pour chaque appel de fonction
  - ce protocole, très sollicité, est très dépendant de l'architecture matérielle

Solution pour une architecture classique



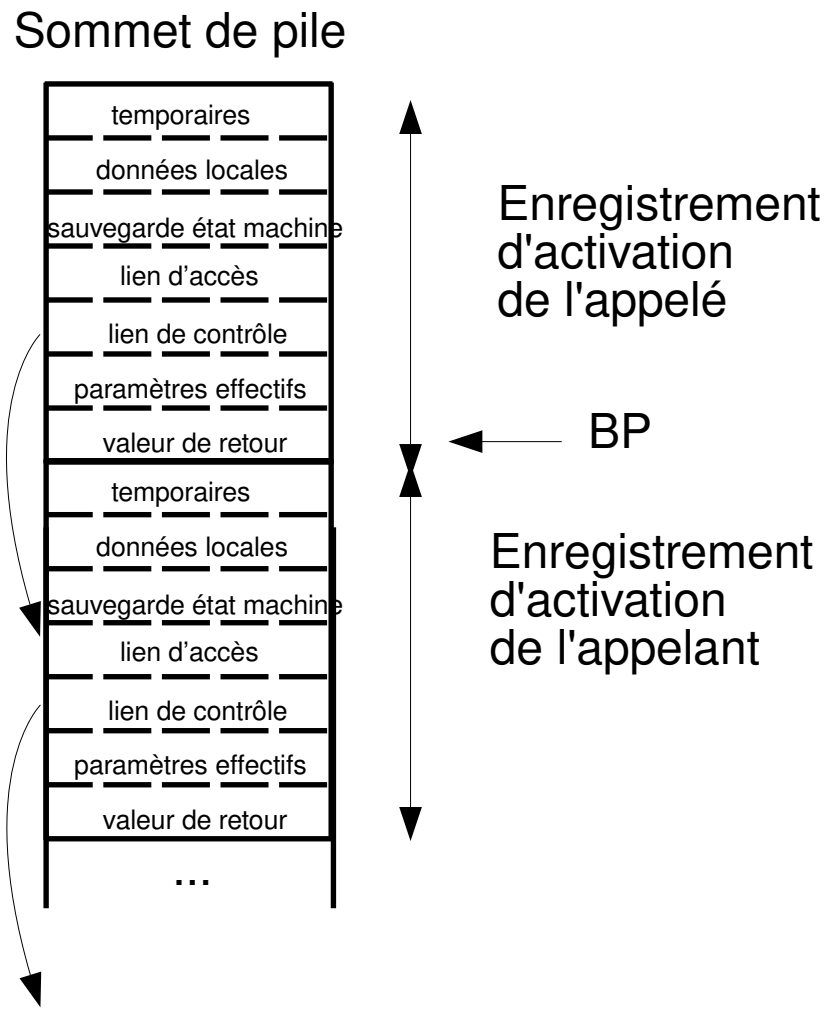
Solution pour une architecture classique

Exemple de protocole d'appel :
l'appelant

1) réserve sur la pile de la place pour la valeur de retour
2) évalue dans l'ordre chacun des arguments et les empile
3) met sur la pile un pointeur sur son propre enregistrement d'activation
4) met sur la pile l'adresse de son instruction à exécuter au retour de l'appelant
5) l'exécution de l'appelé débute

l'appelé

6) sauvegarde sur la pile des registres qu'il va utiliser
7) réserve sur la pile de la place pour ses données locales
8) utilise ensuite la pile pour ses calculs. Dans le code généré par le compilateur, l'accès aux variables locales et paramètre est réalisé par un calcul relatif à BP
9) sauvegarde sa valeur de retour à l'emplacement réservé à cet effet par l'appelant
10) supprime de la pile la place réservée pour ses données locales
11) restaure les registres qu'il avait sauvegardé
12) se termine : saut à l'adresse de l'instruction que l'appelant avait mise sur la pile l'appelant
13) retire de la pile le pointeur sur son propre enregistrement d'activation
14) retire de la pile les arguments qu'il avait empilé
15) récupère la valeur de retour de l'appelé
16) continue son exécution



Solution pour une architecture classique

Dans le mécanisme précédent, le code de l'appelant généré par le compilateur accède aux variables locales et paramètres en calculant leurs adresses relativement à BP

Ce calcul n'est possible, si ce sont les valeurs des variables locales et paramètres qui sont empilées, que si leur taille est connue du compilateur

Dans le cas où la taille d'une valeur de variable locale ou de paramètre n'est pas connue du compilateur, il peut la remplacer par une structure.

- Par exemple, remplacer un tableau de taille inconnue à *compile-time* par un couple (pointeur sur les données, nombre de données)
- Plutôt que d'empiler la donnée de taille variable, le compilateur empile la structure, dont il connaît la taille
- Le compilateur génère le code de l'appelant en sachant qu'il accède aux données à travers la structure

Solution pour une architecture classique

Remarque : l'utilisation de la structure est une indirection (perte de performances)

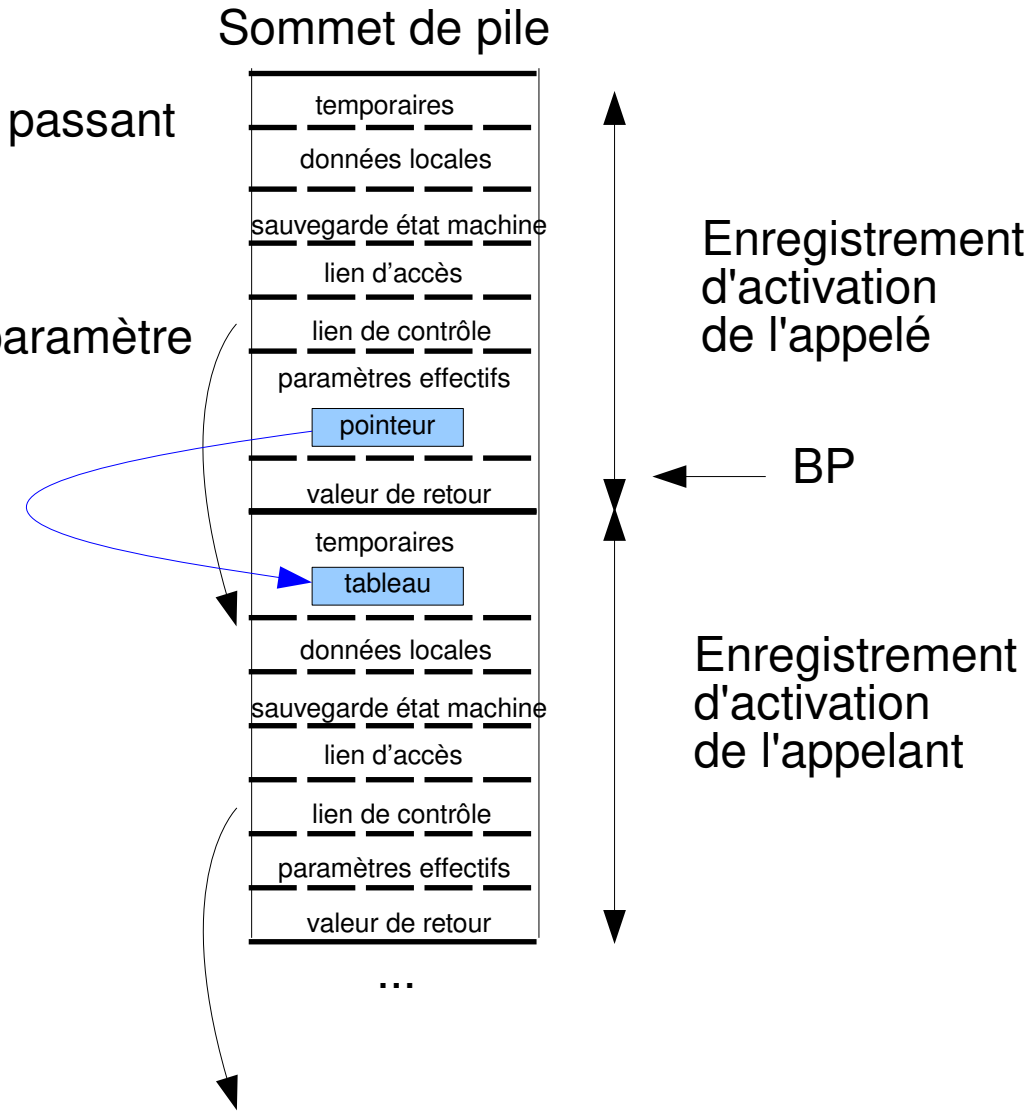
Remarque : la structure peut aussi être un simple pointeur.
- C'est une des raisons pour lesquelles en C,
  - les chaînes de caractères sont toujours représentées par des pointeurs
  - les tableaux sont toujours passés en paramètre de fonction par des pointeurs

Remarque : les données désignées par le compilateur peuvent être placées par l'appelant
- dans la zone « temporaires » de son propre enregistrement d'activation
  - Attention cependant, la zone réservée en mémoire pour la pile est souvent fixée par l'architecture matérielle et de petite taille !
- dans le tas en réalisant de l'allocation dynamique
  - Pas de problème de taille, mais la gestion de l'allocation dynamique est lourde, peu efficace en terme de temps d'exécution et d'occupation de l'espace mémoire, et les accès peuvent être plus lents (défauts de page plus fréquents)

Solution pour une architecture classique

Une fonction *f* en appelle une autre *g* en lui passant un tableau *t* en paramètre :

1) *f* commence par faire une copie de *t* (ici, sur sa zone « temporaires »)
2) dans son code préparant l'appel à *g*, le paramètre passé est en fait un pointeur



Solution pour une architecture classique

Le processeur possède un ou plusieurs registres dédiés à la pile
- Exemple sur i86 :
  - SS (Stack Segment) : contient l'adresse de l'espace réservé à la pile
  - SP (Stack Pointer) : contient l'adresse du sommet de pile relativement à SS
  - BP (Base Pointer) : contient l'adresse à partir de laquelle on trouve la sous-pile de la fonction en cours d'exécution

Les données locales (variables locales et paramètres) à l'exécution d'une fonction sont toutes mises dans la pile, à une adresse calculée par le compilateur relativement à BP et suivant un ordre convenu

Le processeur dispose de deux instructions *call* et *ret*
- *call x* mets sur la pile l'adresse de l'instruction qui la suit, et demande au processeur d'exécuter l'instruction à l'adresse *x*
- *ret* dépile, considère la valeur dépilée comme une adresse x, et demande au processeur d'exécuter l'instruction à l'adresse *x*

La pile contient donc :
- des données provisoires
- des valeurs de paramètres de fonctions
- des valeurs de variables locales de fonctions
- des adresses d'instructions !

Solution pour une architecture classique

Exemple de convention d'appel (Il en existe beaucoup ! (par processeur, compilateur, ...)) :

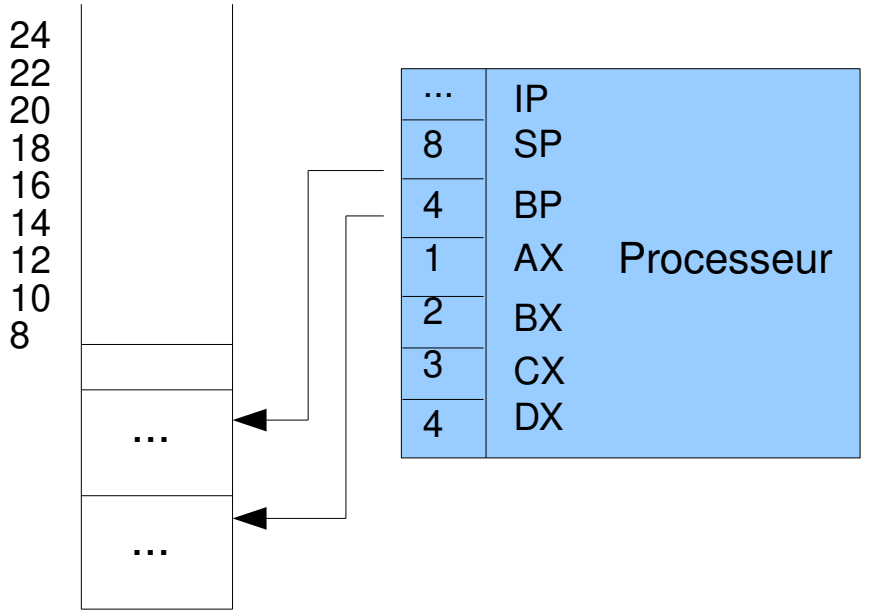
- avant l'utilisation de call (pour appeler f, à l'adresse @f)
  - réserver, sur le sommet de pile, la place nécessaire au stockage des variables locales de f
  - empiler tous les arguments de f dans un ordre donné
- générer le call
  - l'exécution de call mettra l'adresse de l'instruction suivant call en somme de pile
- le code de f
  - commence par
    - mettre bp en sommet de pile
    - bp <- sp
  - référence variables locales et arguments par leurs adresses relativement à BP
  - garanti que, juste avant le ret, la pile aura
    - a son sommet la valeur de retour,
    - juste en dessous, l'ancienne valeur de BP
    - (en donc, juste en dessous, l'adresse de l'instruction suivant le call)
  - termine par
    - dépiler la valeur de retour, la mettre dans un registre particulier
    - restaurer l'ancienne valeur de BP
    - exécuter le ret

Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax
pop bx
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

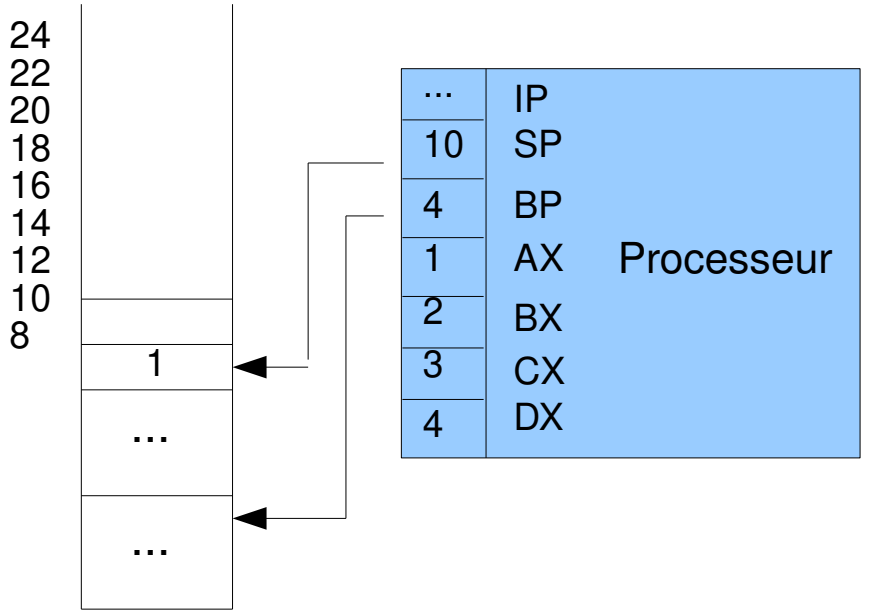


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax
pop bx
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

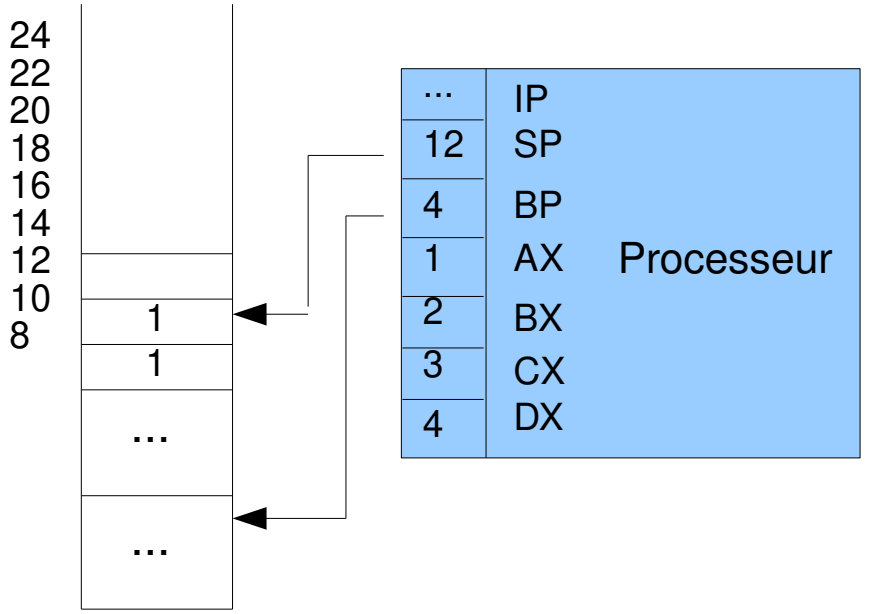


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax
pop bx
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

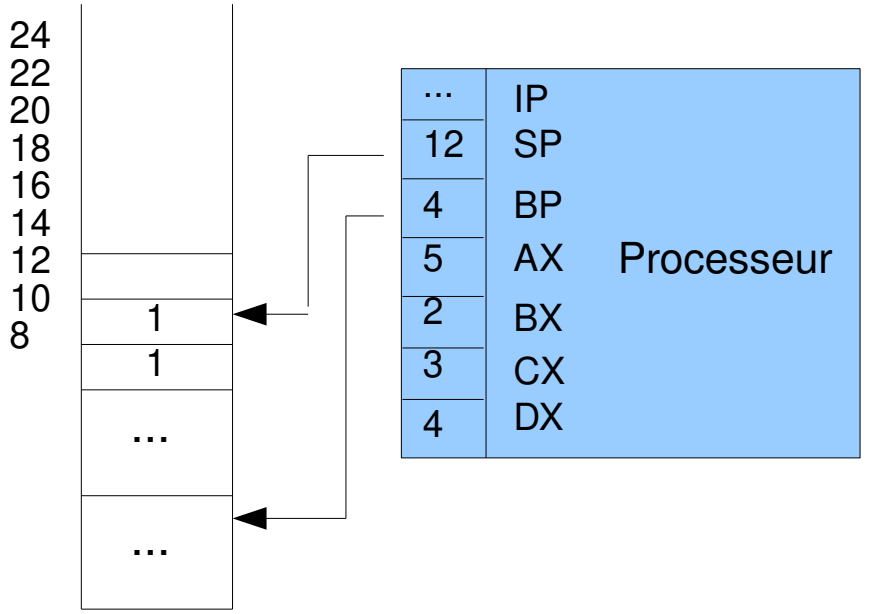


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax
pop bx
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

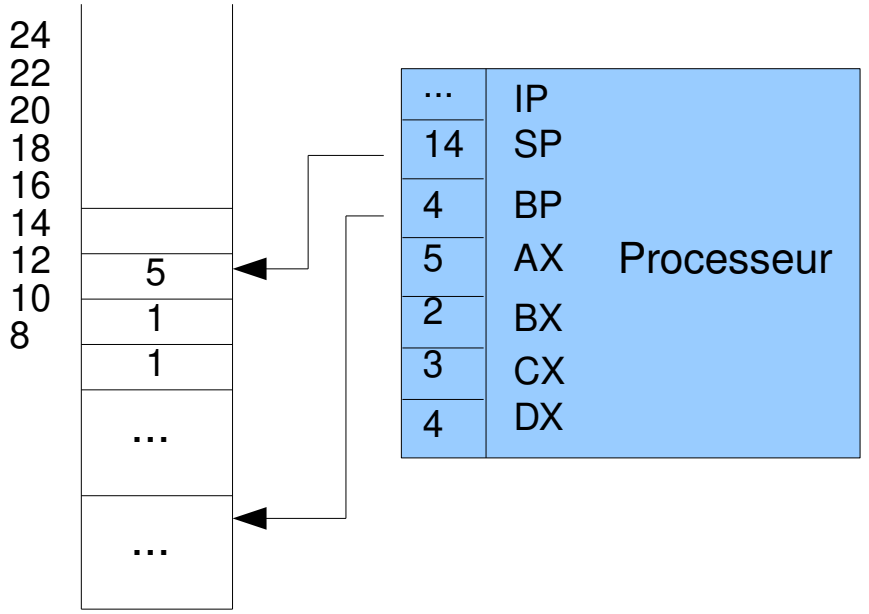


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax
pop bx
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

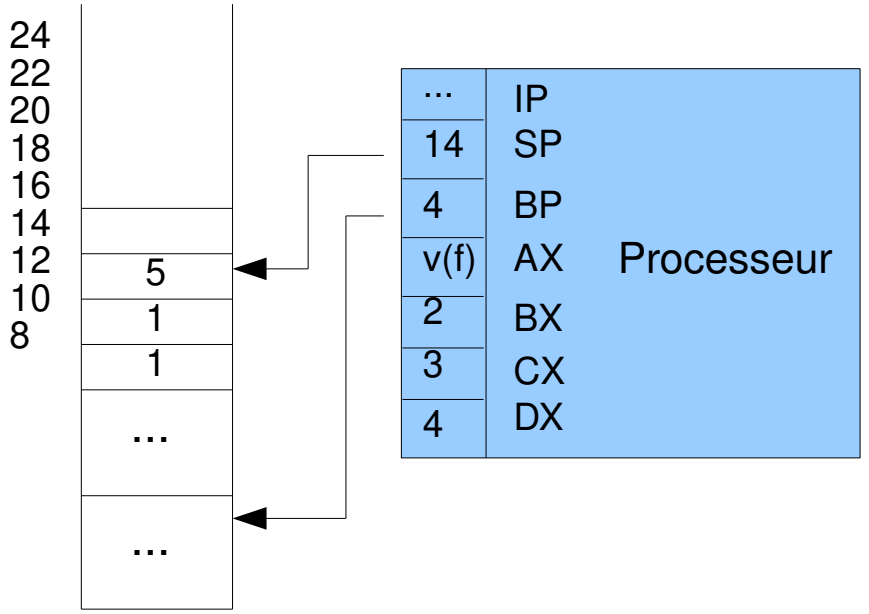


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax
pop bx
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

