Compilation

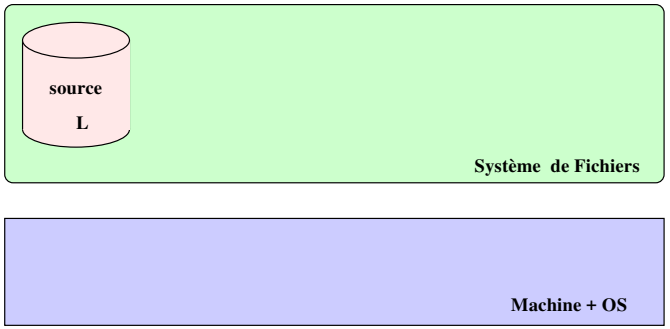
Le 22 octobre 2017

Contents

1 Chaine de compilation

1.1 Exécution d'un programme

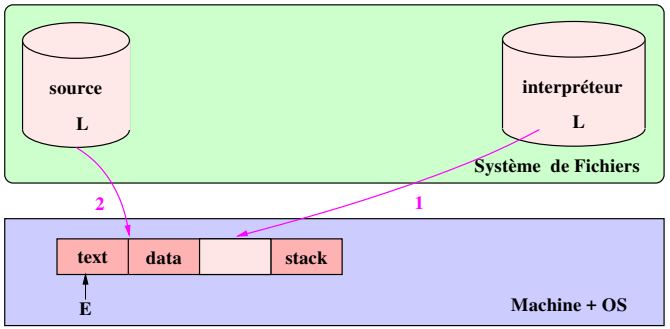
1.1.1 Problème



Une machine: OS + système de fichiers.
Certains fichiers sont des exécutables binaires
L'OS peut les charger et les faire touner en tant que processus.
Soit un un programme écrit en language L,
$\Longrightarrow$ Comment exécuter ce programme

1.1.2 Types d'exécution

1.1.2.1 Interprété



1.1.2.2 Compilé

La compilation est présentée sur la figure 1, l'exécution de l'exécutable généré est présentée sur la figure 2.

1.1.3 Cross compilation

La cross-compilation illustrée sur la figure 3 consiste à générer une exécutable pour une machine-OS X sur une autre machine-OS.
Pour exécuter l'exécutable généré il faut:

- Soit le transférer sur une machine-OS X, et le lancer sur cette machine (voir figure 2).
- Soit le faire tourner dans un émulateur de la machine-OS (voir figure 4).

Exemples de langages émulés: Java
Cross-compilation de Linux vers Windows: MinGW
Émulation de Windows sous Linux: Wine

1.2 Environnement de compilation

La production d'un exécutable binaire à partir d'un code source nécessite un certain nombre d'éléments. L'ensemble de ces éléments est appelé: Environnement de compilation

Les éléments d'un "environnement de compilation" sont des fichiers objet ou des programmes produisant des fichiers objet.

1.2.1 Fichiers objet

Un fichier objet est un fichier binaire avec un format précis. Il contient entre autre:

- segment text: les instructions binaires du programme.
- segment data: les données (variables du programme source)
- table des symboles: adresses (offset dans les segments) des fonctions et variables définies dans le programme.
- table de réallocation: les segments peuvent avoir des "trous".

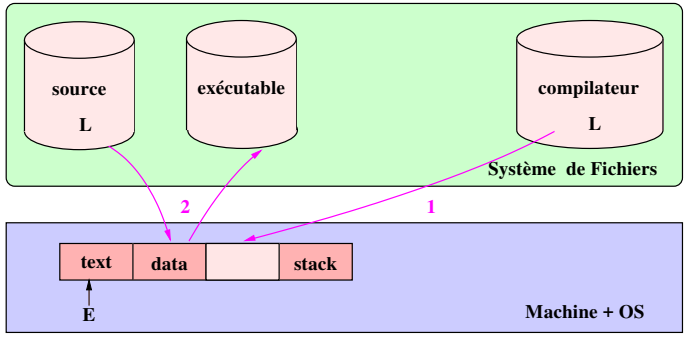


Figure 1: Compilation d'un programme

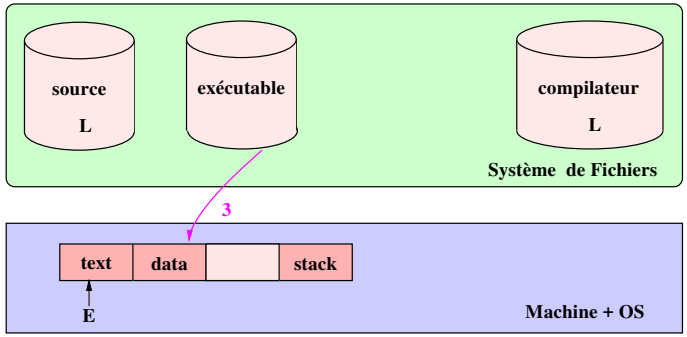


Figure 2: Exécution d'un programme compilé

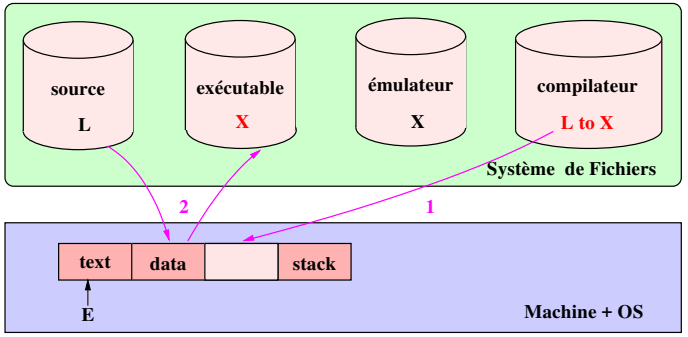


Figure 3: Cross compilation

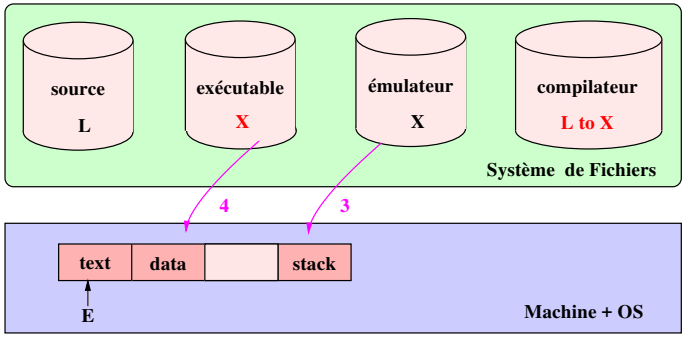


Figure 4: Émulation d'un programme

- Diverses informations: format du fichier, type de fichier objet, où se trouvent les segments et les tables le fichier, à quelles adresses mémoires text et data doivent être chargés, ...

Les principaux types de fichier objets sont:

Objet de base Brique de base permettant de construire les autres.

Bibliothèque statique Un ensemble d'objets de base regroupés dans une archive.

Bibliothèque dynamique Un programme peut le charger en mémoire puis y faire appel.

- Il est avec des trous résolubles.
- Il peut être chargé n'importe où en mémoire (code PIC).
- L'ensemble des bibliothèques dynamiques qui lui sont nécessaires.

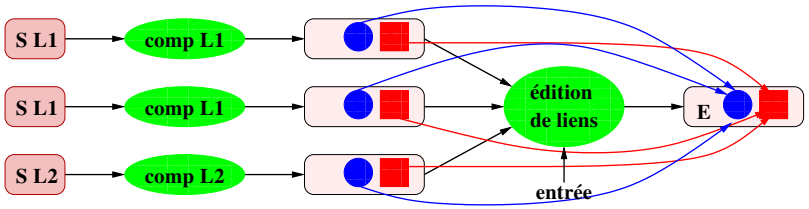
Exécutable L'OS peut le charger en mémoire et le lancer.

- Il est sans trous.
- Les adresses de chargement sont spécifiées.
- L'ensemble des bibliothèques dynamiques qui lui sont nécessaires.
- Il possède un point d'entrée.

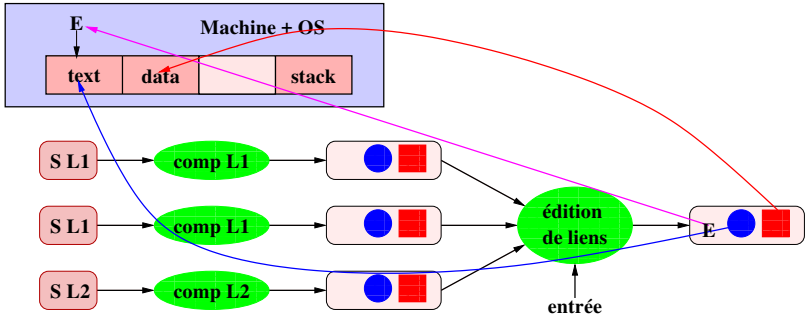
1.2.2 Génération d'un exécutable

1.2.2.1 Génération: Sans bibliothèque

Génération

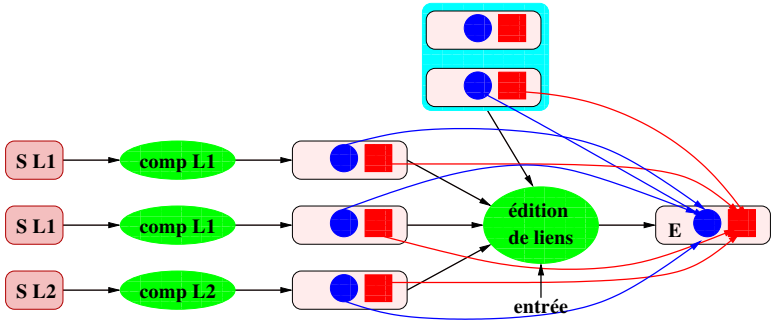


Exécution

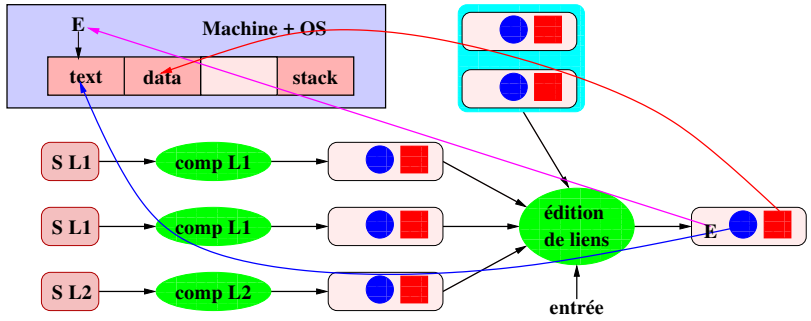


1.2.2.2 Avec des bibliothèques statiques

Génération



Exécution



Algorithme S l'ensemble des symboles non définis, l'éditeur de lien parcourt tous les objets O des bibliothèques, si $S \cap S_O \neq \emptyset$ alors

- O est ajoté à l'exécutable
- $S+ = \{symbolesnondfinisdeO\}$

Particularités

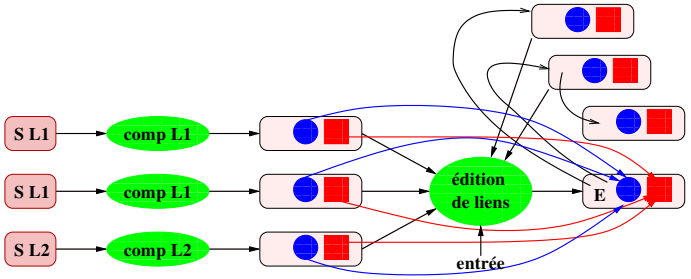
$\Longrightarrow$ L'éditeur de lien de prend que les objets nécessaires.

$\Longrightarrow$ L'ordre des bibliothèques est important.

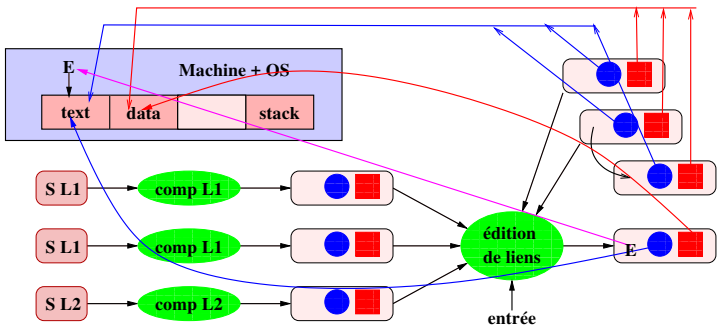
$\Longrightarrow$ Executable statique (auto suffisant car le chargement et l'exécution ne nécessitent pas les bibliothèques).

1.2.2.3 Avec des bibliothèques dynamiques

Génération



Exécution



Édition à la compilation Pour chaque trou, L'éditeur de lien parcourt tous les objets, toutes les bibliothèques dynamiques référencées dans l'ordre pour trouver l'adresse du trou.
Si l'adresse n'est pas trouvée $\Longrightarrow$ abandon.
Si l'adresse est dans une bibliothèque dynamique, le trou est laissé.

Édition de lien au chargement Pour chaque trou, l'éditeur de lien parcourt toutes les bibliothèques dynamiques référencées dans l'ordre pour trouver l'adresse du trou.

Édition de lien à l'exécution Chaque trou est remplacé par une routine de résolution. Quand à l'exécution, on arrive sur cette routine, celle ci parcourt les bibliothèques dynamiques chargées pour trouver l'adresse du trou, puis elle remplace le trou par l'adresse et relance l'exécution.

- Seuls les trous utiles sont résolus.
- Au deuxième passage sur un trou, pas de nouvelle résolution car le trou a été supprimé au premier passage.

Particularités
$\Longrightarrow$ Les bibliothèques sont chargées entièrement.
$\Longrightarrow$ Le segment text des bibliothèques peut être partagé.
$\Longrightarrow$ En changeant les bibliothèques, on peut modifier l'exécution.
$\Longrightarrow$ Exécutable dynamique (non auto suffisant, car son exécution nécessite toutes les bibliothèques.
$\longrightarrow$ Problèmes lors de transfert d'un exécutable d'une machine à une autre.
$\longrightarrow$ Problèmes lors des mises à jour de logiciels: MAJ d'un logiciel B peut entraîner le dysfonctionnement du logiciel A.

1.2.2.4 Avec bibliothèques statiques et dynamiques

Bien sûr, on peut générer un exécutable en mixant bibliothèques statiques et dynamiques.

- Les objets sélectionnés dans les bibliothèques statiques sont ajoutés à l'exécutable.
- Les liens sur les bibliothèques dynamiques sont ajoutés à l'exécutable.

$\Longrightarrow$ on obtient un exécutable dynamique.

1.2.3 Génération d'une bibliothèque statique

Soit N fichiers source file$_i$. On obtient une bibliothèque statique de la manière suivante:

1. pour chaque file$_i$, générez file$_i$.o en appelant le compilateur approprié. On l'arrête avant l'édition de lien.
2. appelez l'archiveur pour regrouper les file$_i$.o dans une archive.

1.2.4 Génération d'une bibliothèque dynamique

Pour générer une bibliothèque dynamique, il faut demander au compilateur de générer du PIC et indiquer à l'édition de lien de générer une bibliothèque dynamique.
C'est le même schéma que la génération d'un exécutable à quelques options près.

1.2.5 Quizz de résumé

1. Un objet est lu par un compilateur.
2. Qui lit un objet exécutable ?
3. Une bibliothèque dynamique a ses symboles résolus?
4. Une bibliothèque dynamique peut toujours être chargée?

5. Une bibliothèque statique peut contenir des bibliothèques dynamiques?
6. Un exécutable (fichier) statique est plus gros que son équivalent dynamique?
7. Un exécutable (mémoire) statique est plus gros que son équivalent dynamique?
8. Un objet exécutable dynamique peut ne pas démarrer?
9. Un objet exécutable statique peut ne pas démarrer?
10. Soit E un exécutable dynamique. On lance E et il charge les bibliothèques dynamiques dll_1, dll_2, ..., dll_n. Faut il toujours les N dll_i pour lancer E?

1.3 Environnement de compilation GNU/Linux

1.3.1 Avant propos.

Pour être concret, nous décrirons ici la chaîne de compilation standard sous linux: elle est composée de nombreux outils:

1. des compilateurs C/C++/fortran/... (C2asm, C++2Asm, ...).
2. un assembleur (as).
3. un éditeur de lien (ld).
4. un front-head (gcc).
5. un ensemble d'objets.

1.3.2 GCC

1.3.2.1 GCC: Vue utilisateur

gcc file.s file.S file.c file.cc file.f file.o lib1.so lib2.a lib... $\Longrightarrow$ a.out
voir figure 5.

$\Longrightarrow$ C'est un front-end de compilation.

$\Longrightarrow$ Il prend N fichiers en entrée et génère 1 objet.

$\Longrightarrow$ Il est configuré pour telle machine et tel OS.

1.3.2.2 Vue interne

voir figure 5.

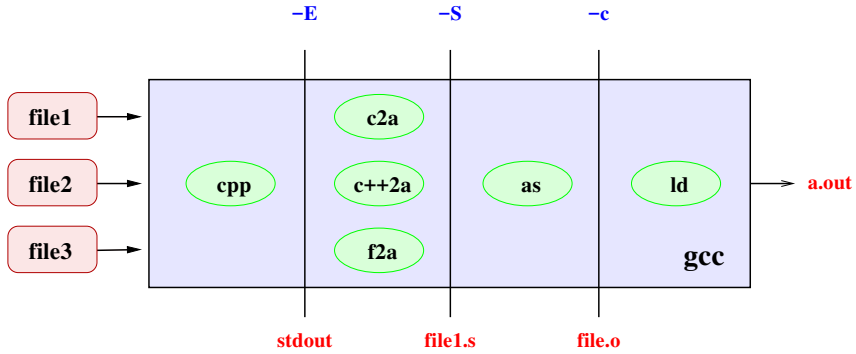
1.3.2.3 Fichiers d'entrée

Par défaut, gcc reconnait les types de fichiers à compiler en fonction de leurs extensions.