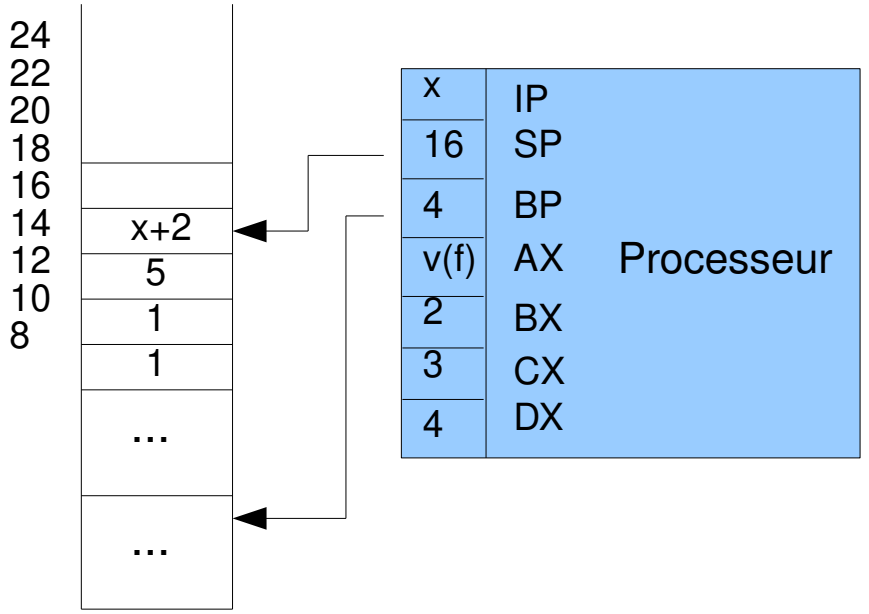


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

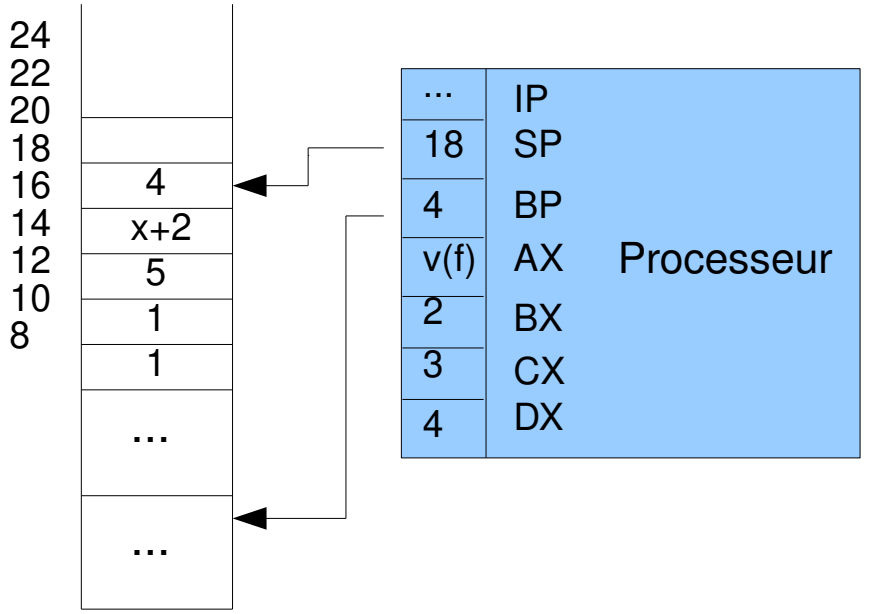


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

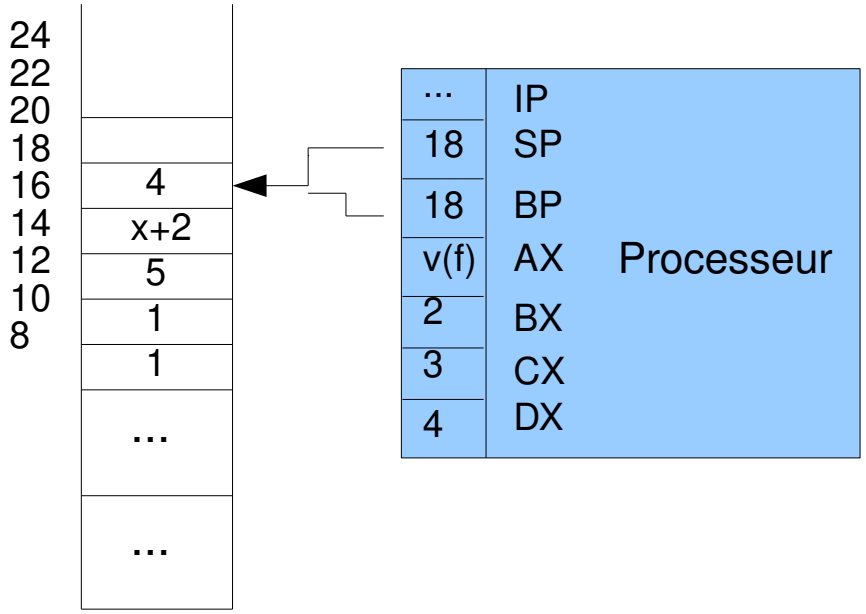


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

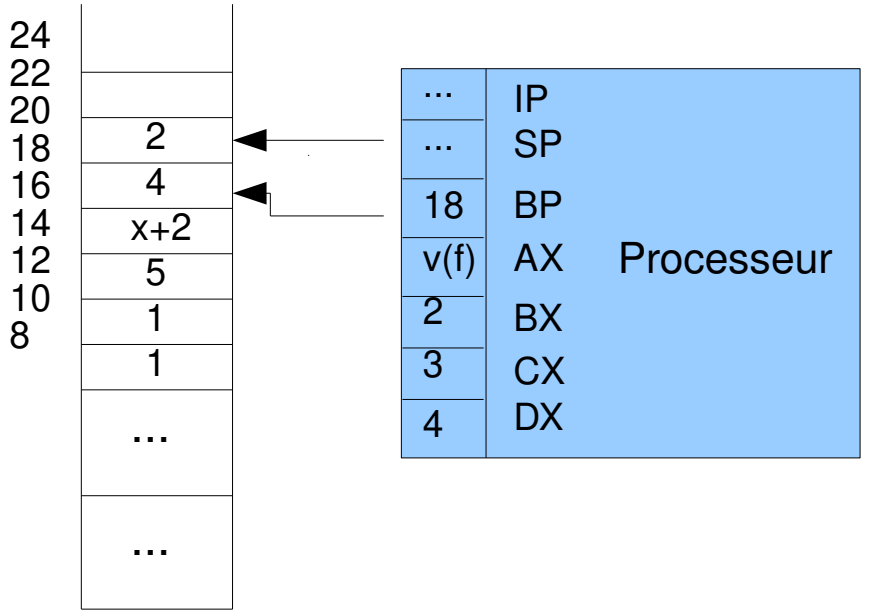


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

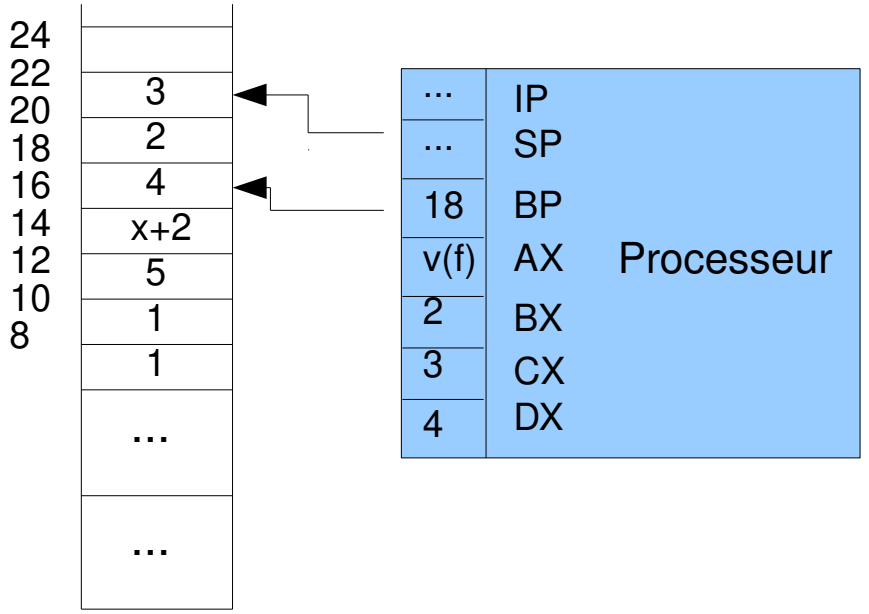


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

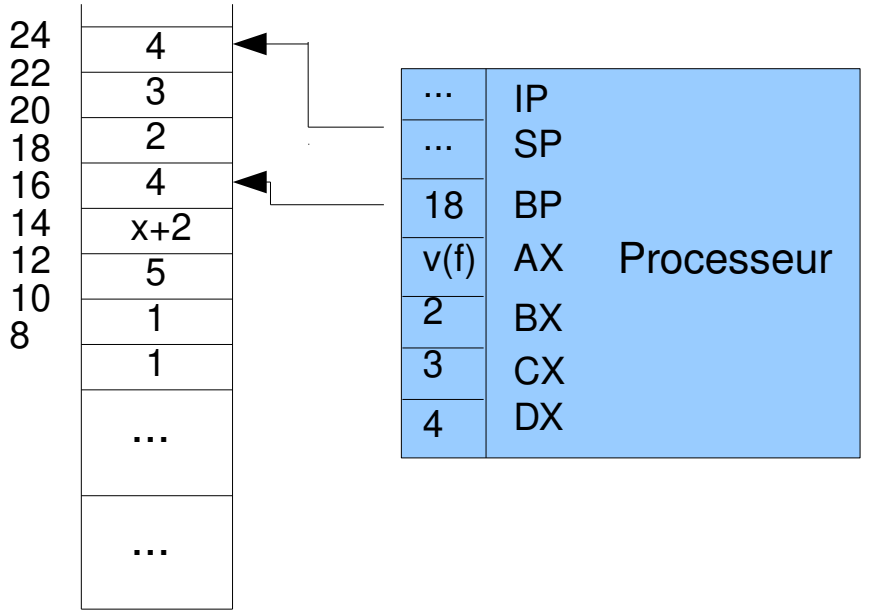


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

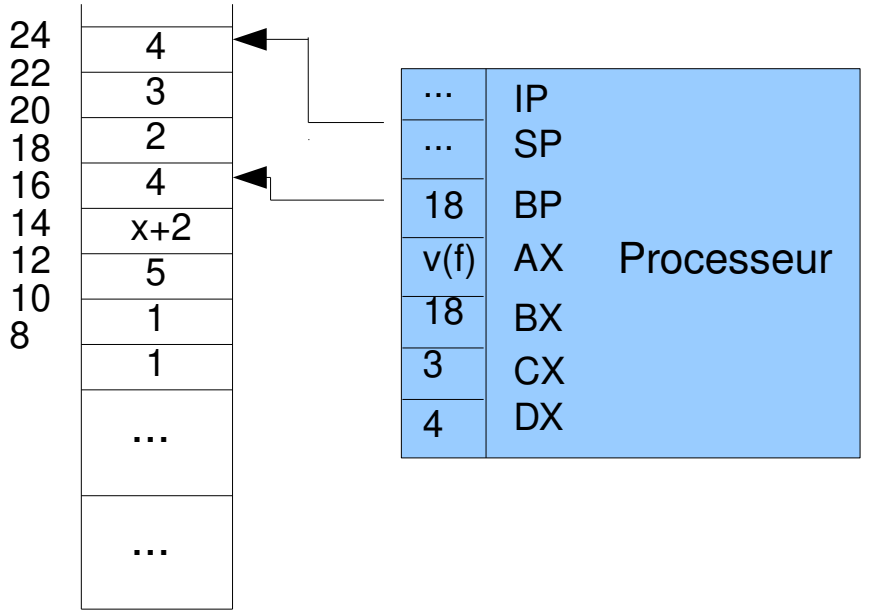


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

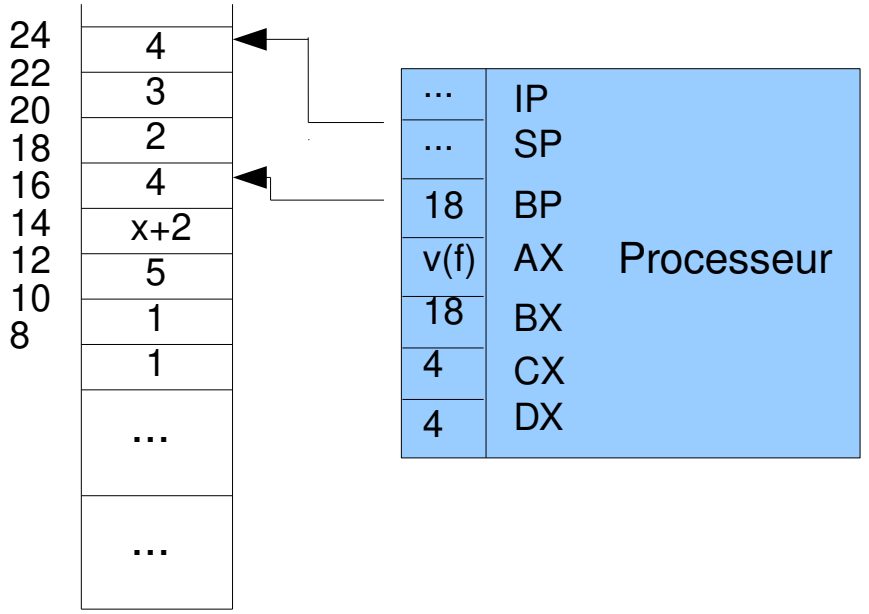


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

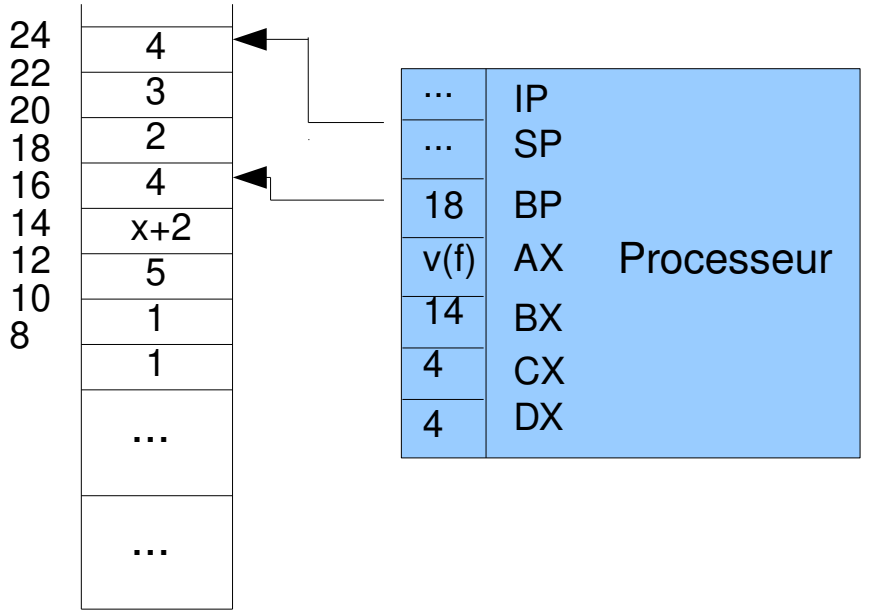


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

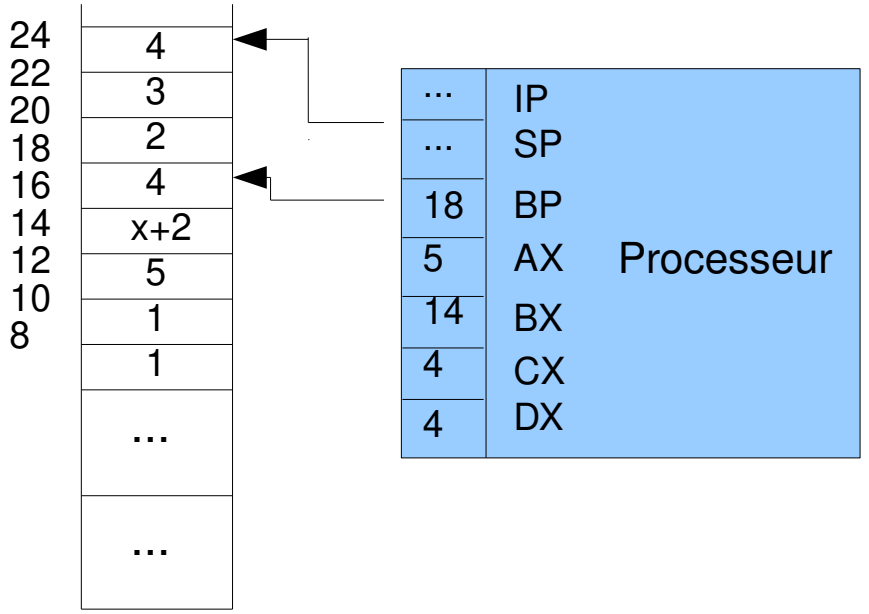


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

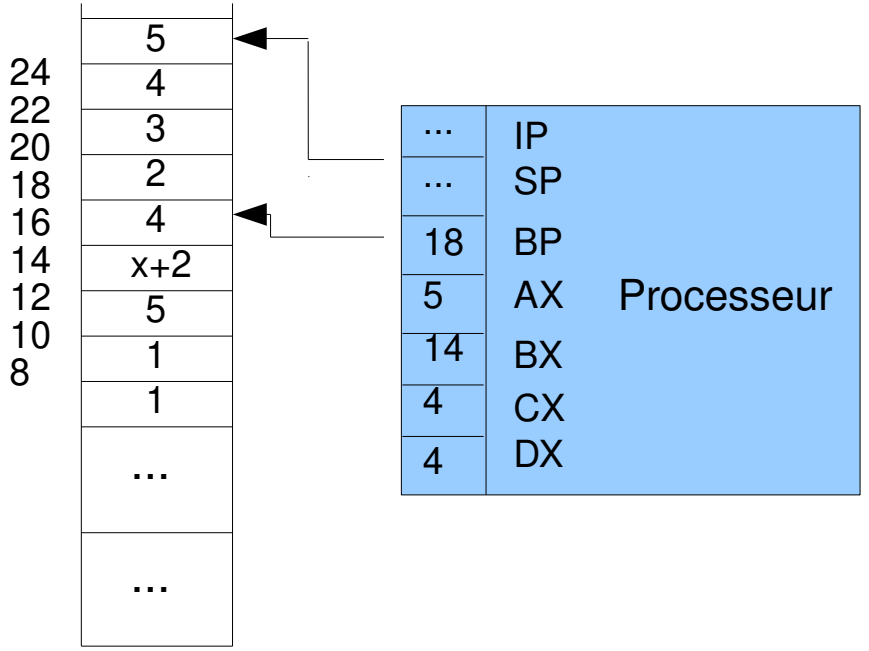


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

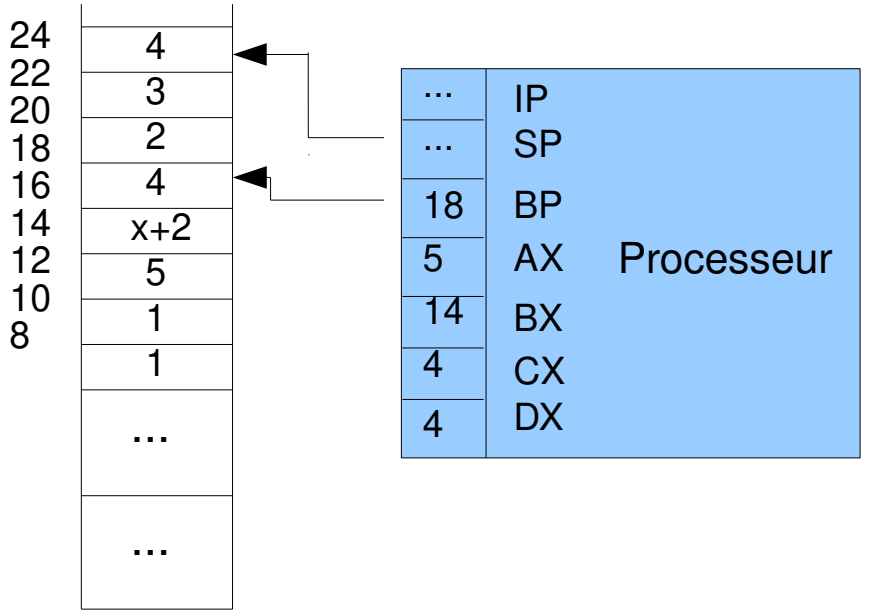


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

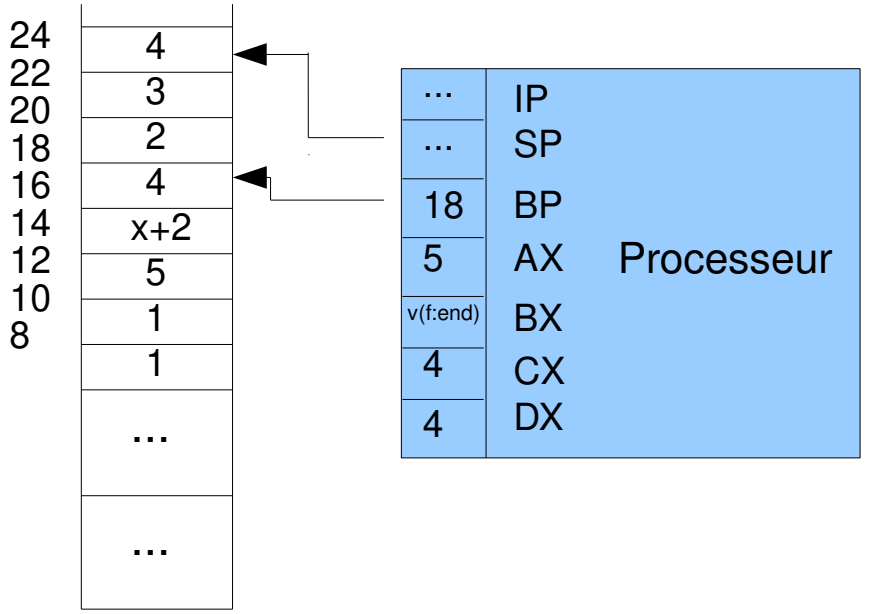


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

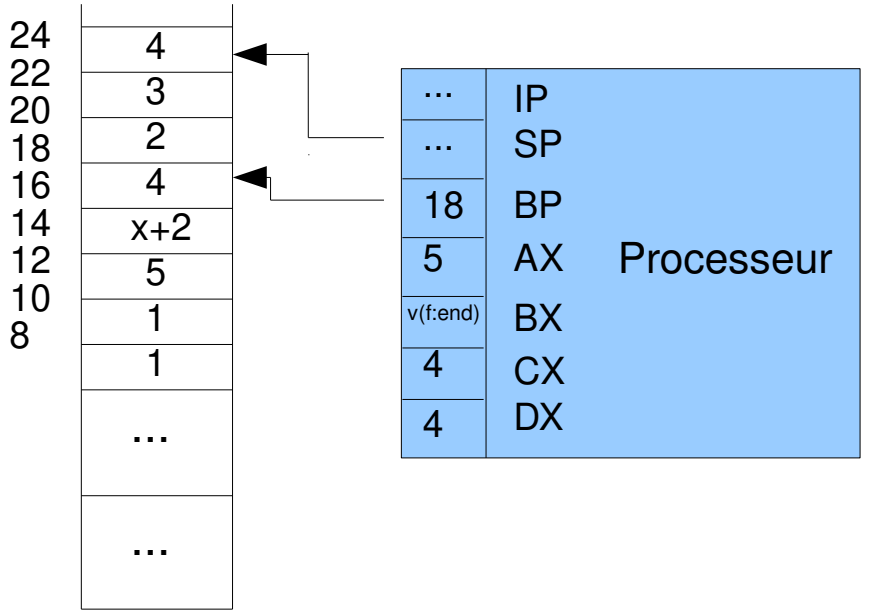


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

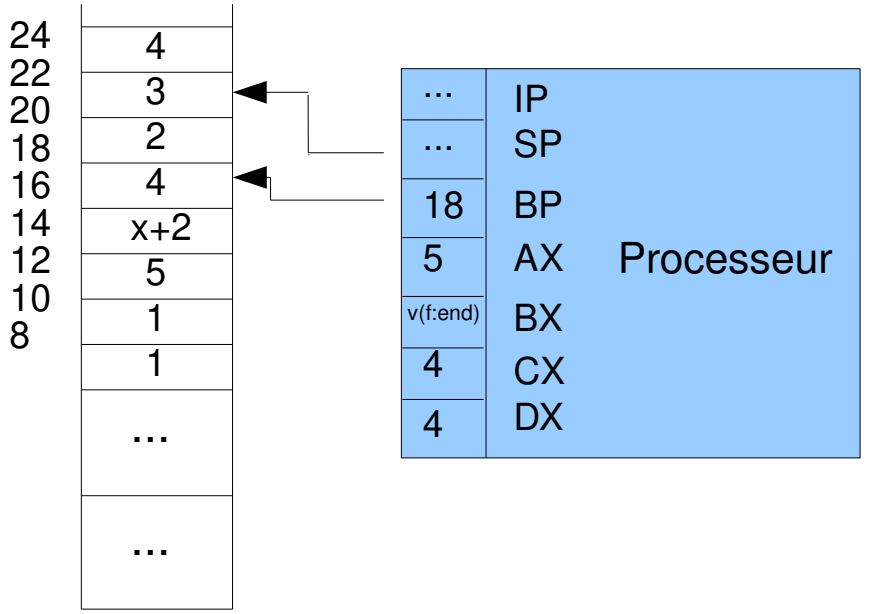


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

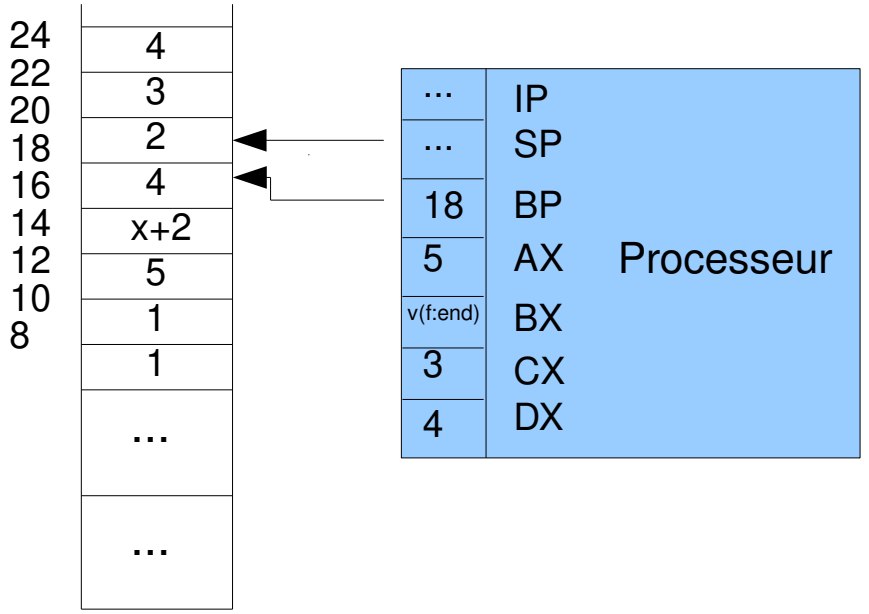


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

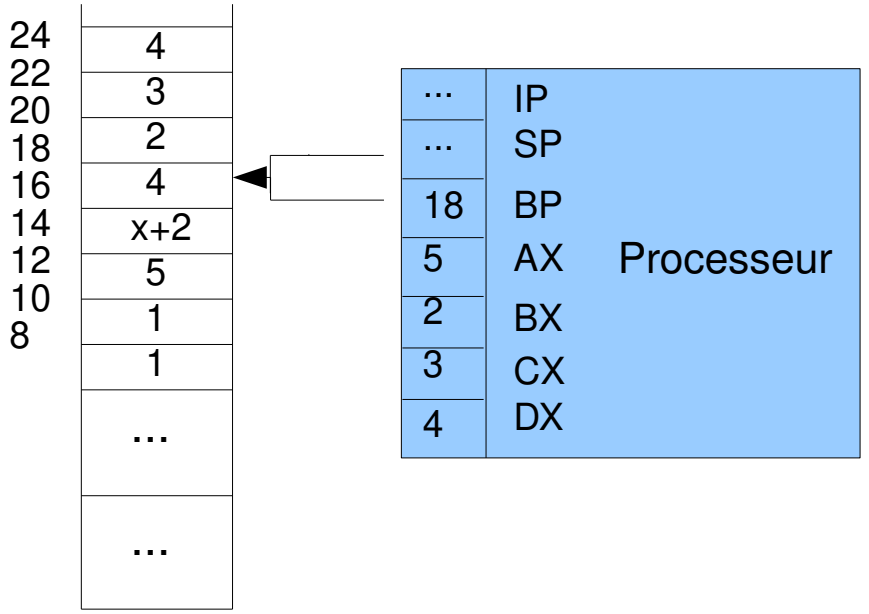


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

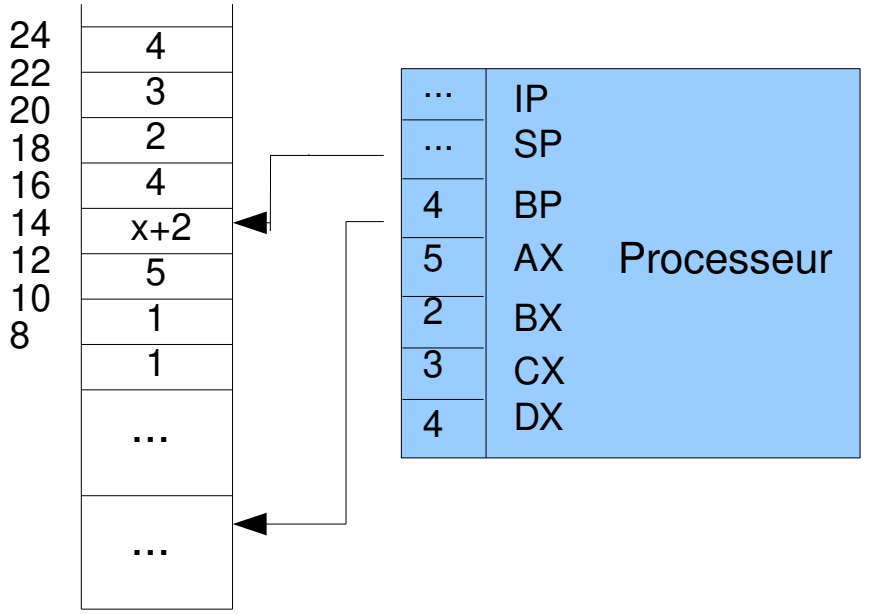


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

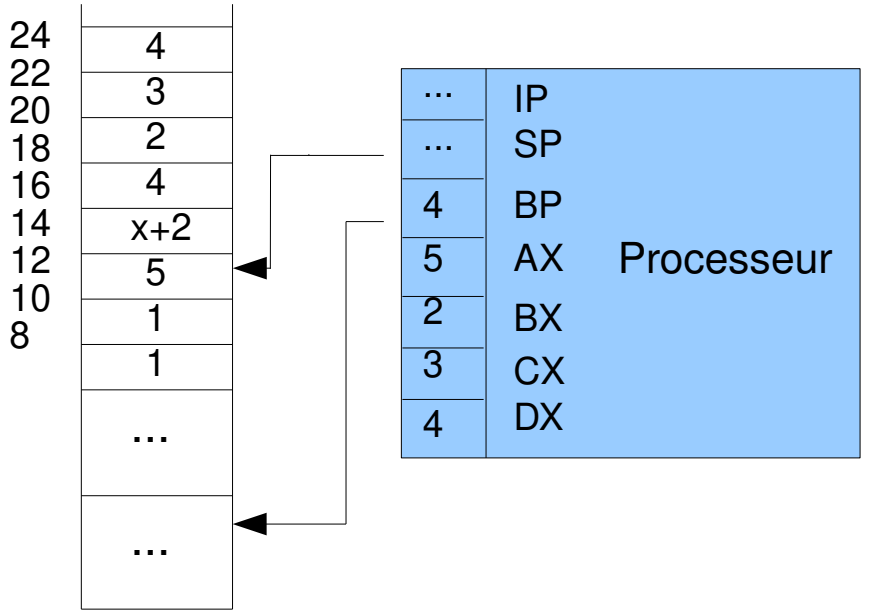


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

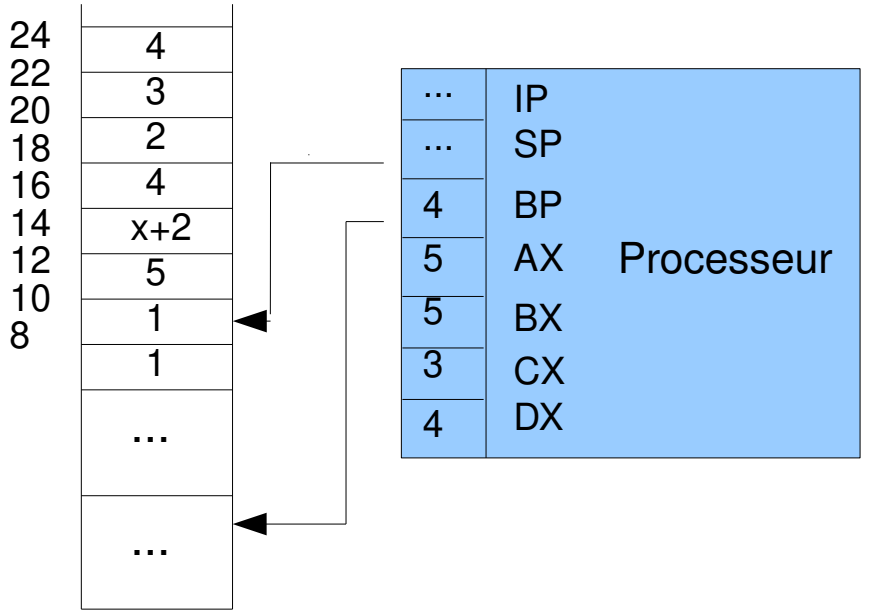


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

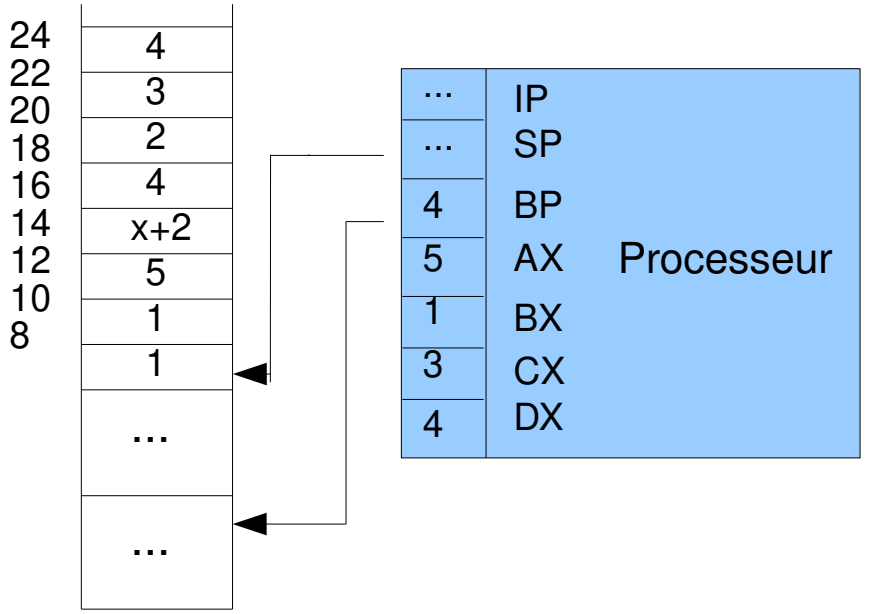


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```

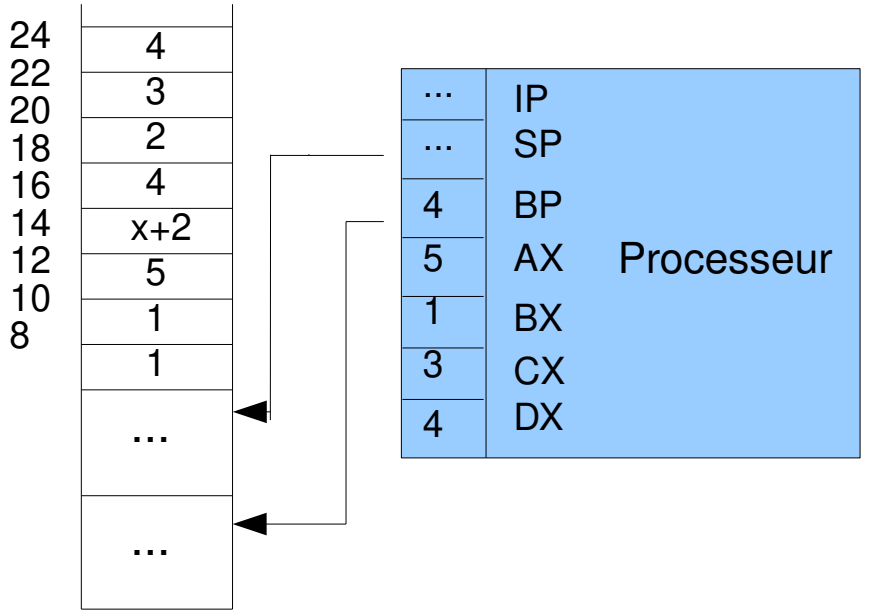


Solution pour une architecture classique

```
:f
;       Start of function f
push bp
cp bp,sp
push bx
push cx
push dx
;       Local variable r
cp bx,bp
const cx,4
sub bx,cx
loadw ax,bx
push ax
pop ax
const bx,f:end
jmp bx
;       End of function f
:f:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
:main
;       Start of function main
push bp
cp bp,sp
push bx
push cx
push dx
;       Preparing function call for f
push ax
push ax
;       Number constant
const ax,5
push ax
const ax,f
call ax ; instruction à l'adresse x
pop bx ; instruction à l'adresse x+2
pop bx
pop bx
push ax
pop ax
const bx,main:end
jmp bx
;       End of function main
:main:end
pop dx
pop cx
pop bx
pop bp
ret
```

```
int f(int r) { int a; int b; return r; }

int main() {
    return f(5);
}
```



Notion de bloc

Notion introduite par ALGOL (1960).

Les blocs comportent une partie déclaration de variables locales et des instructions.

Les blocs ne doivent pas se chevaucher (imbrication hiérarchique).

La portée de la déclaration d'une variable s'étend à tous les blocs inclus où cette variable n'est pas redéclarée.

Ceci permet d'implanter les blocs dans la pile de contrôle comme des procédures sans paramètres et ne retournant pas de valeur.

- la notion d'enregistrement d'activation est généralisée aux blocs, avec une version simplifiée
- entrée dans le bloc : création d'un nouvel enregistrement d'activation
- sortie du bloc : suppression de l'enregistrement d'activation
- peut être simplifié
  - pas d'enregistrement d'activation pour les blocs seulement pour les fonctions
  - les enregistrements d'activation des fonctions contiennent de la place pour toutes les variables locales
  - les noms des variables locales et leurs portées sont gérées par du *masquage*

Attention aux instructions de saut, si les sauts sont autorisés !

Table des symboles

Elle associe des informations à chaque symbole (nom de variable, fonction, classe, étiquette, …)
Par exemple :

- classe de stockage (tas, pile, zone des constantes, ...)
- accessibilité (restreinte, ouverte, ...)
- adresse (pour les fonctions, les l-values, ...)
- type (entier, réel, fonction, ...)
- nature (variable globale, locale, paramètre de fonction, classe, interface, ...)
- valeur (pour les langages fonctionnels, par exemple)

Table : chaîne -> information

Fonctionnalités de base :

- insertion d'une nouvelle association chaîne -> information
- suppression des informations associées à une chaîne
- recherche des informations associées à une chaîne

Fonctionnalités avancées

- intégration de la notion de portée des identificateurs

Implantations courantes

- liste chaînée
- table de hachage

Table des symboles, portée des identificateurs et masquage

Tout symbole défini dans un bloc est associé au bloc
Les autres symboles sont globaux

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

```
int a0 = 1 ;

void f(int a1) {1
        int b1 = 2 ;
        printf(«%d %d », a1, b1) ;
        {2
                int a2 = 3 ;
                printf(«%d », a2, b1) ;
                {3
                        int a3 = 4 ;
                        printf(«%d », a3, b1) ;
                }3
                printf(«%d », a2, b1) ;
        }2
        printf(«%d », a1, b1) ;
}1
```

Table des symboles, portée des identificateurs et masquage

La numérotation peut être réalisée pendant l'analyse syntaxique

Les noms peuvent aussi être modifiés pendant l'analyse syntaxique
- a -> a#bloc

Remarque :
- à l'extérieur d'un bloc, l'ensemble des symboles qui y sont définis sont inutiles
- mais la durée de vie de l'espace mémoire désigné par le symbole peut être supérieure à celle du symbole

Gestion efficace des symboles
- Ils sont ajoutés dans la table des symboles ou dans l'environnement d'exécution à l'entrée du bloc
- Ils sont supprimés de la table des symboles ou de l'environnement d'exécution à la sortie du bloc
- Ils sont accédés prioritairement aux autres

Table des symboles, portée des identificateurs et masquage : Implantation par liste chaînée

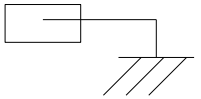
Dernier élément inséré en tête de liste. Exemple pour un schéma de traduction LR :

```
prog : prog definition | definition ;

definition : function | decl ;

function : type ID '(' lparams ')' {
        S'il existe déjà une fonction de nom ID
                Erreur (ni surcharge ni redéfinition)
        Sinon
                Calcul des informations
                Ajouter (ID, infos) à la table
} blocinstr ; {
        Supprimer les |$4| premiers éléments
                de la table
}

lparams : lparams ',' param {
        S'il existe déjà un paramètre de même nom que
                        celui de $3
                Erreur
        Sinon
                $$ = ajouter $3 à $1 }
| param { $$ = $1 ; }
;
```

```
param : type ID ; {
                Calcul des informations
                Ajouter (ID, infos) à la table
                $$ = nouvelle liste ($1, $2)

blocinstr : '{' ldecl linstr '}' ; {
        Supprimer les |$2| premiers éléments
                 de la table
}
ldecl : ldecl decl {
        S'il existe déjà dans $1 une déclaration de
                        même nom que celui de $2
                Erreur
        Sinon
                $$ = ajouter $2 à $1
 }
| %empty        { $$ = liste vide ; };

decl : type ID ';' {
        Calcul des informations
        Ajouter (ID, informations) à la table
        $$ = nouvelle liste ($1, $2)
}

instr dépendant de ID : …
        rechercher ID dans la table
        utiliser sa valeur...
```

Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

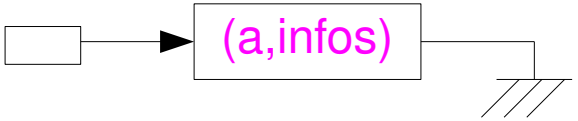
Evolution de la liste pendant l'analyse syntaxique



```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique




```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a){
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