.s	fichier assembleur.
.S	fichier assembleur à préprocesser
.c	fichier C
.cc .cp .cxx .c++ .C .CPP	fichier C++
.f .for .FOR	fichier Fortran
.ads .adb	fichier Ada
.o	fichier objet pré-compilé
.so	bibliothèque dynamique
.a	bibliothèque statique

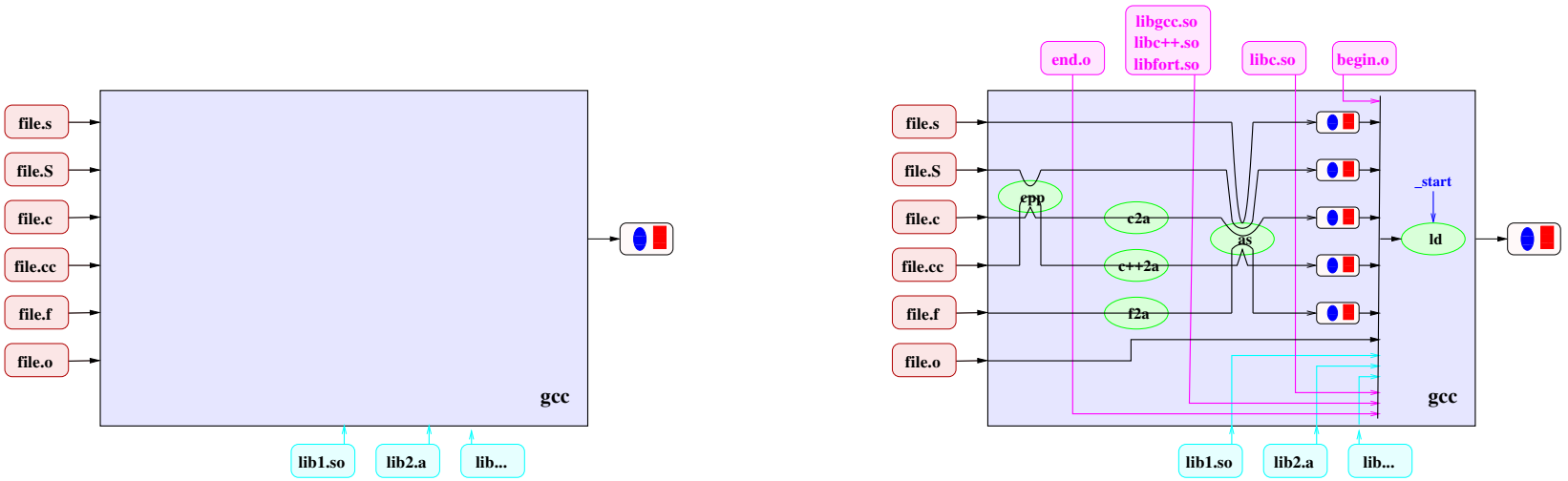
1.3.2.4 Options de contrôle

On peut demander à gcc de s'arrêter dans la chaine de compilation en utilisant les options -E, -S, -c:



On peut renommer le fichier de sortie avec l'option -o filename.

Pour générer une bibliothèque dynamique, il faut demander au compilateur de générer du PIC (-fPIC) et indiquer à l'édition de lien de ne pas générer un exécutable (-shared) ce qu'il fait par défaut.



vue externe vue interne

Figure 5: Environnement GCC

```c
// hello.c
void hello() {
  printf(...);
}
```

```
sh> gcc -fPIC -shared -o hello.so hello.c
sh> objdump -p hello.so
NEEDED libc.so.6
sh>
```

1.3.3 Le chargeur dynamique

1.3.3.1 Chemin des bibliothèques dynamiques

Au chargement/lancement d'un exécutable, le chargeur dynamique cherche les bibliothèques dynamiques nécessaires

1. Dans les répertoires donnés par la variable d'environnement LD_LIBRARY_PATH,
2. Puis dans les répertoires par défaut donnés l'OS.

Soit le programme ci-dessous:

```c
// main.c
void hello();
int main() {
  hello();
  return 0;
}
```

```c
// hello.c
void hello()
{
  printf("hello\n");
}
```

Compilons et lançons le:

```
sh> gcc -fPIC -shared hello.c -o hello.so
sh> gcc main.c hello.so
sh> objdump -p a.out
NEEDED hello.so
NEEDED libc.so.6
sh> ./a.out
./a.out: error while loading hello.so
hello.so: No such file or directory
sh> LD_LIBRARY_PATH=. ./a.out
```

hello
sh>

En jouant sur LD_LIBRARY_PATH, on peut changer la bibliothèque dynamique utilisée par le programme. Générons "../hello.so" qui contient la version française de la fonction hello()".

```
// salut.c
void hello() {
   printf("salut\n");
}
```

```
sh> gcc -fPIC -shared \
    salut.c -o ../hello.so
sh>
```

On peut lancer maintenant au choix a.out en version française ou anglaise:

```
sh> LD_LIBRARY_PATH=.. ./a.out
salut
sh> LD_LIBRARY_PATH=. ./a.out
hello
sh>
```

1.3.3.2 Rediriger des fonctions d'un exécutable

Considérons le programme suivant:

```
// main.c
int main()
{
   un();
   deux();
   un();
   printf("\n");
}
```

```
// undeux.c
#include <stdio.h>

void un()
{ printf("un "); }

void deux()
{ printf("deux "); }
```

On le compile puis on l'exécute par:

```
sh> # génération de ./undeux.so
sh> gcc -fPIC -shared undeux.c -o ./undeux.so
sh> # génération de .a.out et lancement
sh> gcc main.c ./undeux.so
sh> export LD_LIBRARY_PATH=.
sh> ./a.out
un deux un
sh>
```

On désire changer le comportement de la fonction "deux()" sans changer les autres fonctions (la fonction un() dans notre exemple).
Pour cela avant de charger a.out, il suffit de charger une bibliothèque contenant la fonction "deux()". Ainsi quand l'édition de lien dynamique se fera le symbole "deux" étant déjà résolu, il ne sera pas cherché dans undeux.so. Le préchargement de bibliothèque dynamique se fait par la variable d'environnement LD_PRELOAD (LD_PRELOAD= lib1:lib2).

```
// deux.c
#include <stdio.h>

void deux()
{ printf("2 "); }
```

```
sh> gcc -fPIC -shared deux.c -o ./deux.so
sh> ./a.out
un deux un
sh> LD_PRELOAD= deux.so ./a.out
un 2 un
sh>
```

1.3.3.3 Sécurité

Le chargeur dynamique désactive l'utilisation des variables d'environnement LD_LIBRARY_PATH et LD_PRELOAD lorsque l'exécutable est lancé avec un changement de privilège (EID≠ UID).

changement de privilège $\Longrightarrow$ seulement les bibliothèques dynamiques autorisées par le système.

1.4 TD/TP

1.4.1 Chargeur dynamique

Ces exercices présentent les mécanismes utilisés en général et dans GNU/Linux en particulier pour implémenter les bibliothèques dynamiques.

Exercice 1

L'espace virtuel d'un processus en mémoire est donné ci-dessous.

Le chargeur dynamique ajoute les bibliothèques dynamiques à un exécutable après son chargement. Où doit il les placer?

Exercice 2

Cet exercice présente comment les variables sont gérées dans les exécutables dynamiques.

Considérons les deux bibliothèques dynamiques ci-contre que nous appellerons par la suite dll1.so et dll2.so. et les 3 programmes principaux ci-dessous.

```c
// dll1.c
int x;
int*ax() {
  return &x;
}
```

```c
// dll2.c
int y;
int*ax() {
  return &y;
}
```

```c
// main1.c
int main() {
  int*p = ax();
  ...
}
```

```c
// main2.c
extern int x;
int main() {
  int*p = ax();
  int v = x;
}
```

```c
// main3.c
int x;
int main() {
  int*p = ax();
  int v = x;
}
```

Q1 On compile les programmes principaux et on génère les exécutables pij avec l'éditeur de liens statique de la manière suivante:

```
sh> mv dll1.c dll.so
sh> gcc -o p11 main1.c dll.so
sh> gcc -o p21 main2.c dll.so
sh> gcc -o p31 main3.c dll.so
sh>
```

```
sh> mv dll2.c dll.so
sh> gcc -o p12 main1.c dll.so
sh> gcc -o p22 main2.c dll.so
sh> gcc -o p32 main3.c dll.so
sh>
```

(a) Quels sont les exécutables que l'éditeur de liens statique ne générera pas?

(b) Quels sont les exécutables identiques?

(c) Pour "**gcc** f_1.o f_2.o ... l_1.so l_2.so ...", l'édition de liens statique

- vérifie que tous les symboles des .... et .... sont bien ......,
- si c'est le cas il génère un exécutable, qui est toujours le ..... à l'exception des ..... des l_j.so,
- il ne modifie jamais les .... (comme les f_i.o).

Q2 On lance les exécutables de la manière suivante:

```
sh> mv dll1.c dll.so
sh> ./p11 ; ./p12
sh> ./p31 ; ./p32
sh> ./p21
```

```
sh> mv dll2.c dll.so
sh> ./p11 ; ./p12
sh> ./p31 ; ./p32
sh> ./p21
```

(a) Comme ils sont dynamiques, c'est le chargeur/éditeur de liens dynamique qui les charge puis leurs donne la main.

- Il charge l'exécutable en mémoire et met à jour: la table des symboles et la table des trous.

(i) Pour tout dll des ddls de l'objet

- Ajouter les symboles de dll à ....
- Ajouter les trous de dll à ....
- Si dll réfère des ddls reprendre récursivement en .... pour .... non déjà .....

- Avec la table des symboles résoudre les trous.
- Si .... sont résolus alors lancer l'exécutable.

Q3 Quel exécutable le chargeur/éditeur de liens dynamique ne lancera pas?

Q4 Indiquez le segment où le symbole x réside.

dans le main (int v=x)

exe.	dll	exe.bss	dll1.bss
p1	dll1		
p2	dll1		
p3	dll1		
p3	dll2		

dans dll1 (return &x)

exe.	dll	exe.bss	dll1.bss
p1	dll1		
p2	dll1		
p3	dll1		
p3	dll2		

Q5 Comment traduire en assembleur "v=x" dans main2.c et main3.c?

Q6 Comment traduire en assembleur le "return &x" dans ddl1.c?

Suivant les cas il faut renvoyer l'adresse du x dans le exe.bss ou dans le dll1.bss. Une petite table d'indirections est une solution, comme celle ci-contre avec 2 variables. Elle est appelée got pour **G**lobal **O**]bject **T**able.

```
1  .text
2  ax: ...
3  .got
4  got: .long t
5       .long x
6  .bss
7  t:   .long
8  x:   .long
```

(a) Où le chargeur doit il placer le segment ".got".
(b) Où le chargeur doit il placer le segment ".bss" de la dll.
(c) Où sont les trous de ce code?
(d) Une traduction de la fonction ax
int* ax() { return &x }
est donnée si contre.
Que signifie le @ ci-dessus?

```
1  ax:
2    lea  $got-@(%EIP),%eax
3    movl 4(%eax),%eax
4    ret
```

(e) Manque de chance le direct registre comme l'indirect registre sur %eip ne sont pas supportés.
- Écrivez une fonction getpc.
- Donnez une traduction valide de la fonction ax().

Exercice 3

Cet exercice présente comment les fonctions sont gérées dans les exécutables dynamiques.

Q1 Considérons une implémentation des appels de fonctions dynamiques, par une approche similaire à celle des variables dynamiques (exercice précédent) en ajoutant les fonctions à la table got. Ci-dessous le patron avec 3 variables globales et 3 fonctions (f1, ax, f2).

```
1  .text
2    ...
3
4  .got
5  got: .long // v1
6       .long // v2
7       .long // v3
8       .long // f1
9       .long // ax
10      .long // f2
```

(a) Est-ce que les main et les dll suivent cette structure?
(b) Donnez l'assembleur de "int*px=ax()" dans un main.
(c) Donnez l'assembleur de "int*px=ax()" dans une dll (attention PIC).

Q2 Un simple "Hello World" avec QT charge au moins 40 dll. Les dll libc, libX11, libQtCore exportent respectivement environ 2000, 1300 et 10 000 fonctions. En utilisant, l'approche précédente

(a) Combien de got auront une entrée strcmp?

(b) Combien de fonctions comme strcmp?

(c) Donner une estimation du nombre de résolutions de symbole et d'entrées de got que le chargeur/éditeur de lien dynamique devra résoudre pour un simple "Hello World" avec QT.

(d) Ces résolutions et mises à jour des entrées des GOT sont elles utiles?

$\Longrightarrow$ Il faut donc s'orienter sur une résolution fainéante. On ne résout et ne met à jour qu'au premier appel.

Q3 Ci-dessous, le code assembleur d'une approche fainéante pour 3 variables globales et 3 fonctions (f1, ax et f2). Pour la fonction ax, on crée la fonction ax@plt[1], et une entrée dans la table got (la $6^{ième}$) qui est initialisée sur l'adresse du 1^{er} pushl (ligne 10) de la fonction ax@plt.

[1] plt: Procedure Linkage Table

```
1  .plt
2  f1@plt:
3      jmp    20(%ebx)
4      pushl  $0
5      pushl  %ebx
6      jmp   *4(%ebx)
7
8  ax@plt:
9      jmp    24(%ebx)
10     pushl  $1
11     pushl  %ebx
12     jmp   *4(%ebx)
13
14 f2@plt:
15     jmp    28(%ebx)
16     pushl  $2
```

```
17     pushl  %ebx
18     jmp   *4(%ebx)
19
20 .text
21     ...
22 .got
23 got:
24     .long // info
25     .long v1
26     .long v2
27     .long v3
28     .long // fct resolution
29     .long f1@plt+6  // f1
30     .long ax@plt+6  // ax
31     .long f2@plt+6  // f2
```

- Un appel "ax()" est traduit par "`call` ax@str" avec %ebx pointant sur la got. On arrive alors sur la ligne 9.
- Au premier appel, le jump branche sur la ligne 10. On appelle alors la fonction de résolution avec les arguments 1 (identifiant de la fonction ax) et l'adresse de la got. La fonction de résolution
  - trouve l'adresse de la fonction ax(),
  - l'écrit dans l'entrée 6 de la got,
  - dépile 8 octets et saute (jmp) à l'adresse de ax().

(a) Où branche le "ret" de la fonction ax()?
(b) Que se passe-t-il au second appel?

1.4.2 Redéfinition de malloc

L'objectif de ce TP est d'afficher pour un exécutable existant tel "/bin/ls" ou "/bin/grep" le nombre de malloc qu'il a exécutés au cours de son exécution.

Exercice 1

Q1 Qui fournit malloc?
Q2 Comment procéder?
Q3 Quelles devront être les caractéristiques des exécutables?

Exercice 2

```
1  // malloc.c
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int mynb;
6  void mynbdump() {
7    char mess[100];
8    int n=sprintf(mess,
9      "--> %d malloc\n",
10     ...);
11   write(2,...,...);
12 }
13
14 void *malloc(size_t n) {
15   if (n!=0) mynb++;
16   void *addr = ...;
17   return addr;
18 }
```

Le code ci-contre donne le squelette de la bibliothèque dynamique.

Q1 Complétez la fonction mynbdump.
Q2 Quand doit elle être appelée ?
Q3 Dans la fonction malloc, quelle valeur doit prendre addr (ligne 15)?
Q4 On ne peut pas réécrire malloc facilement. Mais (merci aux concepteurs de la libc), complètez la fonction malloc en utilisant la copie d'écran ci-dessous sachant que l'option **-T** de la commande **objdump** affiche la table des symboles.

```
sh> objdump -T /lib64/libc.so.6 | grep -w malloc
00076b40 g DF .text 000001de GLIBC_2.2.5 malloc
sh> objdump -T /lib64/libc.so.6 | grep 00076b40
00076b40 g DF .text 000001de GLIBC_2.2.5 __libc_malloc
00076b40 g DF .text 000001de GLIBC_2.2.5 malloc
sh>
```

Exercice 3

Q1 Ajoutez un appel à mynbdump dans malloc après le "if", puis compiler la bibliothèque.

```
sh> gcc -fPIC -shared -o malloc.so malloc.c
sh>
```

Q2 Essayez la bibliothèque sur "/bin/ls" puis "/bin/grep".

```
sh> LD_PRELOAD=./malloc.so ls
```

–> 1 malloc
...
–> 39 malloc
gnu bee gnat
sh>

Q3 L'appel final à mynbdump n'apparait pas ce qui est normal car on ne l'a pas déclenché. Pour cela il faut mettre un tel appel dans la section ".fini". On pourrait faire ça en assembleur ou en C avec des pragma.

Mais on sait que en C++, les appels au constructeur des variables globales sont placés dans la section ".init" et ceux aus destructeurs dans la dans la section ".fini". Le code ci-contre devrait fonctionner:

```
1 // malloc.cc
2 extern "C" {
3 #include "malloc.c"
4 }
5 struct X {
6   ~X() { mynbdump(); }
7 };
8 static X x;
```

Essayons cette version:

sh> gcc -fPIC -shared -o malloc++.so malloc.cc
sh> LD_PRELOAD=./malloc++.so **ls**
–> 1 malloc
...
–> 39 malloc
gnu bee gnat
sh>

⟹ encore raté.

Q4 En fait l'appel a bien été fait après le main de "ls". Ce qui ne marche pas c'est que après le main de "ls" la fonction de libération de la libc (exit) est appelée avant notre destructeur. "exit" ferme tous les flux libc ouvert et en particulier stderr.
Pour palier ceci:

a) Modifier "mynbdump" pour qu'il écrive dans le fichier 200 "write(200,mess,n)".

b) Lors du lancement de ls rediriger le fichier 200 sur le 2.

sh> gcc -fPIC -shared -o malloc++.so malloc.cc
sh> LD_PRELOAD=./malloc++.so **ls** 200>&2
–> 1 malloc
...
–> 39 malloc
gnu bee gnat
–> 39 malloc
sh>

⟹ Bingo.

Q5 Pour peaufiner, a) commentez l'appel à mynbdump dans la fonction malloc, b) Faite le dup dans le constructeur de X "X() { dup2(2,200); }".

sh> gcc -fPIC -shared -o malloc++.so malloc.cc
sh> LD_PRELOAD=./malloc++.so **ls**
gnu bee gnat
–> 39 malloc
sh>

1.4.3 Un petit exploit

Ces exercices montrent comment modifier le comportement d'un programme dont on a pas les sources. On part du programme /pub/ia/ico/exploit qui ne va pas très loin pour diverses raisons.