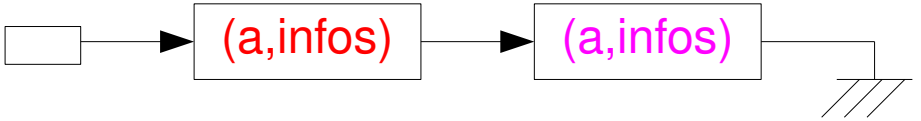


Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

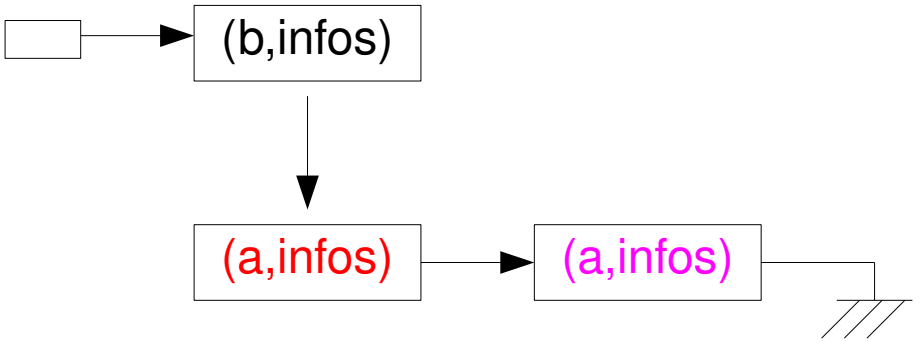


Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

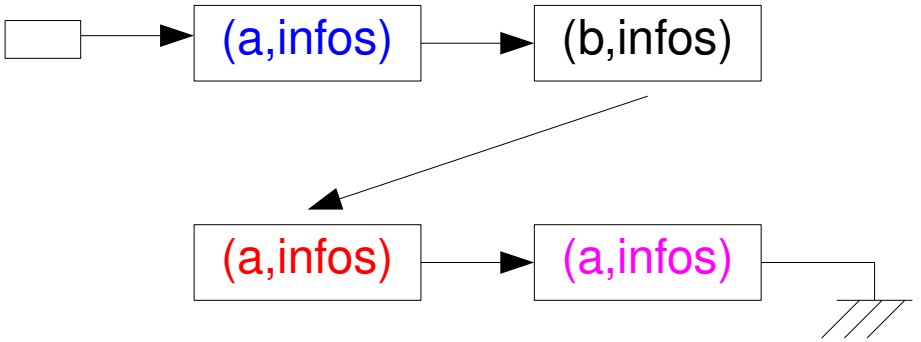


Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

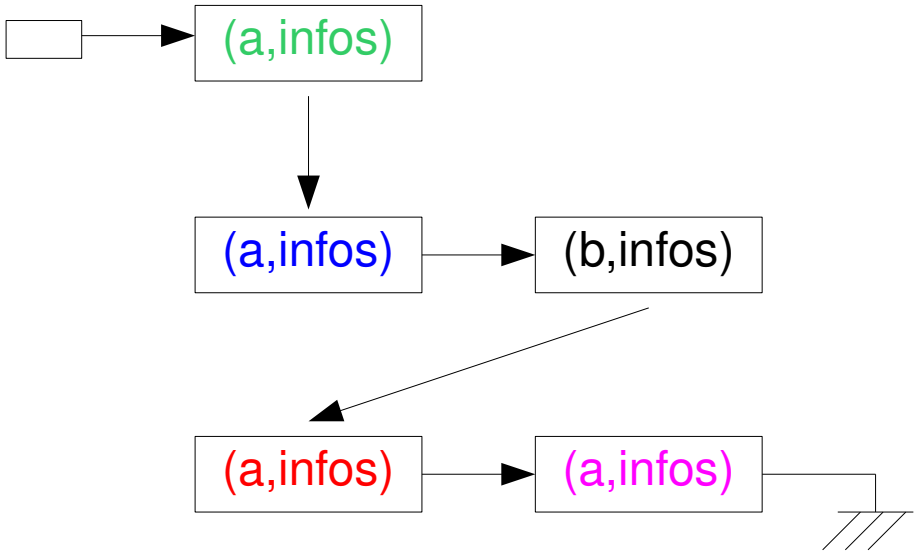


Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

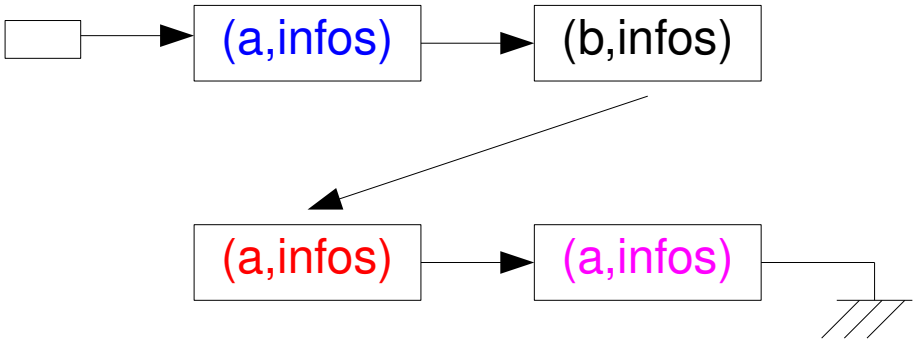


Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

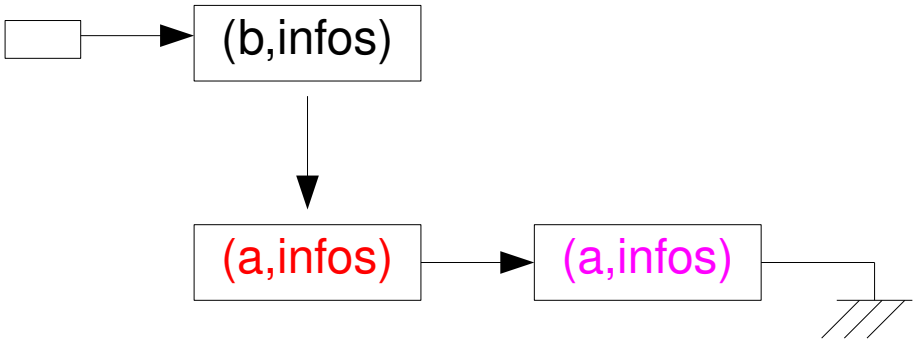
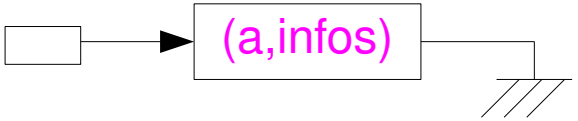


Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

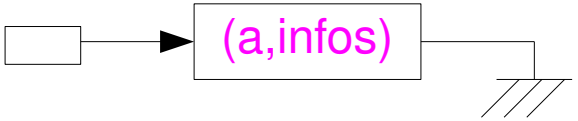



```
int a = 1 ;

void f(int a) {
	int b = 2 ;
	printf(«%d %d », a, b) ;
	{
		int a = 3 ;
		printf(«%d », a, b) ;
		{
			int a = 4 ;
			printf(«%d », a, b) ;
		}
		printf(«%d », a, b) ;
	}
	printf(«%d », a, b) ;
}
```

Table des symboles, portée des identificateurs et masquage : implantation par liste chaînée

Evolution de la liste pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```



Avantage :
- très simple à implanter

Inconvénient :
- complexités des opérations élémentaires : linéaire

Alternative : table de hachage
- plus difficile à implanter
- moins économique en mémoire
- complexité des opérations élémentaires : constante

Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

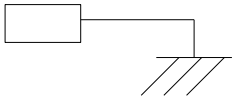
Hachage « fermé » : pas de re-hachage, le hachage amène à la liste des éléments ayant la même valeur de hachage

En sortie de bloc, il faut pouvoir détruire tous les symboles du bloc
- gestion d'une liste de pointeurs de maillons de la table de hachage, qu'on utilise pour réaliser les destructions
- chaque maillon de liste de la table de hachage doit pouvoir être supprimé en temps constant
  - il contient l'adresse du pointeur qui le désigne

Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```



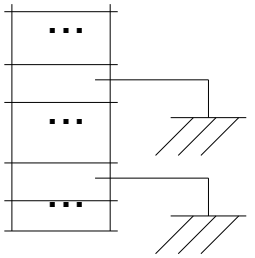


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

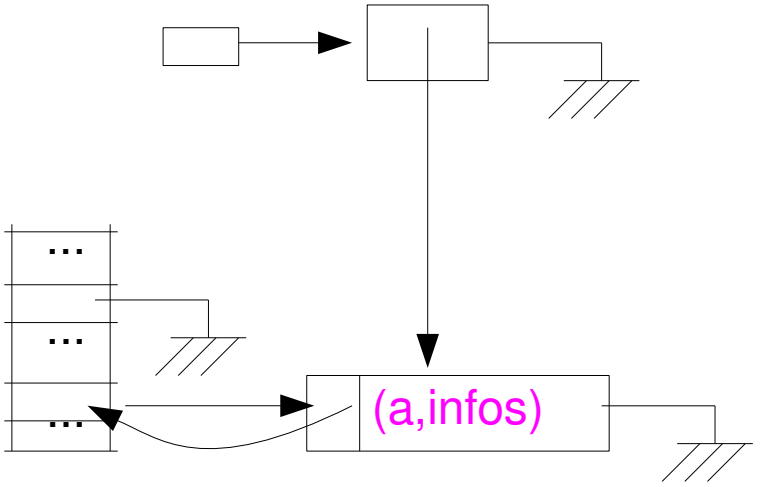


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a){
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