Exercice 1

Lancez le programme /pub/ia/ico/exploit, il s'arrête sur:

```
exploit:  can not open the key file:  No such file or directory
```

Q1 Surcharger l'appel système open, en affichant les noms de fichiers que le programme essaye d'ouvrir.

Q2 Quel fichier pose problème?

Q3 Vous ne pouvez pas le créer (il faut être root pour créer un fichier dans /etc).

- Créez ce fichier key dans votre HOME.
- Faites en sorte que le programme ouvre ce fichier à la place de celui dans /etc.

Exercice 2

Ecrivez "bonjour monsieur" dans le fichier key de votre HOME puis relancez le programme /pub/ia/ico/exploit, il s'arrête sur

```
exploit:  invalid key
```

Q1 Surchargez la fonction strcmp et afficher les chaines de caractères qui sont comparées.

Q2 Modifiez le fichier key de votre HOME.

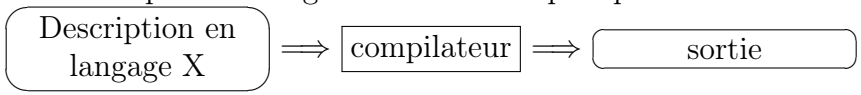
Q3 Relancez le programme /pub/ia/ico/exploit.

2 Compilateur

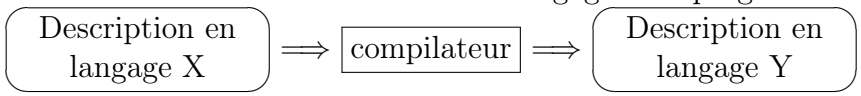
2.1 introduction

2.1.1 Définition

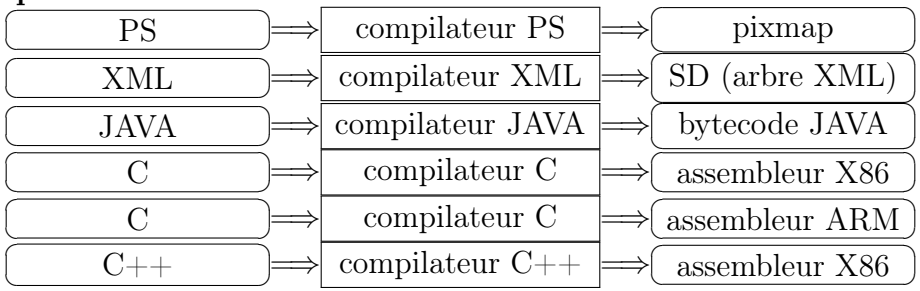
Général Un compilateur est un programme qui lit un flux d'entrée structuré par une grammaire et qui produit une sortie.



De code L'entrée et la sortie sont des langages de programmation.



Exemple



2.1.2 Phases

source $\Longrightarrow$ analyse lexicale $\Longrightarrow$ { (t_0,v_0) (t_1,v_1), ... }

{ (t_0,v_0) (t_1,v_1), ... } $\Longrightarrow$ analyse grammaticale $\Longrightarrow$ AST*

AST $\Longrightarrow$ analyse sémantique $\Longrightarrow$ SD

SD $\Longrightarrow$ génération $\Longrightarrow$ sortie

*: Arbre de syntaxe abstraite

Les étapes de la génération dépendent beaucoup du type de compilateur. Dans la génération de code (langage $\Longrightarrow$ langage) elle se décompose en:

SD $\Longrightarrow$ optimisation globale $\Longrightarrow$ SD (indépendante du langage cible)

SD $\Longrightarrow$ traduction $\Longrightarrow$ SD

SD $\Longrightarrow$ optimisation cible $\Longrightarrow$ SD

SD $\Longrightarrow$ génération $\Longrightarrow$ sortie

2.1.3 Exemple

```
1  int f1(int x, int y)
2  { return x + y; }
3
4  int f2(int a, int b)
5  {
6    int z;
7    z = f1(a,b+b);
8    z = z + 2;
9    return z;
10 }
```

prog := (func)$^+$
func := INT ID '(' INT ID ',' INT ID ')'
'{' (decl|inst)$^+$ '}'
decl := INT ID ';'
inst := RET expr ';' | ID '=' expr ';'
expr := terme | terme '+' terme ';'
terme:= IVAL | ID | ID '(' expr ',' expr ')'

1. Quels sont les terminaux de la grammaire?
2. Donnez le résultat de l'analyse lexicale.
3. Donnez le résultat de l'analyse grammaticale.
4. Donnez une SD générée par l'analyse sémantique.
5. Donnez une SD optimisée.
6. Générez le code X86.

2.2 Analyse lexicale & grammaticale

2.2.1 Grammaire formelle

2.2.1.1 Grammaire

Une grammaire G est constituée de:

- un ensemble fini Σ de symboles terminaux,
- un ensemble fini N de symboles non-terminaux (disjoint de Σ),
- un ensemble fini P de règles de production, une production étant une fonction : $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$,
- un symbole de départ s non-terminal (avec $s \in N$) et appelé axiome.

Une grammaire est donc un quadruplet (N, Σ, P, s).

Par exemple, un troupeau de chèvres peut se définir par la grammaire:

$$\begin{array}{l}\Sigma=\{\ \textit{chèvre}\ \epsilon\ \}\\ N=\{\ \text{TROUPEAU TOP}\ \}\\ \text{S}=\text{TOP}\end{array} \quad P=\left\{\begin{array}{ll}\text{TOP} & \to\text{TROUPEAU}\\ \text{TROUPEAU} & \to\epsilon\\ \text{TROUPEAU} & \to\text{TROUPEAU}\ \textit{chèvre}\end{array}\right\}$$

Une dérivation (notée $\Longrightarrow$) est l'application d'une production sur une suite de symboles.

$\text{TROUPEAU}\ \textit{chèvre}_1 \Longrightarrow \text{TROUPEAU}\ \textit{chèvre}_0\ \textit{chèvre}_1$
$\text{TROUPEAU}\ \textit{chèvre}_1 \Longrightarrow \textit{chèvre}_1$

Le langage de la grammaire G (noté $\mathcal{L}(G)$) est l'ensemble des phrases formées que de symboles terminaux obtenues par une suite de dérivations partant de l'axiome de la grammaire. Pour la grammaire "troupeau" précédente $\mathcal{L}(\textit{troupeau}) = \{\textit{chèvre}^i\}$
En notant $\Longrightarrow^+$ la fermeture transitive de $\Longrightarrow$, la définition formelle de $\mathcal{L}(G)$ est:

$\forall\ p \in \Sigma^*,\ p \in \mathcal{L}(G] \iff \text{s} \Longrightarrow^+ p$

Deux grammaire sont dites faiblement équivalentes si elles engendrent le même langage.

2.2.1.2 Interprétation

Considérons la grammaire "troupeau2" ci-dessous, elle définit un troupeau de caprins.

$$\Sigma = \{\ \textit{chèvre}\ \textit{bouc}\ \epsilon\ \}$$
$$N = \{\ \text{TROUPEAU}\ \text{TOP}\ \}$$
$$\text{S} = \text{TOP}$$

$$P = \left\{ \begin{array}{l} \text{TOP} \rightarrow \text{TROUPEAU} \\ \text{TROUPEAU} \rightarrow \epsilon \\ \text{TROUPEAU} \rightarrow \text{TROUPEAU}\ \textit{chèvre} \\ \text{TROUPEAU} \rightarrow \text{TROUPEAU}\ \textit{bouc} \\ \text{TROUPEAU} \rightarrow \text{TROUPEAU}\ \textit{chèvre}\ \textit{bouc} \\ \text{TROUPEAU} \rightarrow \text{TROUPEAU}\ \textit{bouc}\ \textit{chèvre} \end{array} \right\}$$

De plus elle permet de regrouper les caprins en couples ce qui donne une interprétation à une phrase du langage.

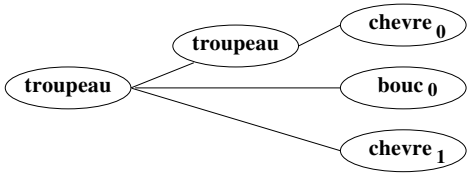
$\textit{chèvre}_0\ \textit{bouc}_0\ \textit{chèvre}_1 \rightarrow (\textit{chèvre}_1, \textit{bouc}_0) + \textit{chèvre}_1$

Pour conserver/extraire cette interprétation, il faut avoir le graphe de dérivation:

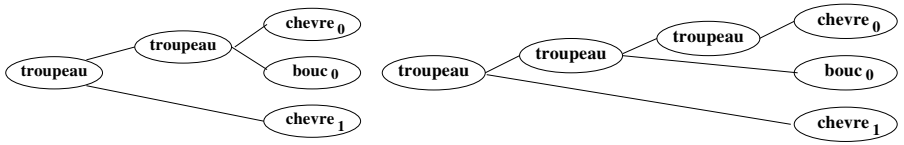
$$\begin{array}{c} \text{TROUPEAU} \rightarrow \textit{epsilon} \\ \downarrow \\ \text{TROUPEAU} \rightarrow \text{TROUPEAU}\ \textit{chèvre}_0 \\ \downarrow \\ \hline \text{TROUPEAU} \rightarrow \text{TROUPEAU}\ \textit{bouc}_0\ \textit{chèvre}_1 \\ \hline \downarrow \\ \text{TOP} \rightarrow \text{TROUPEAU} \end{array}$$

Dans ce graphe la ligne "TROUPEAU $\rightarrow$ TROUPEAU $\textit{bouc}_0\ \textit{chèvre}_1$" montre que le couple ($\textit{bouc}_0\ \textit{chèvre}_1$) a été sélectionné. Une interprétation est aussi mise en exergue plus facilement par son graphe de syntaxe abstraite. Par exemple, le graphe de syntaxe abstraite du graphe de dérivation précédent est donné ci-dessous sous forme textuelle ({...}: indiquant un troupeau) et graphique.