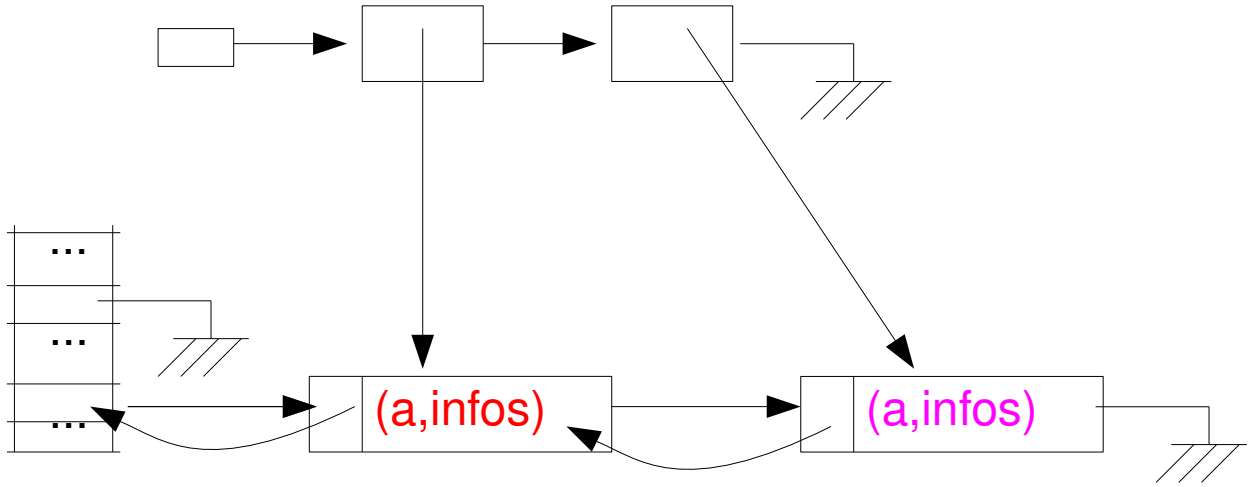


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

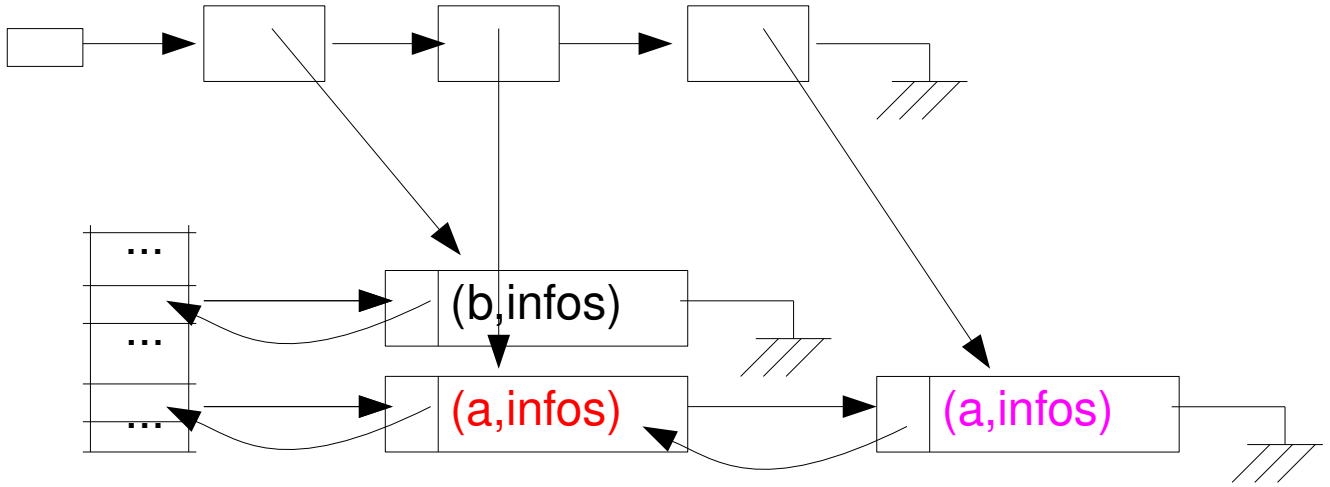


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

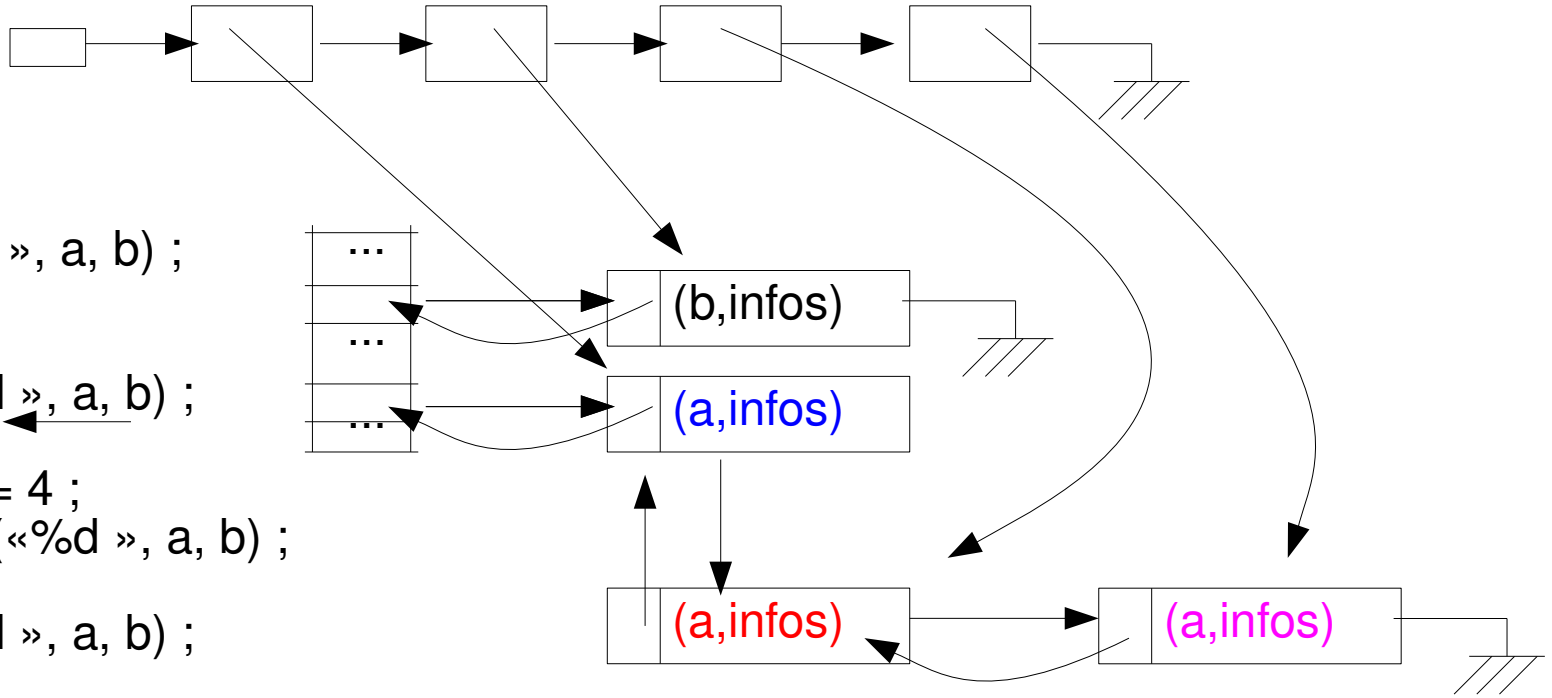


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

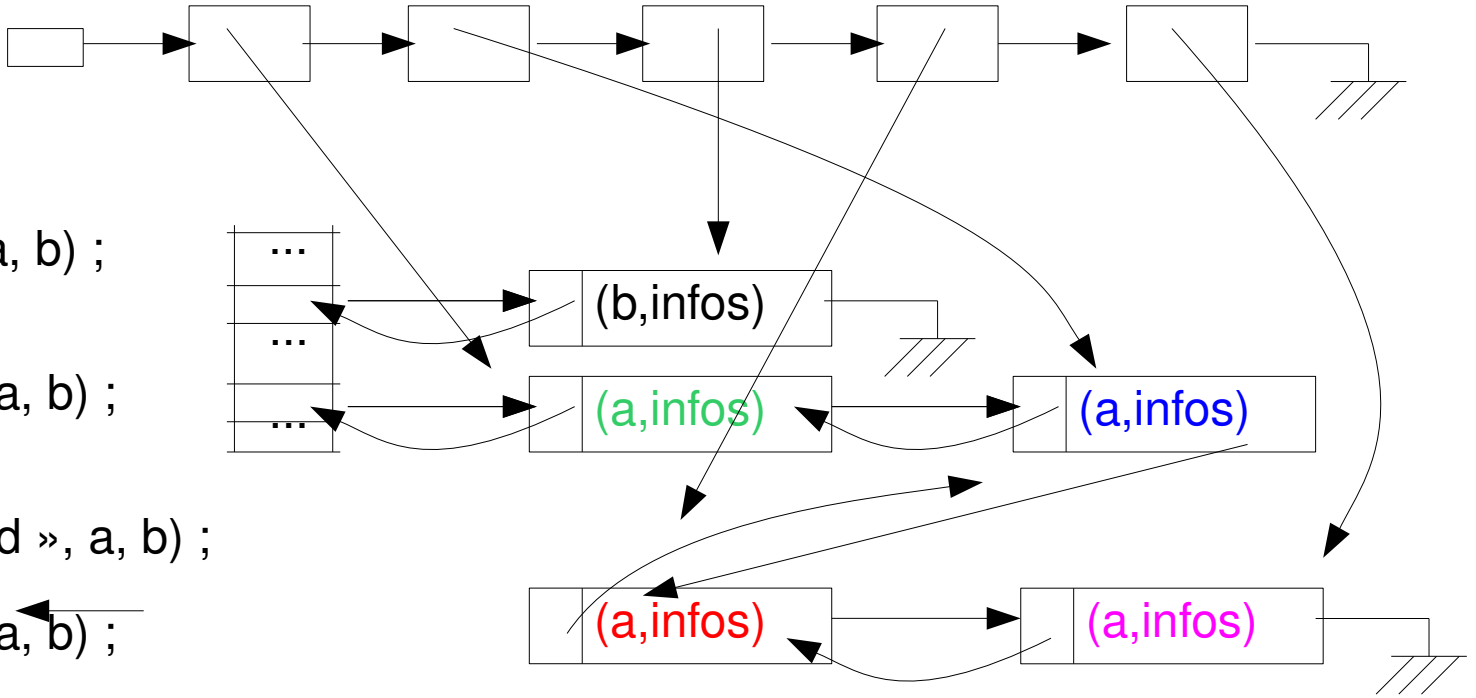


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

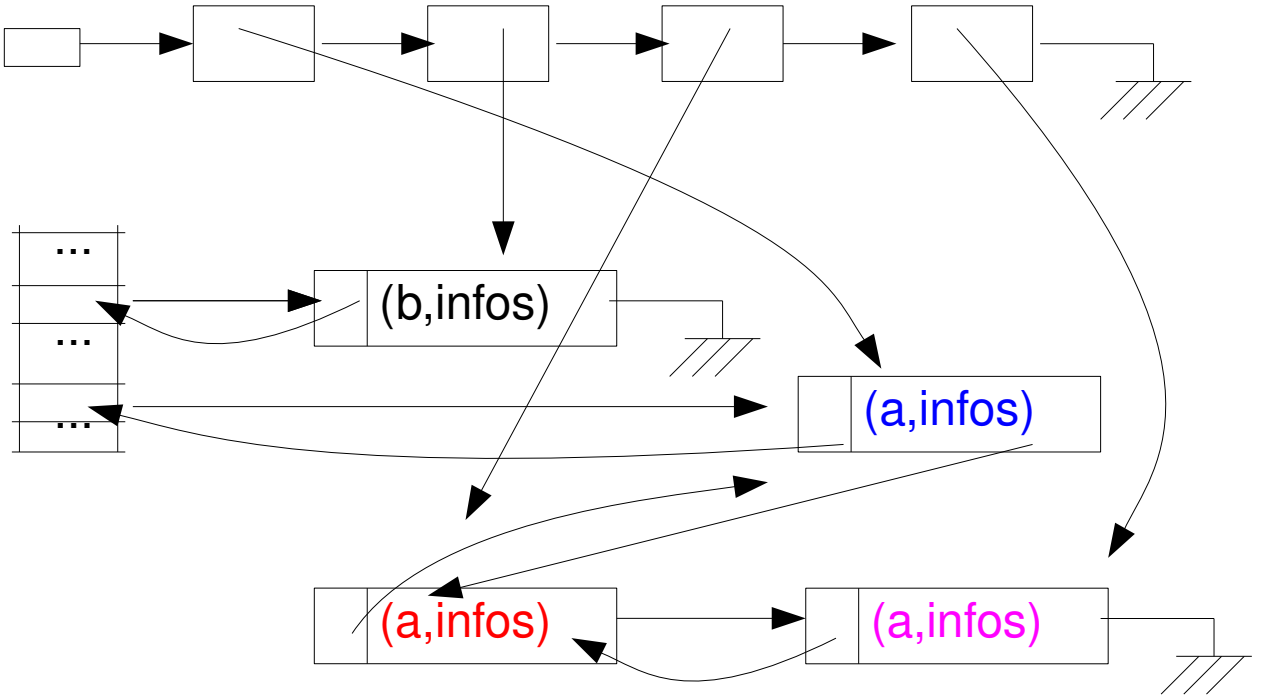


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```

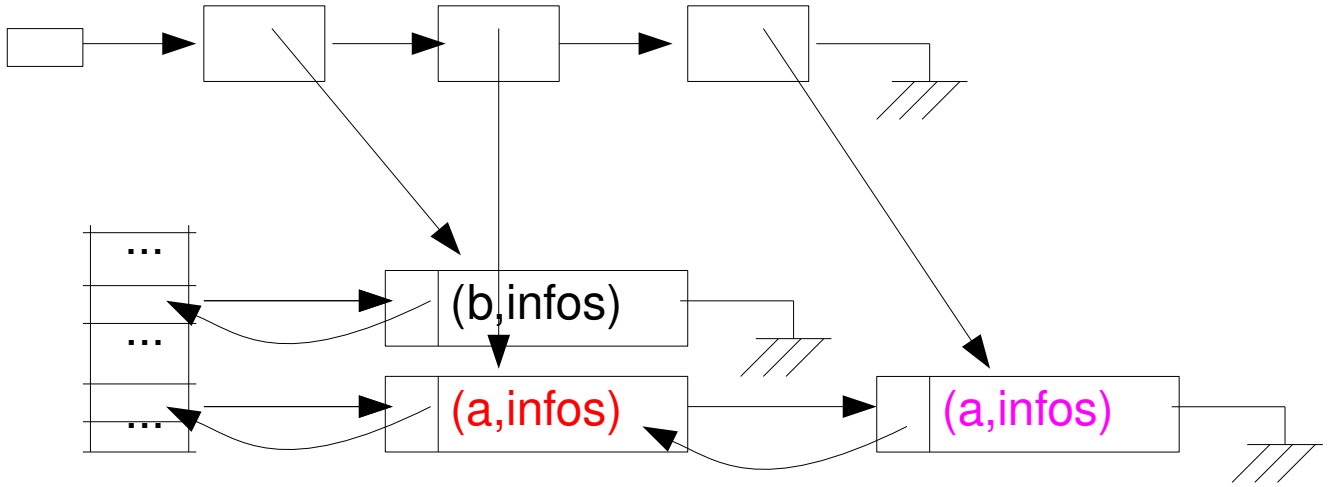
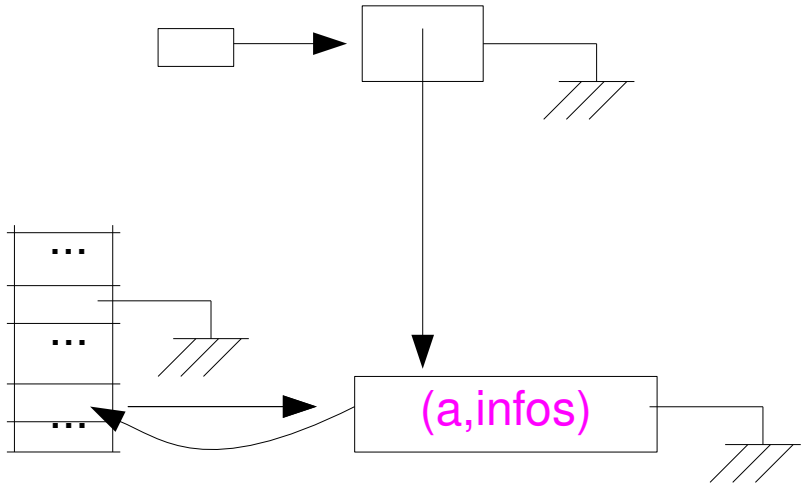


Table des symboles, portée des identificateurs et masquage : implantation par table de hachage

Evolution de la table (variables locales) pendant l'analyse syntaxique

```
int a = 1 ;

void f(int a) {
        int b = 2 ;
        printf(«%d %d », a, b) ;
        {
                int a = 3 ;
                printf(«%d », a, b) ;
                {
                        int a = 4 ;
                        printf(«%d », a, b) ;
                }
                printf(«%d », a, b) ;
        }
        printf(«%d », a, b) ;
}
```



Noms non locaux sans imbrication de fonctions

Ils sont tous globaux, comme en C.
Les espaces mémoires correspondant peuvent être localisés dans une zone globale et statique
Facilite l'implantation de langages dans lesquels les fonctions peuvent être des paramètres
ou des retours d'autres fonctions

```
int m;

int f(int n) { return m+n; }

int g(int n) {  return m*n; }

void b(int x(int n)) {  printf("%d", x(2)); }

int main(int argc, char *argv[]) {
  m=0;
  b(f);
  b(g);
  return EXIT_SUCCESS;
}
```

La zone de stockage globale contient
les codes de f, g, b, main
l'espace mémoire pour m

Leurs adresses sont calculées par le compilateur et sont fixes

Les fonctions sont passées en paramètres en utilisant leurs adresses

Idem si une fonction doit en retourner une autre

Noms non locaux sans imbrication de fonctions

```
program Statique (input, output)
    var r : real ;

    procedure Montrer :
        begin write r end ;

    procedure A :
        var r : real ;
        begin r:=0.125 ; Montrer end ;

    begin
        r:= 0.25 ;
        Montrer ; A ;
    end.
```

Affichage : 0.25 0.25

Noms non locaux avec imbrication de fonctions

N'existe pas en C, existe en Pascal :

```
program Trier (input, output) ;
   var t : array [0..10] of integer ;
   x : integer ;

   procedure Echanger (i, j :integer) ;
      begin
         x := t [i] ; t [i] := t [j] ; t [j] := x
      end { Echanger } ;

   procedure TriRapide (m, n : integer) ;
      var k, v : integer ;

      function Partition (y, z : integer) : integer ;
         var i, j : integer ;
            begin ... t ... v ... Echanger (i,j) ; ...
            end { Partition } ;

      begin ... end { TriRapide } ;

   begin ... end { Trier }.
```

Noms non locaux avec imbrication de fonctions

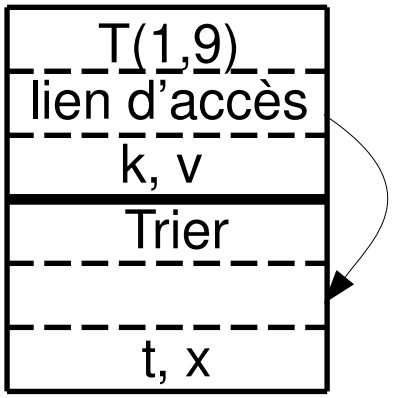
L'implantation utilise la *profondeur d'imbrication* d'une fonction, définie de manière similaire à celle sur les blocs

On ajoute aux enregistrements d'activation un *lien d'accès* :
- si la procédure P est immédiatement imbriquée dans T dans le source, le lien d'accès d'un enregistrement d'activation de P référence le lien d'accès de l'enregistrement d'activation le plus récent de T.

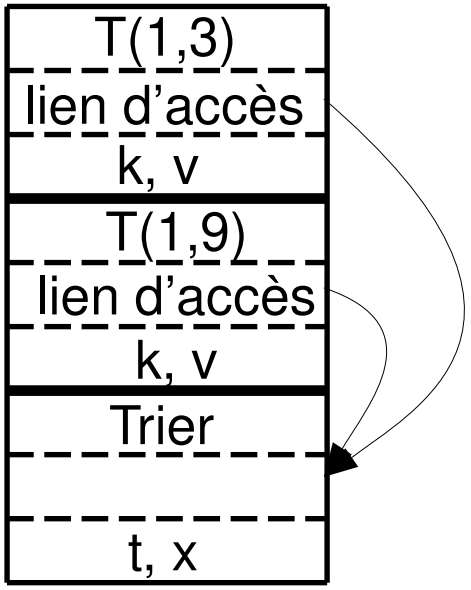
Mise en place des liens d'accès (P de profondeur *np* appelle X de profondeur *nx*) :
- *np* < *nx* : X est nécessairement déclarée dans P.
  - Le lien d'accès de l'enregistrement de X référence celui de P
- sinon : X et P ont mêmes fonctions englobantes jusqu'au niveau *nx* – 1.
  - Il faut donc remonter les niveaux d'imbrication à partir de l'enregistrement de P
    - en suivant *np* – *nx* + 1 liens.
  - Cela donne la référence du lien d'accès de X.

Accès aux noms non locaux : (P de profondeur *np* référence T de profondeur *nt* ≤ *np*) :
- on suit *np* – *nt* liens d'accès à partir de l'enregistrement d'activation de P (actif) pour obtenir l'enregistrement de la procédure où T est locale
- puis on sait où se situe l'emplacement associé relativement au lien d'accès.

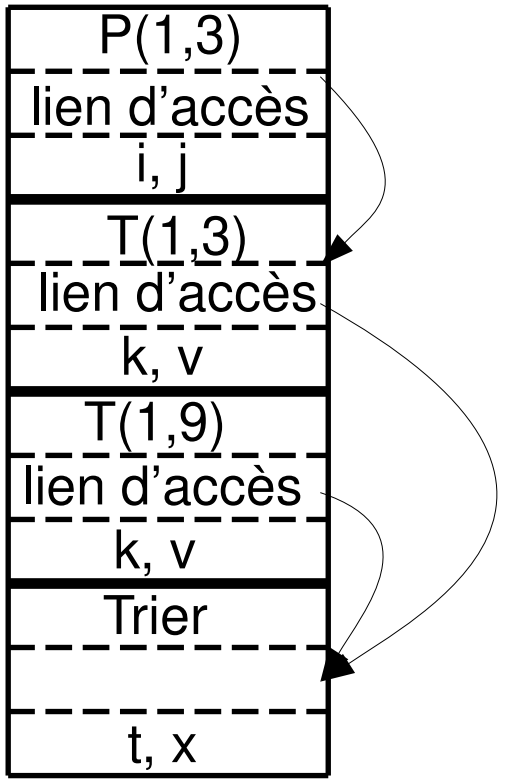
Noms non locaux avec imbrication de fonctions



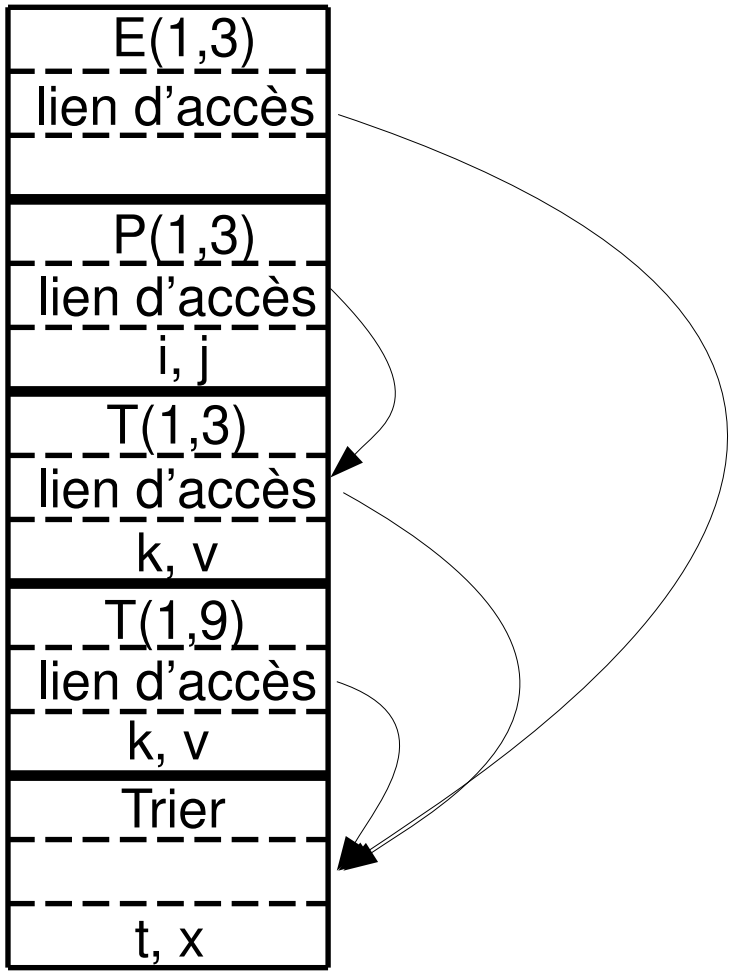
1



2



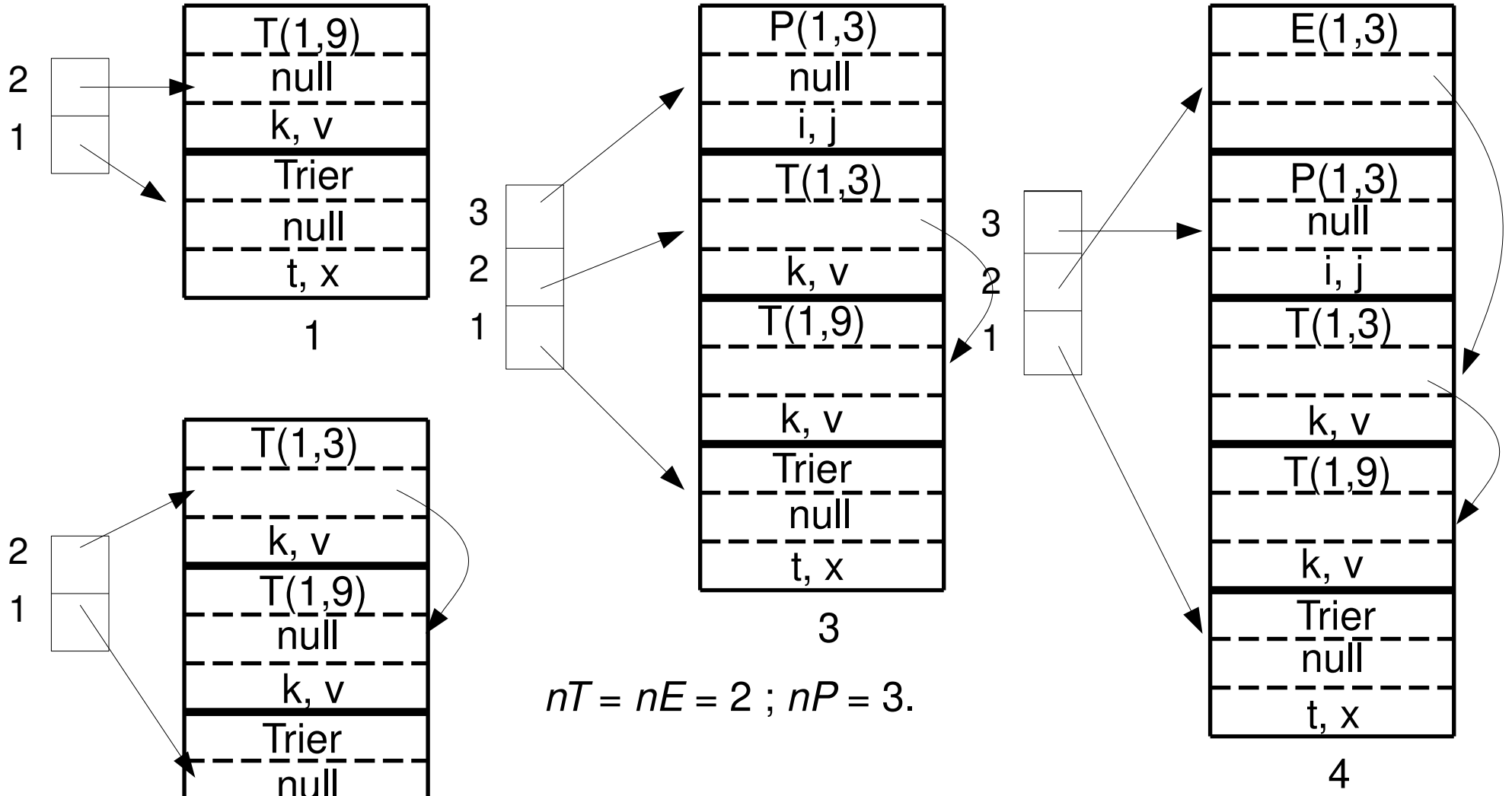
3



4

$nT = nE = 2$; $nP = 3$.

Noms non locaux avec imbrication de fonctions



$nT = nE = 2$; $nP = 3$.

Une implantation par accesseur permet d'éviter de suivre les liens d'accès

Noms non locaux avec imbrication de fonctions

Cas du passage d'une fonction en paramètre :

```
program Param (input, output) ;

   procedure b (function h (n : integer) : integer) ;
      begin writeln (h (2)) end { b } ;

   procedure c ;
      var m : integer ;

      function f (n : integer) : integer ;
           begin f := m + n end { f } ;

      begin m := 0 ; b (f) end { c } ;

   begin c end.
```

Lorsque c appelle b, il fournit le paramètre f ; mais quand b devient active, f est normalement hors de portée de b. Donc c fournit le lien d’accès de f, qui est dans sa portée.

Noms non locaux avec imbrication de fonctions : portée dynamique

```
program Dynamique (input, output)
    var r : real ;

    procedure Montrer :
        begin write r end ;

    procedure A :
        var r : real ;
        begin r:=0.125 ; Montrer end ;

    begin
        r:= 0.25 ;
        Montrer ; A ;
    end.
```

Affichage : 0.25 0.125

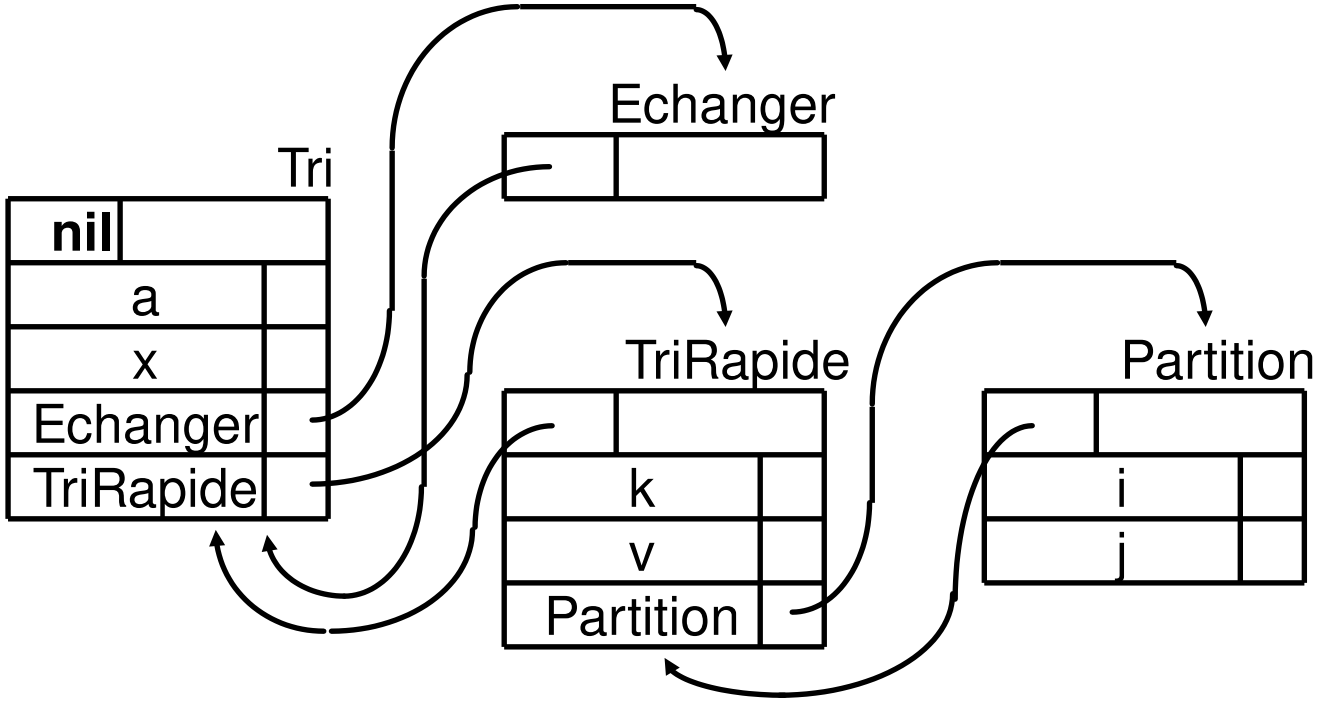
Plutôt que d'utiliser les liens d'accès, on utilise les liens de contrôle.

Seconde implantation : similaire à l'utilisation des accesseurs

Rappel : sans portée dynamique, l'exécution du même programme affiche 0.25 0.25

Table des symboles et imbrication de fonctions : une technique

- Une table des symboles par fonctions
- Les tables des symboles sont chaînées
- Le chaînage reflète l'imbrication des fonctions



Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)

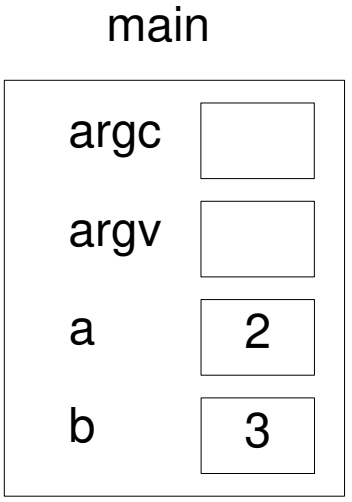
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

```
void f(int a, int *b) {
	a=a+*b ;
	++*b ;
}

int main(int argc, char *argv[]) {
	int a = 2 ; int b = 3 ;
	f(1+a, &a) ;
	return EXIT_SUCCESS ;
}
```

Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

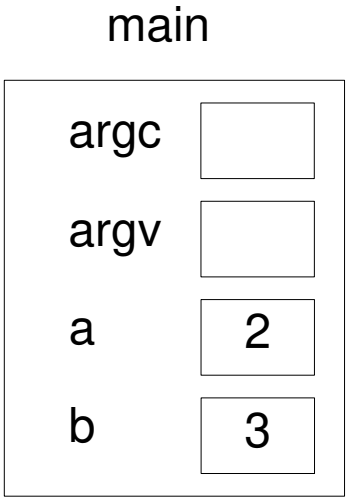
```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```



Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```

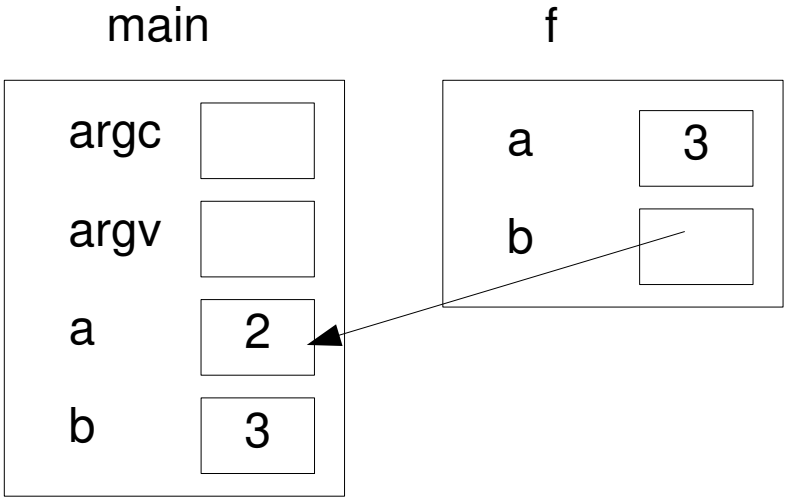


Évaluation des arguments
- le premier vaut 3
- le second pointe sur l'espace de a dans l'enregistrement d'activation de main

Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

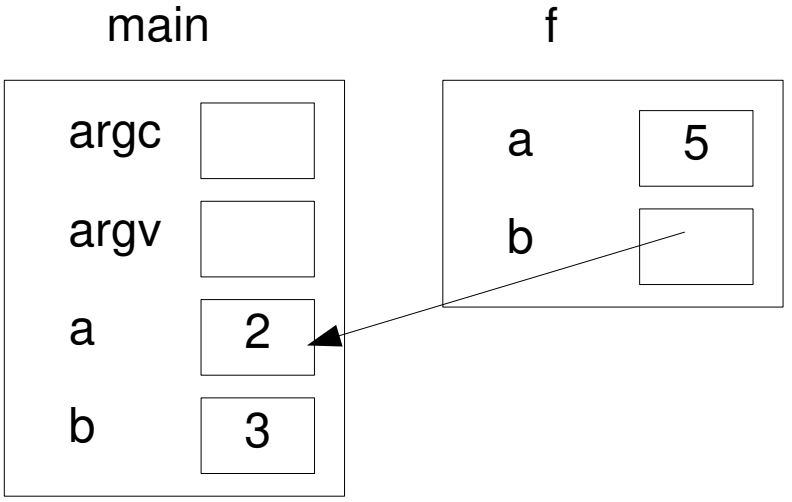
```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```



Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

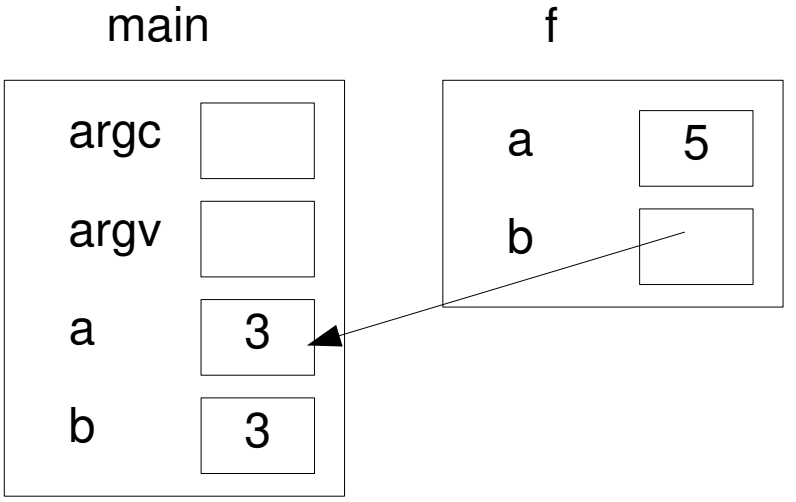
```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```



Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

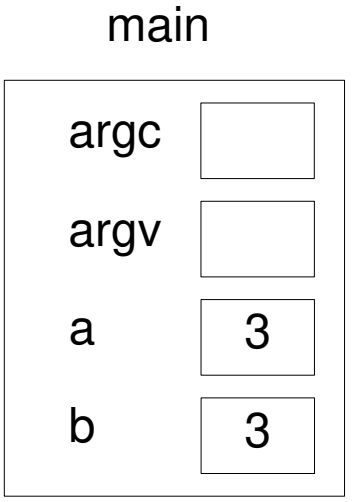
```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```



Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

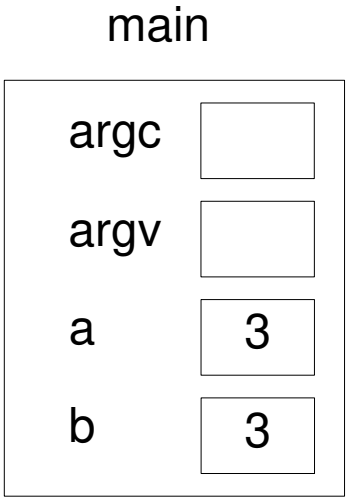
```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```



Passage des arguments : par valeur

C'est le plus courant (C, C++, Java, Pascal, ocaml, etc. C'est le seul possible en C et en Java)
- Chaque argument est évalué avant l'appel de fonction
- Sa valeur est mise dans l'enregistrement d'activation de l'appelé
- Toute modification par l'appelé est réalisée dans sur sa copie
- Il faut utiliser des noms non locaux ou des pointeurs pour que la fonction appelante puisse changer des valeurs à l'extérieur de son corps

```
void f(int a, int *b) {
        a=a+*b ;
        ++*b ;
}

int main(int argc, char *argv[]) {
        int a = 2 ; int b = 3 ;
        f(1+a, &a) ;
        return EXIT_SUCCESS ;
}
```



Passage des arguments : par référence

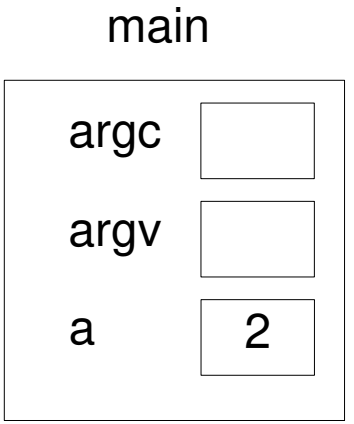
Courant également. Possible en C++, Pascal, …

Si l'argument a une *l-value*, c'est la *l-value* qui est passée en argument

Si l'argument n'en a pas

- un emplacement mémoire est créé
- l'argument est stocké dans cet emplacement
- c'est la *l-value* correspondante qui est passée en argument

```
void f(int a, int &b) {
  ++a;
  b += 2;
}

int main(int argc, char *argv[]) {
  int a = 2;
  f(a, a);
  return 0;
}
```



Passage des arguments : par référence

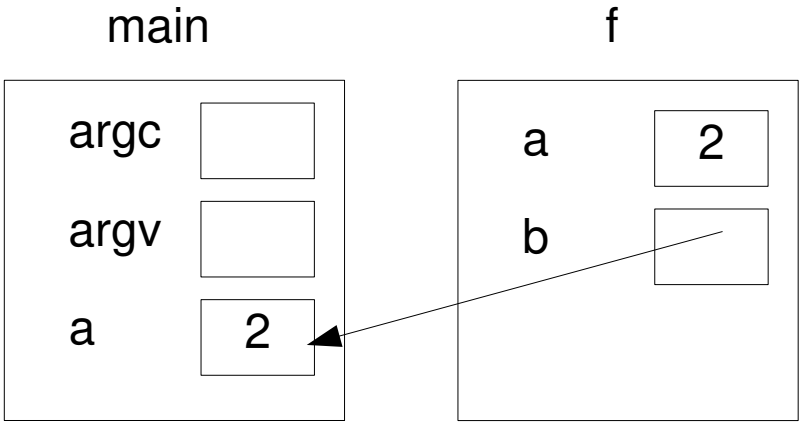
Courant également. Possible en C++, Pascal, …

Si l'argument a une *l-value*, c'est la *l-value* qui est passée en argument

Si l'argument n'en a pas

- un emplacement mémoire est créé
- l'argument est stocké dans cet emplacement
- c'est la *l-value* correspondante qui est passée en argument

```
void f(int a, int &b) {
  ++a;
  b += 2;
}

int main(int argc, char *argv[]) {
  int a = 2;
  f(a, a);
  return 0;
}
```



Passage des arguments : par référence

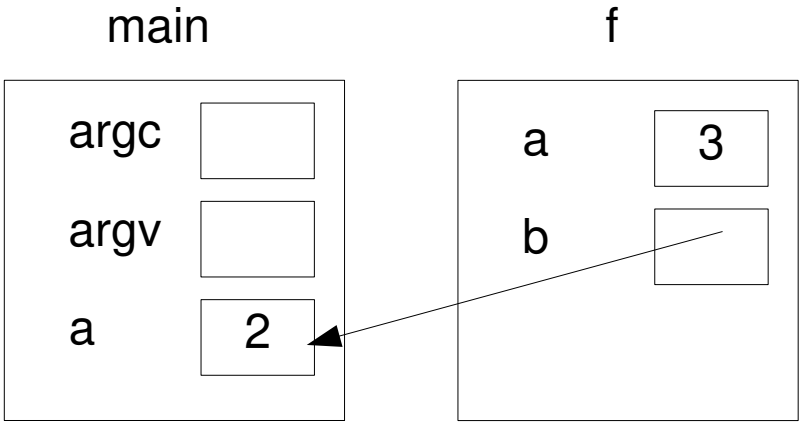
Courant également. Possible en C++, Pascal, …

Si l'argument a une *l-value*, c'est la *l-value* qui est passée en argument

Si l'argument n'en a pas

- un emplacement mémoire est créé
- l'argument est stocké dans cet emplacement
- c'est la *l-value* correspondante qui est passée en argument

```
void f(int a, int &b) {
  ++a;
  b += 2;
}

int main(int argc, char *argv[]) {
  int a = 2;
  f(a, a);
  return 0;
}
```



Passage des arguments : par référence

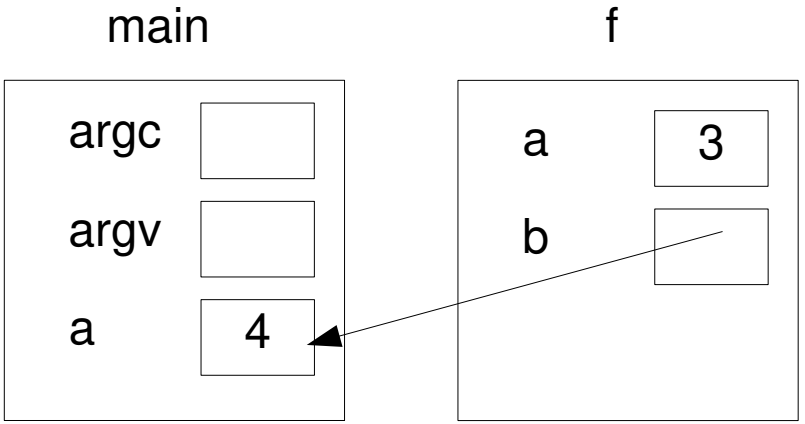
Courant également. Possible en C++, Pascal, …

Si l'argument a une *l-value*, c'est la *l-value* qui est passée en argument

Si l'argument n'en a pas

- un emplacement mémoire est créé
- l'argument est stocké dans cet emplacement
- c'est la *l-value* correspondante qui est passée en argument

```
void f(int a, int &b) {
  ++a;
  b += 2;
}

int main(int argc, char *argv[]) {
  int a = 2;
  f(a, a);
  return 0;
}
```



Passage des arguments : par référence

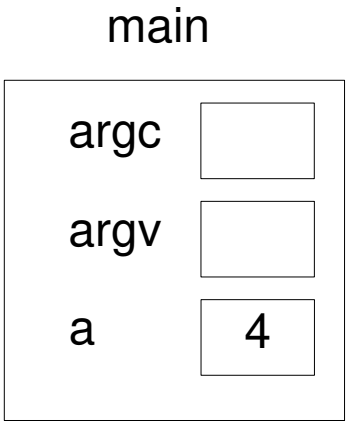
Courant également. Possible en C++, Pascal, …

Si l'argument a une *l-value*, c'est la *l-value* qui est passée en argument

Si l'argument n'en a pas

- un emplacement mémoire est créé
- l'argument est stocké dans cet emplacement
- c'est la *l-value* correspondante qui est passée en argument

```
void f(int a, int &b) {
  ++a;
  b += 2;
}

int main(int argc, char *argv[]) {
  int a = 2;
  f(a, a);
  return 0;
}
```



Passage des arguments : par référence

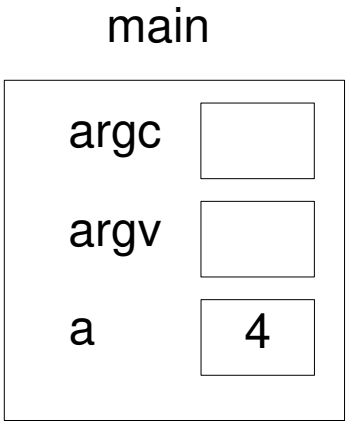
Courant également. Possible en C++, Pascal, …

Si l'argument a une *l-value*, c'est la *l-value* qui est passée en argument

Si l'argument n'en a pas

- un emplacement mémoire est créé
- l'argument est stocké dans cet emplacement
- c'est la *l-value* correspondante qui est passée en argument

```
void f(int a, int &b) {
  ++a;
  b += 2;
}

int main(int argc, char *argv[]) {
  int a = 2;
  f(a, a);
  return 0;
}
```



Passage des arguments : par copie/restauration

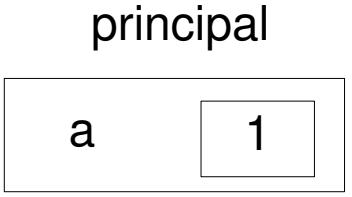
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Passage des arguments : par copie/restauration

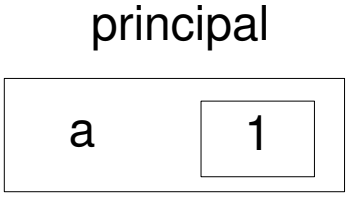
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Passage des arguments : par copie/restauration

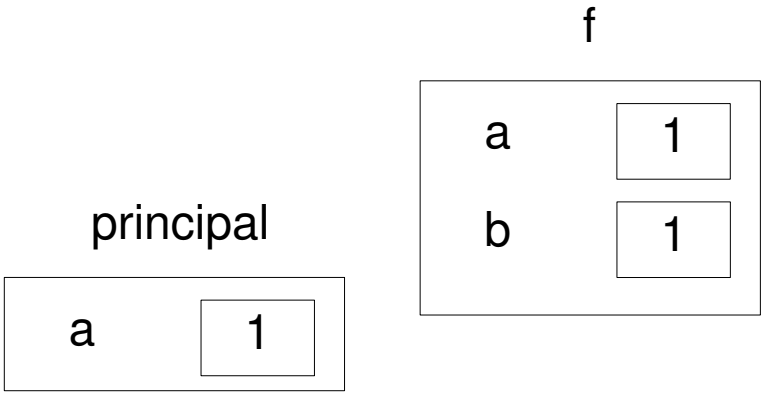
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```

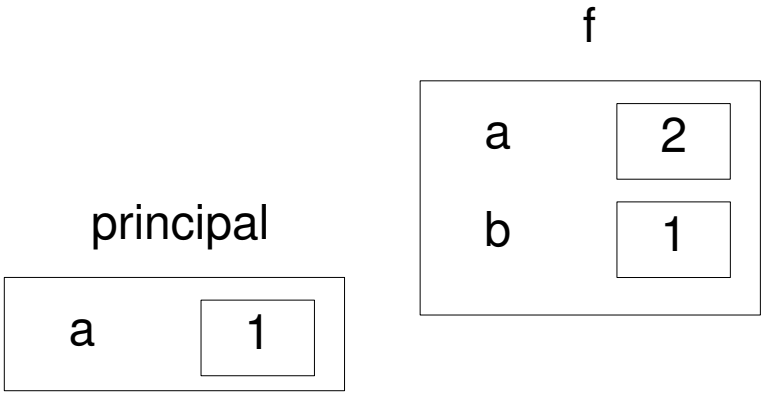


Passage des arguments : par copie/restauration

Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

- Les arguments sont évalués
- Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)
- Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```

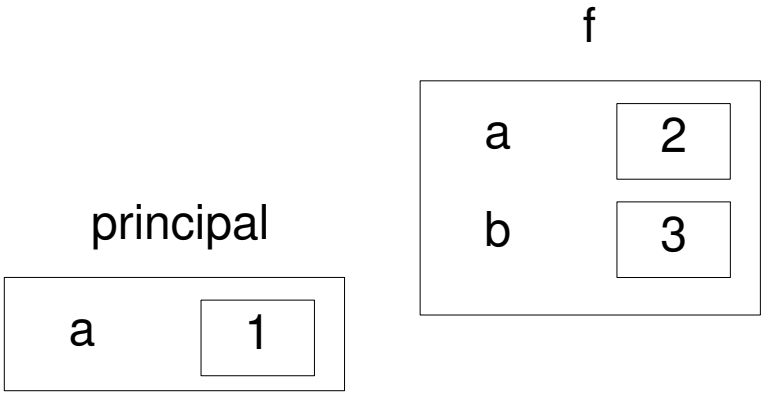


Passage des arguments : par copie/restauration

Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

- Les arguments sont évalués
- Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)
- Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```

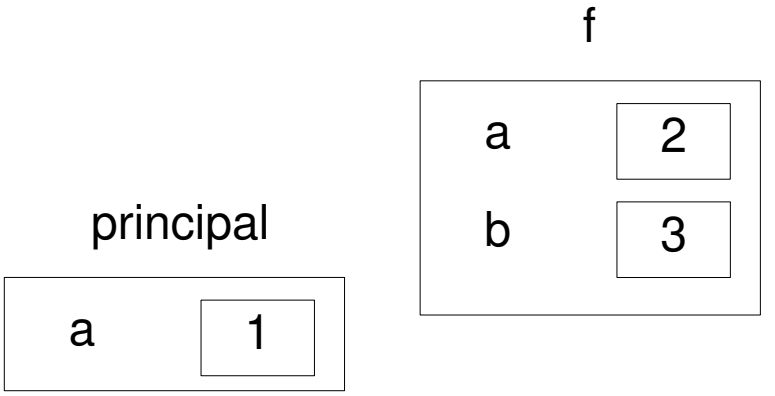


Passage des arguments : par copie/restauration

Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

- Les arguments sont évalués
- Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)
- Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Passage des arguments : par copie/restauration

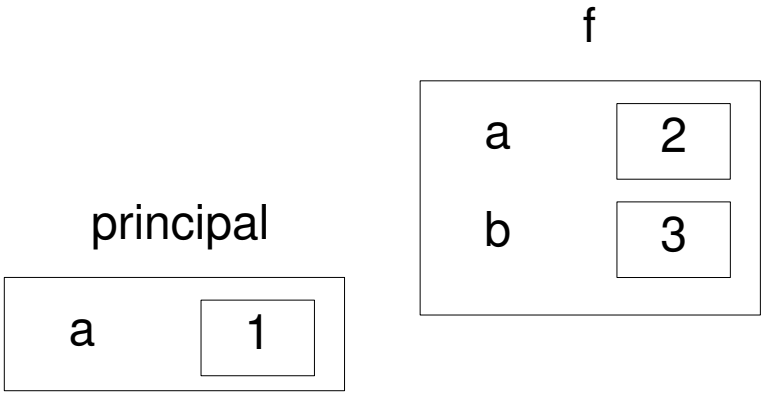
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Au retour de *f* les arguments sont recopiés dans leur endroit de provenance de l'enregistrement d'activation

Passage des arguments : par copie/restauration

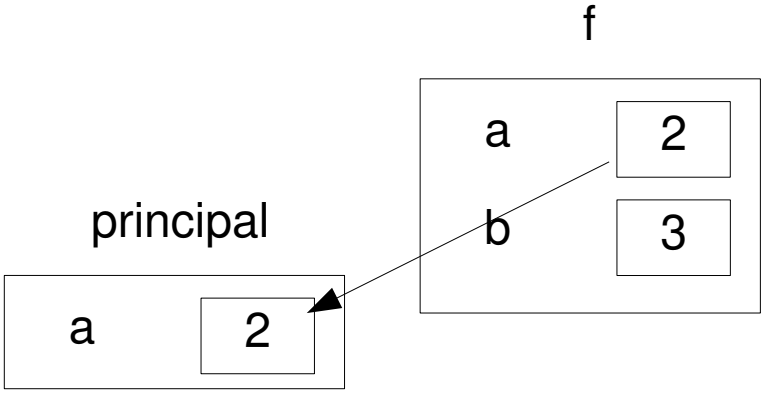
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Au retour de *f* les arguments sont recopiés dans leur endroit de provenance de l'enregistrement d'activation

Passage des arguments : par copie/restauration

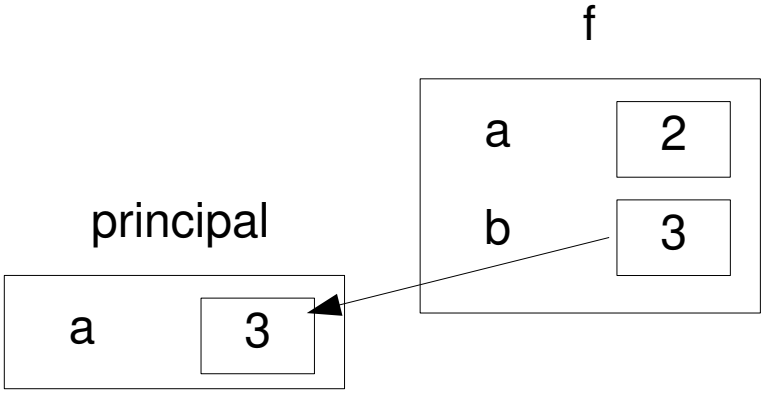
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Au retour de *f* les arguments sont recopiés dans leur endroit de provenance de l'enregistrement d'activation

Passage des arguments : par copie/restauration

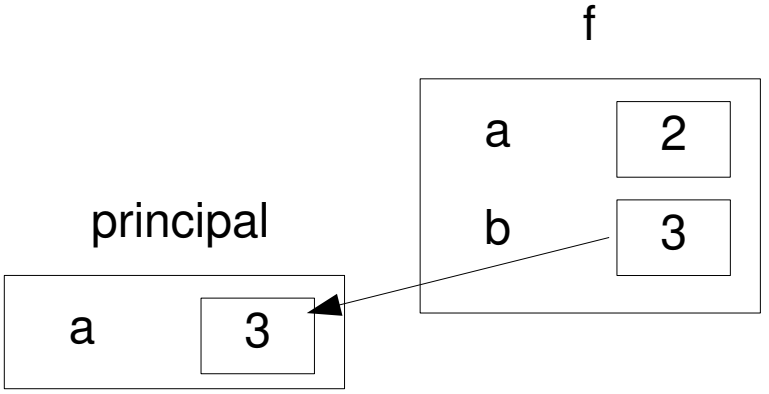
Utilisé en programmation concurrente pour éviter des problèmes liés au passage par référence.

Les arguments sont évalués

Leurs valeurs sont copiés dans l'enregistrement d'activation de l'appelé (passage par valeur)

Au retour de l'appelé, ces valeurs de l'enregistrement d'activation de l'appelé sont recopiées dans l'enregistrement d'activation de l'appelant