$\{\ \{\ \textit{chèvre}_0\ \}\ \textit{bouc}_0\ \textit{chèvre}_1\ \}$



La phrase "$\textit{chèvre}_0\ \textit{bouc}_0\ \textit{chèvre}_1$" peut aussi interpréter par la cette grammaire de 2 autres façons:



Une grammaire G qui ne donne pas une interprétation unique pour chacune des phrases de son langage $\mathcal{L}(G)$ est dite ambigüe.

2.2.1.3 Récursivité gauche/droite

Une production récursive à gauche est de la forme:

$\text{A} \rightarrow^+ \text{A}\ w$

Une production récursive à droite est de la forme:

$\text{A} \rightarrow^+ w\ \text{A}$

Une grammaire est récursive à gauche si elle contient une production récursive à gauche (idem à droite).

Une grammaire peut donc très bien être récursive à la fois à gauche et à droite.

2.2.1.4 Classification

La hiérarchie de Chomsky[2] classe les grammaires formelles en 4 types numérotés de $\emptyset$ à 3. Dans cette classification on a t3<t2<t1<t0 pour la complexité des grammaires et $t0 \subset t1 \subset t2 \subset t3$.

type 0 aucun des suivants

type 1, ou langages contextuels Les règles sont de la forme
$\alpha\,\text{A}\,\beta \rightarrow \alpha\,w\,\beta$ avec $\alpha, \beta, w \in (\Sigma \cup N)^*$ et $\text{A} \in N$.
Exemple $a^i b^i c^i$, $a^i e^* b^i f^i c^i$, $a^i b^{2i} c^{3i+1} d^i$.

type 2, ou langages algébriques Les règles sont de la forme
$\text{A} \rightarrow w$ avec $w \in (\Sigma \cup N)^*$ et $\text{A} \in N$.
Exemple $a^i b^i$, $a^i c^* b^i$, $a^i b^{2i+1}$.

type 3, ou langages rationnels Les règles sont de la forme
soit linéaire gauche : $\text{A} \rightarrow \text{B}\,a$, $\text{A} \rightarrow a$
soit linéaire droite : $\text{A} \rightarrow a\,\text{B}$, $\text{Q} \rightarrow a$
avec $a \in \Sigma$ et $\text{A}, \text{B} \in N$.
Exemple $a^i b^j c^k$, $(a|b)^* c^k$.

Les grammaires peuvent aussi être classées par leurs analyseurs (algorithme qui donne le graphe de dérivations d'une phrase).

[2] Noam Chomsky

grammaire	analyseur fiable	analyseur approximatif
type 0		
type 1 (contextuelle)		
type 2 (algébrique)		automate à pile: ll et lr
type 3 (rationnel)	automate d'états fini	

Soit AN un analyseur et G une grammaire non ambigüe, on dit que grammaire G est AN si l'analyseur AN est capable d'analyser toutes les phrases de $\mathcal{L}(G)$.
Si l'analyseur AN n'est capable de traiter qu'un sous ensemble P de $\mathcal{L}(G)$, on dit que G est conflictuelle pour AN sur $\mathcal{L}(G)$-P.

$\Longrightarrow$ L'ambigüité est une propriété de la grammaire.

$\Longrightarrow$ Les conflits sont une faiblesse de l'analyseur.

Bien sûr, pour une grammaire ambigüe, l'analyseur génère des conflits. Il y donc des "bons" (dus à la grammaire) et mauvais (dus à la faiblesse de l'analyseur) "conflits".

2.2.1.5 Exercices

Exercice 1

Considérons la grammaire "GL":

$$\Sigma=\{\ chèvre\ \epsilon\ \}\quad N=\{\ \text{TROUPEAU TOP}\ \}\quad \text{S}=\text{TOP}\quad P=\left\{\begin{array}{l}\text{TOP}\longrightarrow\text{TROUPEAU}\\ \text{TROUPEAU}\longrightarrow\epsilon\\ \text{TROUPEAU}\longrightarrow\text{TROUPEAU}\ chèvre\end{array}\right\}$$

Considérons la grammaire "GR":

$$\Sigma=\{\ chèvre\ \epsilon\ \}\quad N=\{\ \text{TROUPEAU TOP}\ \}\quad \text{S}=\text{TOP}\quad P=\left\{\begin{array}{l}\text{TOP}\longrightarrow\text{TROUPEAU}\\ \text{TROUPEAU}\longrightarrow\epsilon\\ \text{TROUPEAU}\longrightarrow chèvre\ \text{TROUPEAU}\end{array}\right\}$$

a) Ces grammaires sont-elles récursive gauche ou droite?

b) Donnez pour GL et GR, la suite de dérivations qui conduit à la phrase "$chèvre_0\ chèvre_1\ chèvre_2$".

c) Donnez les arbres de syntaxe abstraite correspondant aux 2 dérivations précédentes.

d) GL et GR sont-elles équivalentes?

Exercice 2

Considérons la grammaire "troupeau":

$$\Sigma=\{\ chèvre\ bouc\ \epsilon\ \}\quad N=\{\ \text{TROUPEAU TOP}\ \}\quad \text{S}=\text{TOP}\quad P=\left\{\begin{array}{l}\text{TOP}\longrightarrow\text{TROUPEAU}\\ \text{TROUPEAU}\longrightarrow\epsilon\\ \text{TROUPEAU}\longrightarrow\text{TROUPEAU}\ chèvre\\ \text{TROUPEAU}\longrightarrow\text{TROUPEAU}\ bouc\\ \text{TROUPEAU}\longrightarrow\text{TROUPEAU}\ chèvre\ bouc\\ \text{TROUPEAU}\longrightarrow\text{TROUPEAU}\ bouc\ chèvre\end{array}\right\}$$

a) Cette grammaire est-elle récursive gauche ou droite?

b) Donnez le (ou les) arbre(s) de syntaxe abstraite de la phrase "$chèvre_0\ chèvre_1\ chèvre_2\ chèvre_3$"

c) Donnez le (ou les) arbre(s) de syntaxe abstraite de la phrase "$bouc_0\ chèvre_0\ chèvre_1\ chèvre_0$"

d) Donnez le (ou les) arbre(s) de syntaxe abstraite de la phrase "$bouc_0\ chèvre_0\ chèvre_1\ chèvre_1$"

e) La grammaire est elle ambigüe?

f) Donnez les phrases du langage $\mathcal{L}(troupeau)$ qui ne sont pas ambigües.

Exercice 3

Considérons la grammaire "exp":

$$\Sigma=\{\ v\ +\ *\ \}\quad N=\{\ \text{EXPR TOP}\ \}\quad \text{S}=\text{TOP}\quad P=\left\{\begin{array}{l}\text{TOP}\longrightarrow\text{EXPR}\\ \text{EXPR}\longrightarrow v\\ \text{EXPR}\longrightarrow\text{EXPR}\ +\ \text{EXPR}\\ \text{EXPR}\longrightarrow\text{EXPR}\ *\ \text{EXPR}\end{array}\right\}$$

a) Quel est le type de cette grammaire?

b) Cette grammaire est-elle récursive gauche ou droite?

c) Donnez le (ou les) arbre(s) de syntaxe abstraite de la phrase "1 + 2 * 3". Cette grammaire est elle ambigüe?

d) La phrase "1+2+3" est-elle ambigüe?

e) Proposez une grammaire équivalente mais non ambigüe. Pour cela, il faut partir de "un terme est une somme de monômes".

f) Donnez le graphe de dérivation et l'arbre de syntaxe abstraite de la phrase "1 + 2 * 3".

Exercice 4

Une grammaire est dite croissante si pour toute règle $u \rightarrow v$, le nombre d'éléments de u est inférieur ou égal au nombre d'éléments de v.
Toute grammaire croissante peut être transformée en grammaire contextuelle et vice versa.
Considérons la grammaire G:

$$\Sigma = \{\ a\ b\ c\ \} \quad N = \{\ \mathrm{S}\ \mathrm{B}\ \} \quad \mathrm{S} = \mathrm{S} \quad P = \left\{ \begin{array}{ll} \mathrm{S} & \rightarrow abc \\ \mathrm{S} & \rightarrow a\mathrm{S}\mathrm{B}c \\ c\mathrm{B} & \rightarrow \mathrm{B}c \\ b\mathrm{B} & \rightarrow bb \end{array} \right\}$$

a) Quel est le type de cette grammaire?

b) Cette grammaire est-elle récursive gauche ou droite?

c) Complètez la suite de dérivation conduisant à la phrase "aaabbbccc".