```
fonction f(entier a, entier b) {
    a:=a+1 ;
    b:=b+2 ;
}

principal {
    entier a=1 ;
    f(a, a) ;
}
```



Attention : l'ordre dépends de l'implantation !

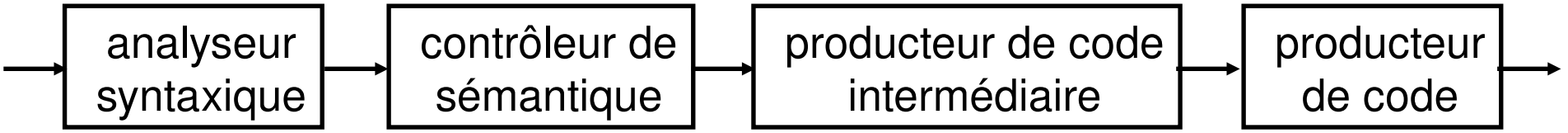
Au retour de *f* les arguments sont recopiés dans leur endroit de provenance de l'enregistrement d'activation

Passage des arguments : par nom

Les arguments ne sont pas évalués avant d'être passés à l'appelant.
Ils sont remplacés dans le corps de la fonction appelée.
Équivalent des macros du C.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
  x = min(a, b);           ==>  x = ((a) < (b) ? (a) : (b));
  y = min(1, 2);           ==>  y = ((1) < (2) ? (1) : (2));
  z = min(a + 28, *p);     ==>  z = ((a + 28) < (*p) ? (a + 28) : (*p));
```

Code intermédiaire



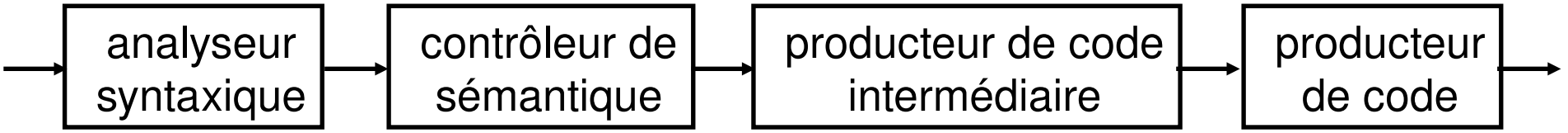
Sépare les parties frontales et finales de la compilation
Facilite la portabilité, l'optimisation
Les programmes sont représentés par des arbres abstraits ou des graphes acycliques

Génération de code proche de l'assembleur, machine indépendant
- code à trois adresses

Le code produit doit
- préserver la sémantique du code d'entrée
- être efficace
  - temps d'exécution
  - compacité du code produit
  - consommation d'énergie

Le temps de production doit rester raisonnable
- les problèmes d'optimalité du code sont difficiles (temps de calculs longs)
- des heuristiques efficaces peuvent être utilisées en remplacement

Code intermédiaire



Sépare les parties frontales et finales de la compilation
Facilite la portabilité, l'optimisation
Les programmes sont représentés par des arbres abstraits ou des graphes acycliques

Code à trois adresses :
- adapté aux structures de contrôles imbriquées et aux expressions algébriques
- proche d'un langage d'assemblage

Instructions :

x := y *op* z **x := *op* z**

- x, y et z sont des constantes ou des noms
- *op* un opérateur quelconque.

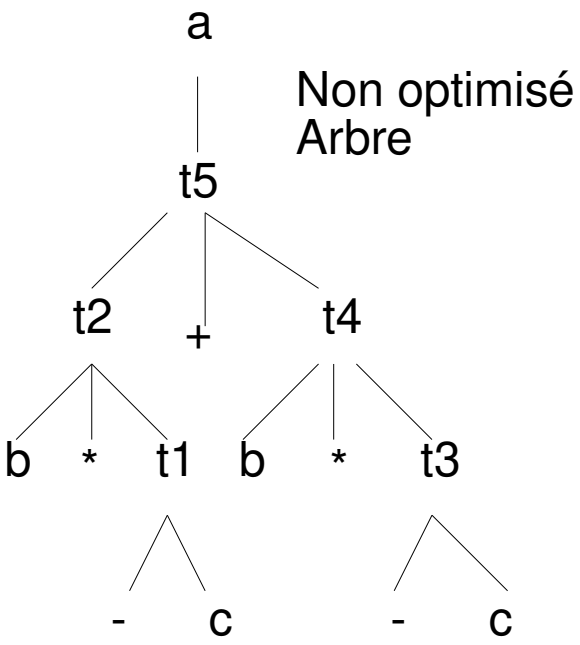
Code à 3 adresses

```
a := b * - c + b * - c
```

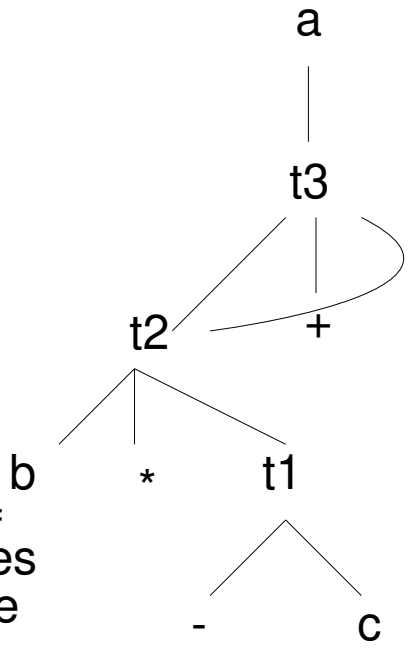
```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
 a := t5
```

ou

```
t1 := - c
t2 := b * t1
t3 := t2 + t2
 a := t3
```



Non optimisé
Arbre



Optimisé
Graphe acyclique

Temporaires =
non-feuilles
non-racine

Code à 3 adresses

- Affectation `x := y op z`
- Affectation `x := op y`
- Copie `x := y`
- Branchement inconditionnel `goto L`
- Branchement conditionnel `if x op y goto L`
- Affectation avec indices `x := y [i]` `x [i] := y`
- Pointeurs et adresses `x := & y` `x := * y` `* x := y`
- Appels de procédure

Génération de code à 3 adresses
Expressions

Dirigée par la syntaxe

|| : concaténation NouveauTemp : création nouveau temporaire

Prod : production d'une instruction

Règle	**Action**
$I \rightarrow$ **`id :=`** E	*I.code* := *E .code* || *Prod* (**`id`**.*place* '**`:=`**' *E.place*)
$E \rightarrow E + E$	*E.place* := *NouveauTemp()* ; *E.code* := E_1 *.code* || E_2 *.code* || *Prod* (*E . place* '**`:=`**' E_1 *. place* '**`+`**' E_2 *. place*)
$E \rightarrow E * E$	*E.place* := *NouveauTemp()* ; *E.code* := E_1 *.code* || E_2 *.code* || *Prod* (*E . place* '**`:=`**' E_1 *. place* '**`*`**' E_2 *. place*)
$E \rightarrow - E$	*E.place* := *NouveauTemp()* ; *E.code* := E_1 *.code* || *Prod* (*E . place* '**`:=`**' '**`MoinsUnaire`**' E_1 *. place*)
$E \rightarrow (E)$	*E.place* := E_1 *.place* ; *E.code* := E_1 *.code*
$E \rightarrow$ **`id`**	*E.place* := **`id`**.*place* ; *E.code* :=' '

Génération de code à 3 adresses Instructions

Règle	**Action**
$I \rightarrow$ **`while`** E **`do`** I	*I.start* := *NouveauLabel()* ; *I.end* := *NouveauLabel()* ; *I.code* := *Prod* (*I. start* '**`:`**') || *E .code* || *Prod* ('**`if`**'*E . place* '**`=`**' '**`0`**' '**`goto`**' *I.end*) || I_1 *. code* || *Prod* ('**`goto`**' *I.start*) || *Prod* (*I.end* '**`:`**')

Convention pour les booléens:

- 0 = faux
- non 0 = vrai

comme en C

Génération de code à 3 adresses
Déclarations

Ici, la technique d'une table des symboles par fonction avec chaînage des tables est utilisée
La création des tables des symboles est réalisée pendant l'analyse syntaxique
une pile PtrTbl sert à gérer les tables des symboles au fur et à mesure de l'analyse
les variables locales sont stockées dans un espace mémoire propre à chaque fonction
une pile depl contient les tailles de ces espaces pour stocker les variables locales aux fonction
la production $D \rightarrow$ **`proc id`** ; *N D* **`;`** *I* représente la déclaration d'une procédure interne, ses déclarations locales et ses instructions (version simplifiée)

$P \rightarrow M\ D$ { *AjouterTaille* (*Sommet* (*PtrTbl*), *Sommet* (*depl*)) ;
Depiler (*PtrTbl*) ; *Depiler* (*depl*) }

$M \rightarrow \varepsilon$ { *t* := *CreerTable* (**nil**) ; *Empiler* (*t*, *PtrTbl*) ; *Empiler* (0, *depl*) }

$D \rightarrow D$ **`;`** D

$D \rightarrow$ **`proc id`** ; $N\ D$ **`;`** I
{ *t* := *Sommet* (*PtrTbl*) ; *AjouterTaille* (*t*, *Sommet* (*depl*)) ;
Depiler (*PtrTbl*) ; *Depiler* (*depl*) ;
EntrerProcedure (*Sommet* (*PtrTbl*), **`id`**.*nom* , *t*) }

$D \rightarrow$ **`id :`** T { *Entrer* (*Sommet* (*PtrTbl*), **`id`**.*nom* , *T.type*, *Sommet* (*depl*)) ;
Sommet (*depl*) := *Sommet* (*depl*) + *T.taille* }

$N \rightarrow \varepsilon$ {*t* := *CreerTable* (*Sommet* (*PtrTbl*)) ;
Empiler (*t*, *PtrTbl*) ; *Empiler* (0, *depl*) }

Génération de code à 3 adresses
Déclarations

$P \rightarrow M\ D$ { *AjouterTaille* (*Sommet* (*PtrTbl*), *Sommet* (*depl*)) ;
Depiler (*PtrTbl*) ; *Depiler* (*depl*) }

$M \rightarrow \varepsilon$ { *t* := *CreerTable* (**nil**) ; *Empiler* (*t*, *PtrTbl*) ; *Empiler* (0, *depl*) }

$D \rightarrow D$ **;** D

$D \rightarrow$ **proc id** ; $N\ D$ **;** I
{ *t* := *Sommet* (*PtrTbl*) ; *AjouterTaille* (*t*, *Sommet* (*depl*)) ;
Depiler (*PtrTbl*) ; *Depiler* (*depl*) ;
EntrerProcedure (*Sommet* (*PtrTbl*), **id**.*nom* , *t*) }

$D \rightarrow$ **id :** T { *Entrer* (*Sommet* (*PtrTbl*), **id**.*nom* , *T.type*, *Sommet* (*depl*)) ;
Sommet (*depl*) := *Sommet* (*depl*) + *T.taille* }

$N \rightarrow \varepsilon$ {*t* := *CreerTable* (*Sommet* (*PtrTbl*)) ;
Empiler (*t*, *PtrTbl*) ; *Empiler* (0, *depl*) }

CreerTable(*x*) crée une nouvelle table initialisée par un pointeur vers *x*.

Entrer(*table*, *nom*, *type*, *place*) et *EntrerProcedure*(*table*, *nom*, *table_nom*) gèrent les insertions dans la table des symboles.

AjouterTaille (*table*, *taille*) stocke dans la table sa taille totale.

Génération de code à 3 adresses
Affectations

Version simplifiée :

I → **id :=** *E*
{ *p* := *Rechercher* (**id**.*nom*) ;
if *p* **<> nil then**
I.code := *E .code* || *Prod* (**id**.*place* '**:=**' *E.place*)
else *erreur* }

E → *E* +*E*
{ *E.place* := *NouvTemp()* ; *E.code* := E_1 *.code* || E_2 *.code* ||
Prod (*E . place* '**:=**' E_1 *. place* '**+**' E_2 *. place*) }

E → (*E*)
{ *E.place* := E_1 *.place* ; *E.code* := E_1 *.code* }

E → **id**
{ *p* := *Rechercher* (**id**.*nom*) ;
if *p* **<> nil then**
begin *E.place* := **id**.*place* ; *E.code* :=' ' **end**
else *erreur* }

Il faudrait aussi :
- vérifier le typage des membres gauche et droit de l'affectation attribut synthétisé « type »
- pouvoir remplacer **id** par une *l-value*

Génération de code à 3 adresses
Tableaux

Exemple du C (ou de Java)
int t[N_1]...[N_k] : t est un tableau de N_1 tableaux de … de N_K int

On pose
- w_k = taille d'un int
- w_{k-1} = taille d'un tableau de N_k int = N_k*w_k
- w_{k-2} = taille d'un tableau de N_{k-1} tableaux de N_k int = N_{k-1}*w_{k-1}
- …
- w_1 = taille d'un tableau de N_2 tableaux de … de N_k int = N_2*w_2
- base = adresse du premier élément du tableau

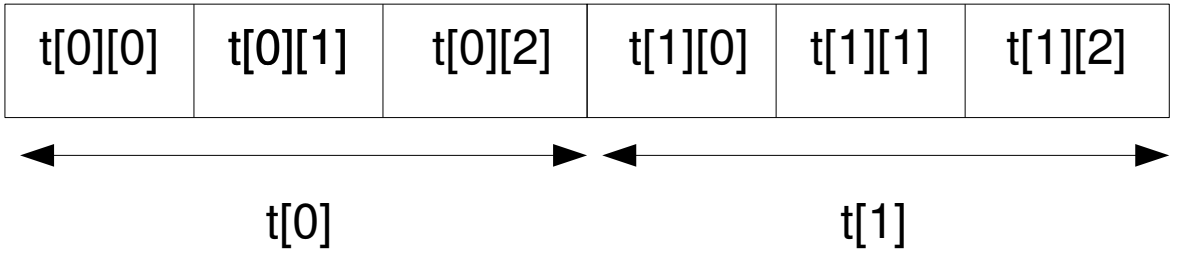
Les w_i peuvent être calculés par le compilateur quand l'analyse de la déclaration du tableau est terminée.

Hypothèse : les éléments du tableau sont rangés en mémoire
- de manière contiguë
- par ordre croissant des indices des éléments

Adresse de t[j_1]...[j_k] = base + j_1*w_1+...+j_k*w_k

Génération de code à 3 adresses
Tableaux

int t[2][3] : t est un tableau de 2 tableaux de 3 entiers



L'analyse de la gauche vers la droite du type du tableau fourni des sous-tableaux dont les dimensions vont en décroissant

Dans la grammaire, le non terminal L représente les expressions extraites des tableaux

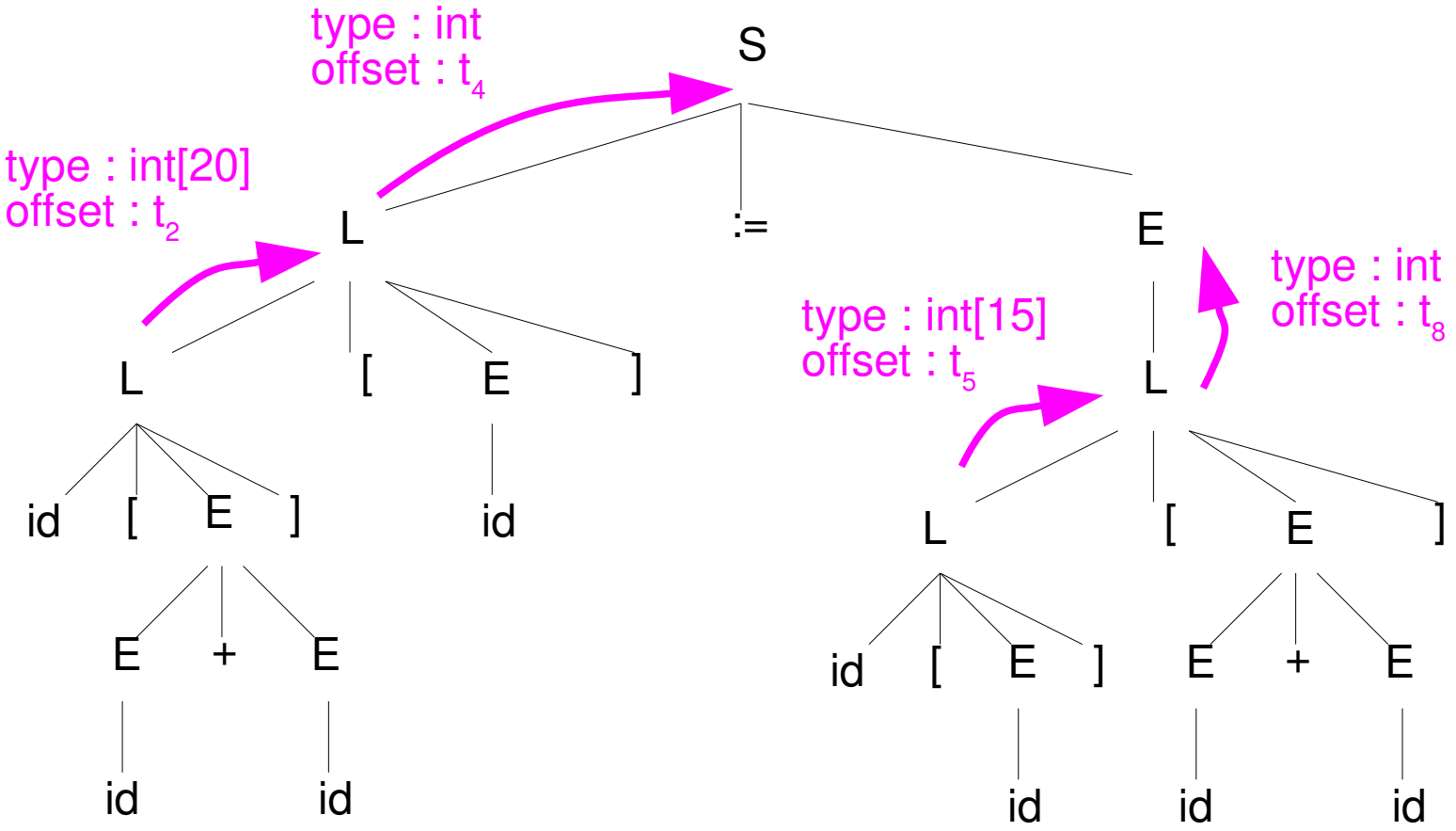
Attributs synthétisés :
- L.array : pointeur sur la table des symboles pour ce tableau
- L.type : type du sous-tableau engendré par L
- L.offset : décalage par rapport à l'adresse de base du tableau

Génération de code à 3 adresses
Tableaux

S → **id** := E { Comme précédemment }
S → L := E { *S.code := E_1.code || Prod(L.array.base '[' L.offset '] :=' E_1.place)* }
E → E+E { Comme précédemment }
E → **id** { Comme précédemment }
E → L { *E.place* := *NouveauTemp()*;
E.code := L_1 *.code || Prod(E.place ':=' L.array.base '[L.offset '])* }
L → **id** [E] { L.array := *Rechercher* (**id**.*nom*) ;
if *L.array* ***<> nil then***
L.offset := NouveauTemp();
L.type:= L.array.elementType
*L.code:= E.code || Prod(L.offset ':=' E.place*width(L.array.elementType))*
else *erreur* }
L → L[E] { L.array:= L_1.array ; L.type:= L_1.type.elementType ; t:= NouveauTemp() ;
L.offset:= NouveauTemp() ;
L.code:= E.code || Prod(t ':=' E.place '*'width(L.type)) || Prod(L.offset ':=' L_1.offset '+' t) }

Génération de code à 3 adresses
Tableaux

Exemple : code produit par l'analyse de t[a+b][c]:=u[a][b+c] avec int t[10][20] et int u[5][15]



t_1:=a+b
t_2:=t_1*width(int[20])
t_3:=c*width(int)
t_4:=t_2+t_3

t_5:=a*width(int[15])
t_6:=b+c
t_7:=t_6*width(int)
t_8:=t_5+t_7

t_9:=u[t_8]

t[t_4]:=t_9

Génération de code à 3 adresses
Tableaux

Si la syntaxe des tableaux est **`T [`**$\mathbf{n_1}$**`,…,`**$\mathbf{n_k}$**`]`**

la grammaire naturelle est

G → **`id [`** *Eliste* **`]`** | **`id`**
Eliste → *Eliste* **`,`** *E* | *E*

Dans ce cas les dimensions du tableau doivent être passées comme attributs hérités de *Eliste*. On pourra plutôt utiliser la grammaire suivante, moins naturelle mais qui ne nécessite pas d'attributs hérités:

G → *Eliste* **`]`** | **`id`**
Eliste → *Eliste* **`,`** *E* | **`id [`** *E*

Optimisation de la création des temporaires

Idée sur la réduction de $E \rightarrow E + E$ pour éviter de créer trop de temporaires

On utilise un compteur de temporaires *c*

- Si les deux *E* en partie droite ont comme attribut *E.place* des noms explicites, on utilise un nouveau temporaire t_c comme *E.place* de la réduction.
- Si l'un des deux a comme attribut *E.place* un nom et l'autre un temporaire t_i, celui-ci peut être réutilisé comme *E.place* de la réduction (c est inchangé).
- Si les deux ont comme attributs des temporaires, ce sont nécessairement t_i et t_{i+1}.
  On peut donc réutiliser t_i comme *E.place* de la réduction.
  Le futur nouveau temporaire sera donc t_{i+1} (c=i+1).