$\mathrm{S} \Longrightarrow a\mathrm{S}\mathrm{B}c \Longrightarrow aa\mathrm{S}\mathrm{B}c\mathrm{B}c \Longrightarrow ...$

d) Donnez le graphe de syntaxe abstraite correspondant à la dérivation précédente.

2.2.2 Analyse lexicale

2.2.2.1 Rôle

- Découper le flux d'entrée en symbole (couple: identifiant+valeur).
- Manger les séparateurs humains (espace, saut de ligne, ...).
- Manger ce qui ne fait pas partie du langage source comme les commentaires.

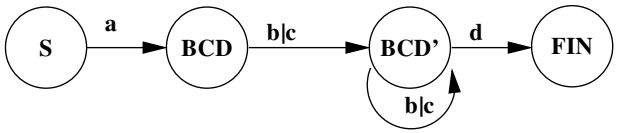
2.2.2.2 Une expression régulière

Entrées usuelles

- Expression régulière: $a(b|c)^+d$
- Grammaire rationnelle: $S \rightarrow a\,BCD$; $BCD \rightarrow b\,BCD' | c\,BCD'$; $BCD' \rightarrow b\,BCD' | c\,BCD' | d$

Sorties

- Automate d'états fini.



- Analyseur LL (chapitre suivant)

Complexité Le nombre de lettres du symbole reconnu

2.2.2.3 Ensemble d'expressions régulières

Principe On regroupe les automates (ou grammaires) en un(e) seul(e).

Algorithme Parcourir l'automate en stockant tous les symboles reconnus.

$\Longrightarrow$ **Un seul symbole reconnu**
On le sélectionne.

$\Longrightarrow$ **Plusieurs symboles reconnus**

- Si un seul plus long alors le sélectionner.
- Si plusieurs plus longs alors on en sélectionne un suivant un ordre pré-établi.

La recherche suivante commence après le symbole reconnu.

Complexité

- En général le nombre de lettres du symbole reconnu.
- Quadratique dans quelques cas rares:
  mot:a^{n+1}; regexp=$\{$ a $a^n b$ $\}$

2.2.3 Analyse grammaticale

2.2.3.1 Analyse LL

La famille LL est l'acronyme de "**L**eft to right scan and **L**eftmost derivation".

Principe

Pour analyser une phrase, il est tentant de créer une fonction récursive pour chaque symbole non terminal qui regroupe toutes les règles qui le produisent et qui mange les symboles terminaux nécessaires. Par exemple:

E→P + E
E→P * P
E→P v

```
1  int E() {
2    if (P()==0) return 0;
3    int t = lire_tk();
4    if (t=='+') return E();
5    if (t=='*') return P();
6    return nt=='v' ;
7  }
```

Une telle approche interdit une récursivité gauche car sinon on aura
int E() { ... E() ...}
et on part dans une récursion infinie.

Avec une telle approche "LL(0)", on ne va pas très loin par exemple ces règles ne sont pas analysables:
E→P E; E→ v;
Pour pouvoir les analyser il faut savoir quelle règle s'applique supposons que P commence soit par le terminal a soit par le terminal b (on note premier(P)={a,b}) et nt est le prochain terminal non encore consommé, on peut alors écrire la fonction E():

```
1  int E() {
2    if (nt=='a' || nt=='b') return P() || E();
3    if (nt=='v') { avance(); return 1; }
4    return 0;
5  }
```

On anticipe d'un terminal, c'est du LL(1).

Si "premier(P)={v})", on ne peut toujours pas choisir entre les deux règles, on peut anticiper sur 2 terminaux nt_0 et nt_1 et calculer l'ensemble second(P), c'est du LL(2).

LL(k) $\Longrightarrow$ on anticipe sur les k prochains symboles terminaux pour écrire les fonctions des symboles non terminaux.

Table LL(1)

Un analyseur LL(1) est basé sur une table qui donne pour chaque symbole non terminal, la production à appliquer en fonction en fonction du prochain symbole terminal.

$\Sigma = \{ + * v \}$
$N = \{ S\ E \}$
$P = \left\{ \begin{array}{l} S \to E\ \$ \\ E \to + E\ E \\ E \to * E\ E \\ E \to v\$ \end{array} \right\}$
s $= S$

	+	*	v	$
S	$S \to E\$$			
E	$E \to + E\ E$	$E \to * E\ E$	$E \to v$	

En remplissant cette table, si on trouve 2 ou plus règles dans une case alors la grammaire n'est pas LL(1).

Exemple

Considérons la grammaire suivante:

$\Sigma = \{ v + * \}$
$N = \{ E \}$
s $= E$
$P = \left\{ \begin{array}{l} E \to E + E \\ E \to E * E \\ E \to v \end{array} \right\}$

Elle est ambigüe, mais "sympa" car elle génère des AST pratiques. Pour LL(1), on est obligé de la rendre non ambigüe.

$\Sigma = \{ v + * \}$
$N = \{ E \}$
s $= E$
$P = \left\{ \begin{array}{l} E \to E + M \\ E \to M \\ M \to M * v \\ M \to v \end{array} \right\}$

Elle est récursive à gauche il faut la transformer en récursive droite.

$\Sigma = \{ v + * \epsilon \}$
$N = \{ E \}$
s $= E$
$P = \left\{ \begin{array}{l} E \to M\ E' \\ E' \to + M\ E' \\ E' \to \epsilon \\ M \to v M' \\ M' \to * v\ M' \\ M' \to \epsilon \end{array} \right\}$

Calculons la table de LL(1)

	+	*	v	$
E			$E \rightarrow M\ E'$	
E'	$E' \rightarrow +\ M\ E'$			$E' \rightarrow \epsilon$
M			$M \rightarrow v\ M'$	
M'		$M' \rightarrow *\ v\ M'$		$M'' \rightarrow \epsilon$

Il n'y a jamais plus d'une règle par case, cette grammaire est bien LL(1). Donnez l'arbre de syntaxe abstraite de la phrase 1*2+3*4

En pratique

Si la grammaire n'est pas LL:

- Transformation de la grammaire
  $\Longrightarrow$ perte de lisibilité
- AST obtenus demande un travail pour les remettre sous forme exploitable.

$\Longrightarrow$ pas très utilisable sur des grammaires complexes.

Si la grammaire est LL (ex: langage rationnel récursif à droite):

$\Longrightarrow$ analyseurs efficaces et petits.

2.2.3.2 Analyse LR

La famille LR est l'acronyme de "**L**eft to right scan, **R**ightmost derivation" soit dérivation de la gauche vers la droite avec priorité pour la droite.

Ses caractéristique sont:

- C'est une approche par automate: (état,terminal) $\longrightarrow$ état.
- Réorganisation de l'automate pour palier l'explosion du nombre d'états (infini pour grammaire récursive droite).
- Des raffinements de cette approche donne les analyseurs de cette famille d'analyseurs:

  LR(0) LR(1) Une analyse LR sans ou avec la connaissance du prochain terminal.

  SLR(1) "Simple Left-to-right Rightmost derivation" variante de LR(1) (moins d'états que LR(1)).

  LALR(1) "Look-Ahead Left-to-right Rightmost derivation" autre variante de LR(1) (moins d'états que LR(1)).

$$\Sigma=\{\ 1\ 2\ 3\ +\ \} \quad N=\{\ S\ T\ V\ \} \quad \text{s} = S \quad P=\left\{\begin{array}{l} S \rightarrow T \\ T \rightarrow V \\ T \rightarrow T+V \\ V \rightarrow 1|2|3 \end{array}\right\}$$

Figure 6: Grammaire G

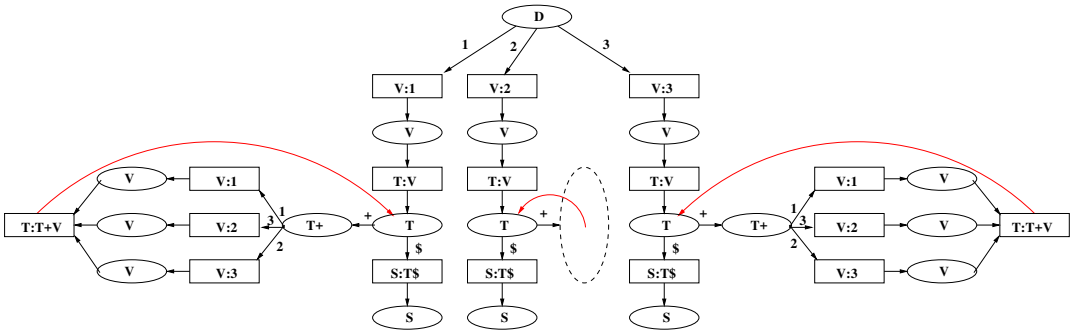


Figure 7: Automate d'analyse de la grammaire G

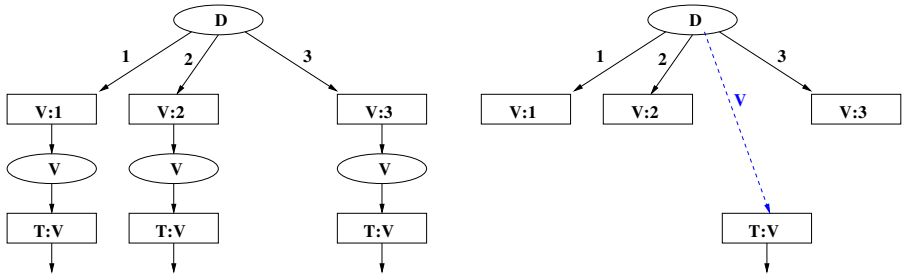
Principe

La grammaire G présentée sur la figure 6 est parfaitement analysé par l'automate de la figure 7. On remarque

- 2 types d'états, les *shift* ($\boxed{\text{T}} \xrightarrow{+}$) qui avancent dans le flux de terminaux et *reduce* ($\boxed{\text{T:T+V}}$) qui reconnaissent une règle sans consommer de terminaux du flux.
- Une explosion du nombre d'états.
- Mais beaucoup de parties du graphe sont homomorphes.