On ne peut évidemment réutiliser des temporaires que s'ils ont le même type, ce que sait le compilateur.

Génération de code à 3 adresses Fonctions

On ajoute au jeu d'instruction du code à 3 adresses :
- param t : ajouter de la valeur de la variable t à une liste de paramètres
- t = call *nomFonction*, nombreParamètres : appel de la fonction *nomFonction* à *nombreParamètres* paramètres, stockage du résultat dans *t*

Les codes à 3 adresses de chaque fonction sont générés séparément, et associés à la déclaration de chaque fonction dans la table des symboles

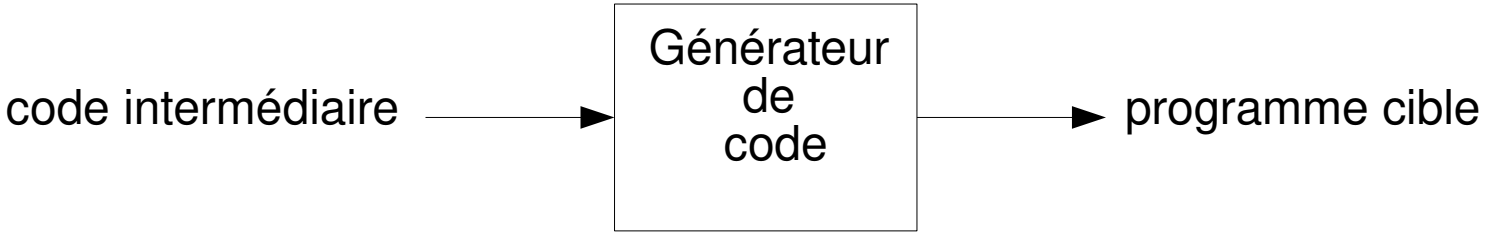
- L'instruction (*a* est un tableau d'entiers)

n = f(a[i])

est compilée en

t_1 = i*width(int)
t_2 = a[t_1]
param t_2
t_3 = call f, 1
n = t_3

Production de code cible



Exemple de code intermédiaire : code à 3 adresses

Hypothèses sur le code intermédiaire :
- les valeurs qui y apparaissent sont toutes manipulables par la machine cible
- les opérations de contrôle, conversions, etc. ont toutes eu lieu

Problématique :
- produire le code cible le plus efficace possible
  (temps d'exécution ? taille ? consommation d'énergie ?)
- très souvent, les problèmes d'optimisation qui apparaissent sont difficiles, voire NP-complets
- on utilise alors des heuristiques qui fonctionnent suffisamment bien
- plus la machine cible a un langage complet, plus il est facile de produire du code efficace

Production de code cible

Pour réaliser une traduction efficace il faut :

- placer chaque variable pertinemment
  - dans un registre de préférence
    (surtout pour les variables utilisées intensivement localement)
  - dans la mémoire sinon
- il faut donc une stratégie d'utilisation des registres
  - *allocation des registres* : choix de l'ensemble des variables qui vont résider
    dans les registres à un point donné du programme
  - *assignation des registres*: choix du registre dans lequel une variable va être stockée
    - l'assignation optimale est un problème NP-complet
- traduire les instructions à 3 adresses dans les meilleurs instructions de la machine cible
  - individuellement
  - il peut aussi être pertinent de regrouper les instructions à 3 adresses pour mieux les traduire
  - il peut aussi être pertinent de permuter des instructions pour avoir un code produit
    plus efficace
    - à nouveau, le choix de la meilleure permutation est un problème NP-complet
- placer les instructions et les données en mémoire (calcul effectif des adresses)
  - si le code cible est destinée à un programme d'assemblage, alors ce dernier peut le faire
    (les adresses restent alors des noms symboliques dans le programme traduit)

Découpage du flot d'instructions à 3 adresses : blocs de base

Un bloc de base est formé de la plus grande suite d'instructions possible
- sans possibilité d'arrêt
- sans branchement autre qu'à la fin de la séquence

Instruction de tête : première instruction d'un bloc de base

Algorithme : dans la suite des instructions, on repère
- les instructions de branchement ou d'arrêt
- leurs destinations

La première instruction est une instruction de tête
Toute instruction destination d'un branchement est une instruction de tête
Toute instruction suivant une instruction de branchement est une instruction de tête

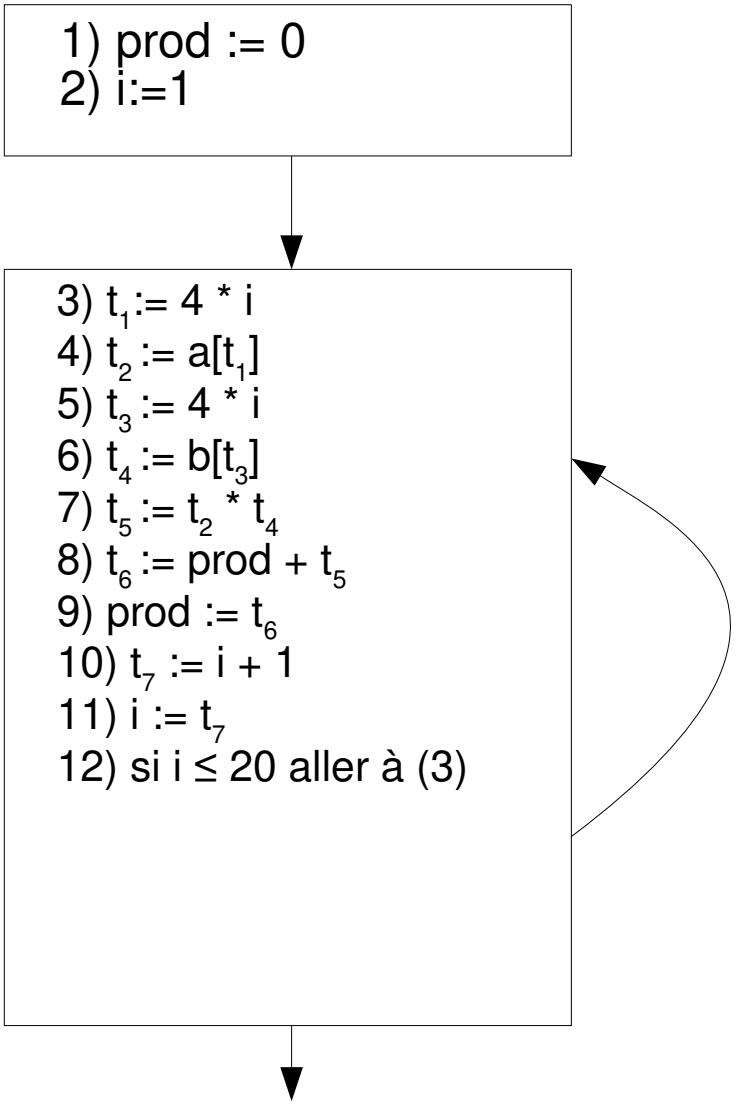
Les instructions de tête obtenues délimitent les blocs de base

L'organisation du code en blocs de base peut être représenté par un graphe
- les nœuds sont les blocs de base
- les arcs la relation entre les blocs de base donnée par le flot de contrôle (sauts, suite)
- un nœud initial : le premier bloc de base
- des nœuds finaux : ceux terminant le flot d'exécution

Découpage du flot d'instructions à 3 adresses : blocs de base

Exemple :

```
prod:= 0 ;
i:=1 ;
do
      prod:= prod+a[i]*b[i] ;
      i:=i+1 ;
while (i ≤ 20) ;
```



Noms actifs dans les blocs de base

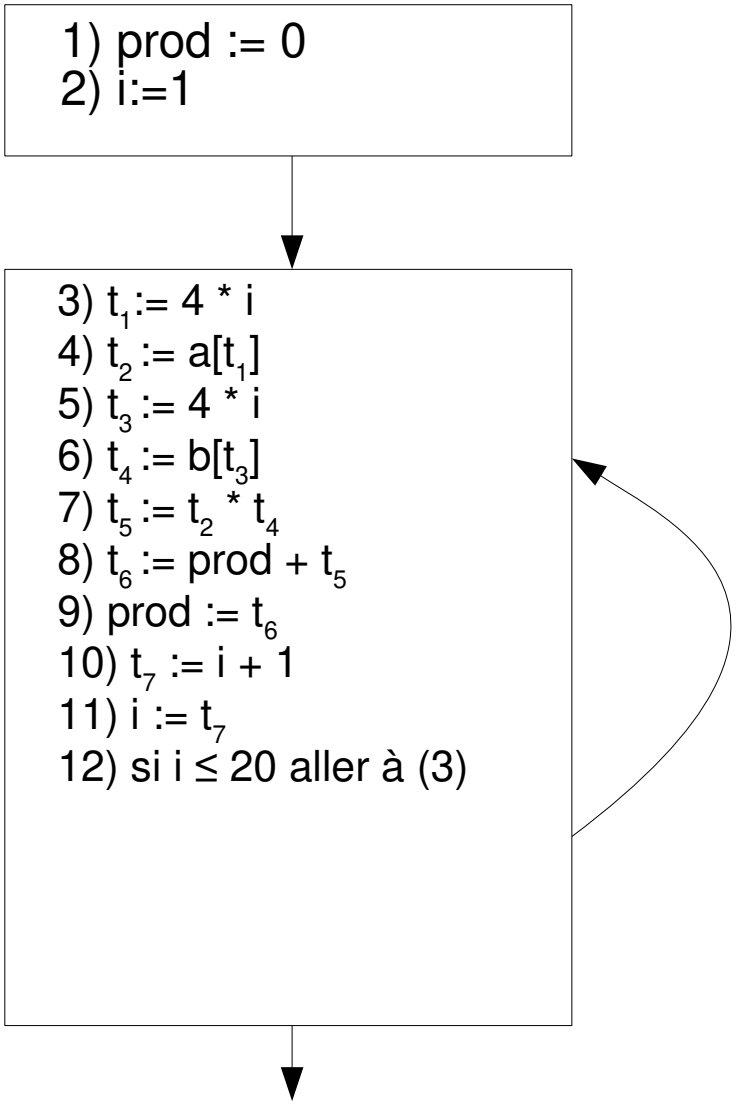
Pour chaque bloc de base, une analyse en partant de la fin :

Pour chaque symbole, on détermine
- l'instruction de sa dernière utilisation
- s'il est actif ou non (actif=utilisé plus tard)

Au départ,
- toutes les variables non temporaires du bloc sont marquées actives à la sortie du bloc
- si la production de code a 3 adresses a été faite pour, aucune variable temporaire n'est active en fin de bloc
- sinon, on peut supposer que le producteur de code a 3 adresses a marqué les variables temporaires qui sont actives en fin de bloc

Pour l'instruction *i*) *x*:=*y* op *z*
- associer à l'instruction *i* les informations de la table relatives à l'activation et à l'utilisation ultérieure de *x*, *y* et *z*
- dans la table des symboles
  - désactiver *x*
  - pas de prochaine utilisation de *x*
  - activer *y* et *z* : instruction de prochaine utilisation *i*

Pour les autres instructions, même principe



Noms actifs dans les blocs de base

Représentation graphique du résultat de l'application de l'algorithme

Exemple d'interprétation : après exécution de l'instruction 6, sont actives : t_2, t_4, i, prod, a , b

Exemple d'utilisation : après exécution de l'instruction 4, t_1 n'est plus nécessaire. Il est donc possible de réutiliser la variable t_1 plutôt que t_2 en membre gauche de l'affectation

3) $t_1 := 4 * i$

4) $t_2 := a[t_1]$

5) $t_3 := 4 * i$

6) $t_4 := b[t_3]$

7) $t_5 := t_2 * t_4$

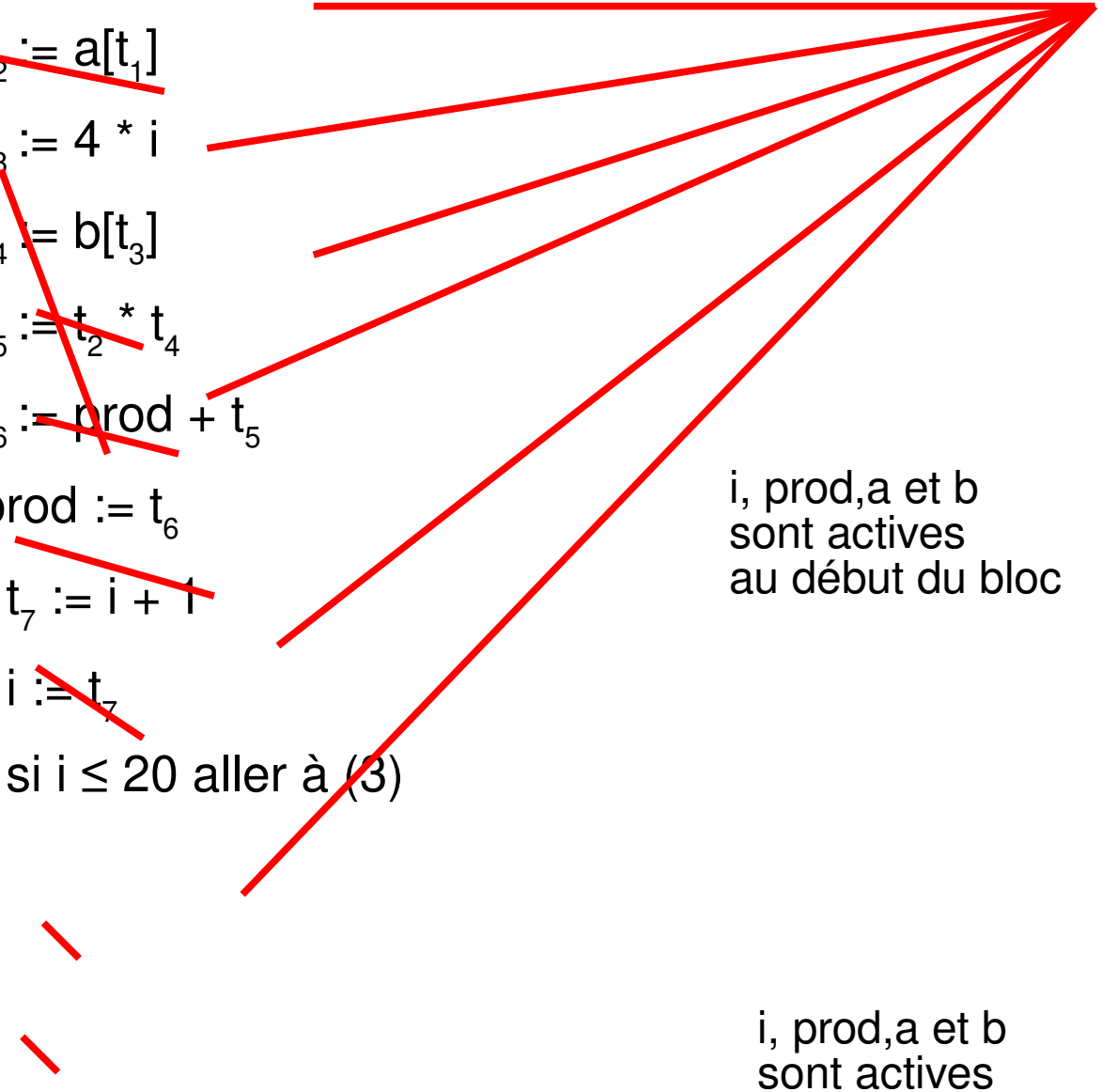
8) $t_6 := prod + t_5$

9) $prod := t_6$

10) $t_7 := i + 1$

11) $i := t_7$

12) si $i \leq 20$ aller à (3)



i, prod,a et b sont actives au début du bloc

i, prod,a et b sont actives en fin de bloc

Noms actifs dans les blocs de base

3) $t_1 := 4 * i$

4) $t_2 := a[t_1]$

5) $t_3 := 4 * i$

6) $t_4 := b[t_3]$

7) $t_5 := t_2 * t_4$

8) $t_6 := prod + t_5$

9) $prod := t_6$

10) $t_7 := i + 1$

11) $i := t_7$

12) si $i \leq 20$ aller à (3)

En parcourant les instructions de haut en bas et en utilisant le calcul précédent, on peut réduire de beaucoup le nombre de variables

3) $t_1 := 4 * i$

4) $t_1 := a[t_1]$

5) $t_2 := 4 * i$

6) $t_2 := b[t_2]$

7) $t_1 := t_1 * t_2$

8) $t_1 := prod + t_1$

9) $prod := t_1$

10) $t_1 := i + 1$

11) $i := t_1$

12) si $i \leq 20$ aller à (3)

Un générateur de code simple

Il faut utiliser les registres de manière appropriée :
- peut être imposé dans certaines situations (instructions) par la machine cible
- le plus possible pour éviter les temporaires
- certains registres sont spécialisés (pointeurs de pile, d'instruction, etc)

Hypothèses sur les instructions de la machine cible :
- LOAD registre, adresse
- STORE adresse, registre
- OP reg_1, reg_2, reg_3 pour chaque opération OP

Si ça n'est pas le cas, on remplace chacune de ces instructions par une combinaison des instructions de la machine cible

Structures de données :
- Pour chaque registre, la liste des variables qu'il contient (descripteur de registres)
- Pour chaque variable, la liste des emplacements mémoire et des registres où se trouve sa valeur (descripteur de données)
- Une fonction getRegs(instruction) qui sélectionne des registres à utiliser pour exécuter cette instruction
  - getRegs à accès aux descripteurs de registres et de données

Un générateur de code simple

Pour chaque instruction, on définit un schéma de traduction

- x=y+z
  - appeler getRegs(x=y+z) : on obtient trois registres R_x, R_y, R_z
  - si y n'est pas déjà dans R_y,
    - l'y charger par un LOAD R_y,y
    - mettre à jour les descripteurs de données et de registres
  - idem pour z
  - générer ADD R_x,R_y,R_z
  - mettre à jour les descripteurs de données et de registres
- x=y
  - appeler getRegs(x=y), qui ne doit retourner qu'un seul registre R_y
  - si y n'est pas dans R_y, charger y dans R_y par LOAD R_y,y
  - mettre à jour les descripteurs de données et de registre
    - x et y se trouvent dans R_y
- …
- fin de bloc
  - si une variable est active à la fin d'un bloc, il faut la placer dans l'espace mémoire qui lui est dédié

Un générateur de code simple : getRegs(x=y+z)

- si y est déjà dans un registre, le sélectionner pour le stockage de y
- sinon, s'il y a un registre vide, le sélectionner pour le stockage de y
- sinon on sélectionne un registre R contenant une variable v
  - si v est stocké quelque part ailleurs que dans R, alors on sélectionne R pour y
  - sinon si v est x et si x n'est ni y ni z, alors on sélectionne R pour y
  - sinon si v n'est pas réutilisée plus tard, on sélectionne R pour y
  - dans tous les autres cas on copie v dans son espace mémoire
- pareil pour z
- pour x, les règles sont quasiment identiques. Les différences sont :
  - comme on calcule une nouvelle valeur pour x, un registre contenant uniquement x est toujours un bon choix, même quand y ou z est x
  - si y n'est plus utilisé après l'instruction, et que R ne contient que y, alors $R_y=R_y+R_z$ est un choix pertinent

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

R_1	R_2	R_3

a	b	c	d	t_1	t_2	t_3
a	b	c	d			

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

R_1	R_2	R_3
a	t_1	

a	b	c	d	t_1	t_2	t_3
a,R_1	b	c	d	R_2		

LOAD R_1,a
LOAD R_2,b
SUB R_2,R_1,R_2

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

R_1	R_2	R_3
t_2	t_1	c

a	b	c	d	t_1	t_2	t_3
a	b	c,R_3	d	R_2	R_1	

LOAD R_1,a
LOAD R_2,b
SUB R_2,R_1,R_2
LOAD R_3,c
SUB R_1,R_1,R_3

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

R_1	R_2	R_3
t_2	t_1	t_3

a	b	c	d	t_1	t_2	t_3
a	b	c	d	R_2	R_1	R_3

LOAD R_1,a
LOAD R_2,b
SUB R_2,R_1,R_2
LOAD R_3,c
SUB R_1,R_1,R_3
ADD R_3,R_2,R_1

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

R_1	R_2	R_3
t_2	a,d	t_3

a	b	c	d	t_1	t_2	t_3
R_2	b	c	d,R_2		R_1	R_3

LOAD R_1,a
LOAD R_2,b
SUB R_2,R_1,R_2
LOAD R_3,c
SUB R_1,R_1,R_3
ADD R_3,R_2,R_1
LD R_2,d

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

R_1	R_2	R_3
d	a	t_3

a	b	c	d	t_1	t_2	t_3
R_2	b	c	R_1			R_3

LOAD R_1,a
LOAD R_2,b
SUB R_2,R_1,R_2
LOAD R_3,c
SUB R_1,R_1,R_3
ADD R_3,R_2,R_1
LD R_2,d
ADD R_1,R_3,R_1

Un générateur de code simple : exemple

t_1=a-b
t_2=a-c
t_3=t_1+t_2
a=d
d=t_3+t_2

SORTIE DE BLOC

R_1	R_2	R_3
d	a	t_3

a	b	c	d	t_1	t_2	t_3
R_2,a	b	c	R_1,d			R_3

LOAD R_1,a
LOAD R_2,b
SUB R_2,R_1,R_2
LOAD R_3,c
SUB R_1,R_1,R_3
ADD R_3,R_2,R_1
LD R_2,d
ADD R_1,R_3,R_1
STORE R_2, a
STORE R_1,d

La suite ?

Les optimisations présentées ici sont élémentaires.

Beaucoup d'optimisations sont encore possibles :
- optimisations locales du code des blocs de base : factoriser les expressions communes, etc
- optimisations globales du code
- …

Mais le cours s'arrête ici !

FIN

Bibliothèque de symboles

∪γ•ε∈∩≠∅∪⊆