Réduction du graphe d'états

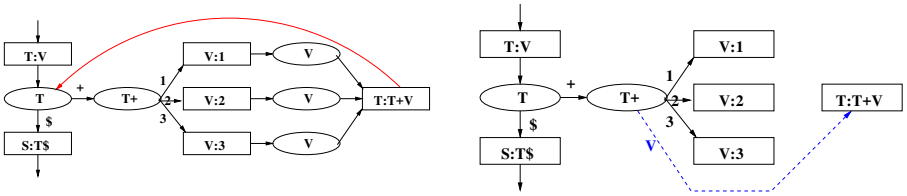
Le haut du graphe de la figure 7 peut être transformé de la manière suivante:



Son parcourt se fait de la manière suivante:

- Sur un *shift*, on empile l'état.
- Sur un *reduce* d'une règle $L \rightarrow r_1 r_2 \ldots r_n$ on ajoute en début du flux le non terminal L, puis on dépile n états et on va dans le dernier état dépilé.

Ce genre d'automate est appelé automate à pile.
On peut aussi appliquer la même transformation sur la boucle.



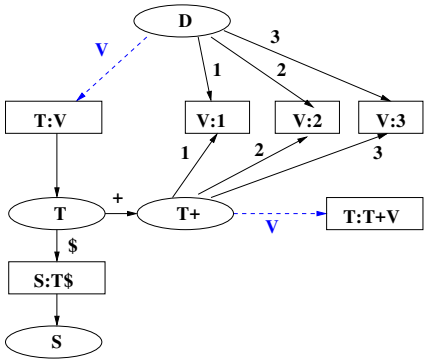
On obtient alors l'automate 1 de la figure 8. En appliquant le même type de transformations, on obtient finalement l'automate 2. Le nombre d'états est passé de 43 à 10 dans ce dernier automate à pile (ou 22 à 4, les *reduce*étant des états intégrables dans leurs prédécesseurs).

Automate LR (à pile)

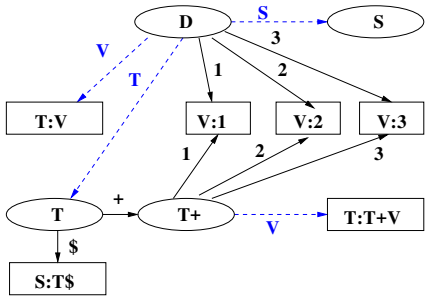
L'automate (3) de la figure 8 est l'automate LR(1) genéré par bison. Dans ces automates chaque état correspond à un ensemble de règles en cours de reconnaissance ou prêtes à être réduites. Le • indique où on en est dans la règle.

Conflits

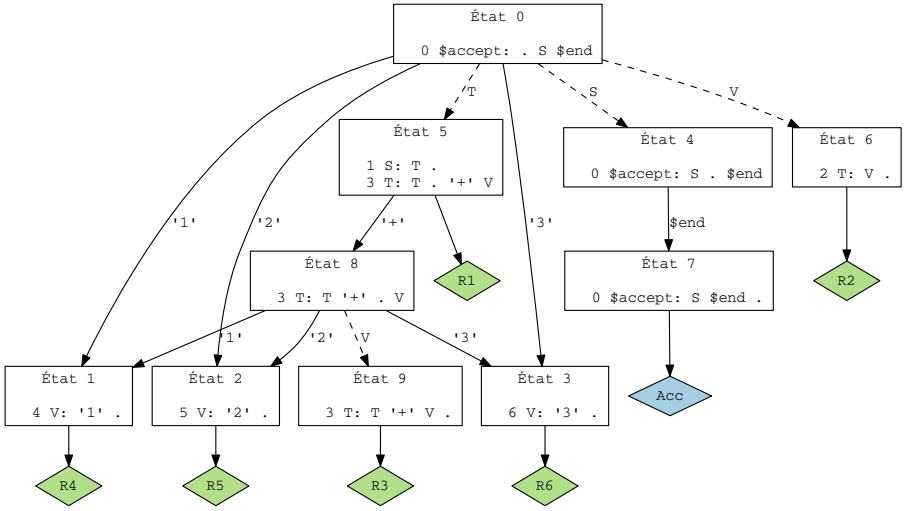
Les analyseurs rendent compte des ambigüités du langage ou de leur faiblesse en levant des conflits:



(1)



(2)



(3)

Figure 8: Transformation de l'automate de la figure 7

shift/reduce Il est levé pour un état et un terminal quand l'analyseur à le choix entre avancer dans une règle (avec ce terminal) ou réduire une règle terminée. Par exemple en LR(0), avec $S \to a\,a | X\,a$; $X \to a$; on aura un état $\boxed{X : a\bullet;\ S : a\bullet a}$.

reduce/reduce Il est levé pour un état si deux règles sont terminées. Par exemple en LR(0), avec $S \to X\,b | Y\,c$; $X \to a$; $Y \to a$; on aura un état $\boxed{X : a\bullet;\ Y : a\bullet}$.

On note que le conflit *shift/shift* n'est pas possible.

En pratique

Même si les approches LR ne supportent pas toutes les grammaires algébriques, elles sont capables d'analyser les langages de programmation

- Sans modifications profondes de leurs grammaires.
  $\Longrightarrow$ pas de perte de lisibilité
- En produisant des AST proches de la grammaire du langage.
  $\Longrightarrow$ utilisables sans modification.
- De plus elles permettent d'ajouter des hooks (ex: pragma de précédence dans bison) qui permettent de "corriger" des grammaires ambigües qui sont en général plus lisible et dont les AST générés sont plus proches du metier du compilateur cible.

$\Longrightarrow$ analyseurs tout à fait utilisables.

2.2.4 Exercices

Exercice 1

Considérons la grammaire G suivante: $\begin{array}{l}\Sigma = \{\ a\ \} \\ N = \{\ L\ \} \\ \mathrm{s} = L\end{array}$ $P = \left\{ \begin{array}{l} L \to L\,a \\ L \to a \end{array} \right\}$

a) Est-elle LL(1)?

b) Est-elle LR(1)?

c) On remplace dans la grammaire G la règle $L \to L\,a$ par $L \to a\,L$.

   i) Cette nouvelle grammaire est elle équivalente à la grammaire G?

   ii) Peut on dans un analyseur utiliser l'une ou l'autre indifféremment?

   iii) Cette grammaire est elle LL(1)?

   iv) Donnez une grammaire équivalente LL(1).

d) On peut rendre récursive à droite n'importe quelle grammaire récursive à gauche. Pour cela il suffit de changer chaque règle S récursive à gauche de la forme

$S \to S\,a_1 \mid S\,a_2 \mid \ldots \mid S\,a_n \mid b_1 \mid b_2 \mid \ldots \mid b_m$

par

$S \to b_1\,S' \mid b_2\,S' \mid \ldots \mid b_m\,S'$

$S' \to a_1\,S' \mid a_2\,S' \mid \ldots \mid a_n\,S'$

   i) Rendez la grammaire G récursive à droite.

   ii) Cette nouvelle grammaire est-elle LL(1)?

   iii) Cette nouvelle grammaire est-elle LR(1)?

   iv) Soit AG l'analyseur LR(1) de la la grammaire G, soit AG' l'analyseur LR(1) de cette nouvelle grammaire. Lequel est le plus performant?

Exercice 2

Donnez la grammaire de listes de listes d'entiers. Les listes sont séparées par un séparateur.

Exercice 3

Définissez une grammaire décrivant des arbres binaires.

Exercice 4

Définissez une grammaire qui décrit un graphe orienté valué.

Exercice 5

Sur la grammaire G (connue sous "dangling else") ci-dessous, un analyseur LR indique un conflit shift/reduce. On supposera que l'analyseur LR choisit le shift sur un tel conflit.

$$\Sigma=\{\ i\ if\ else\ \}$$
$$N=\{\ P\ \}$$
$$s=P$$
$$P=\left\{\begin{array}{l} P\rightarrow i \\ P\rightarrow if\ P \\ P\rightarrow if\ P\ else\ P \end{array}\right\}$$

1. Indiquez pour chacune des entrées suivantes les interprétations possibles et celle que sélectionne l'analyseur LR.

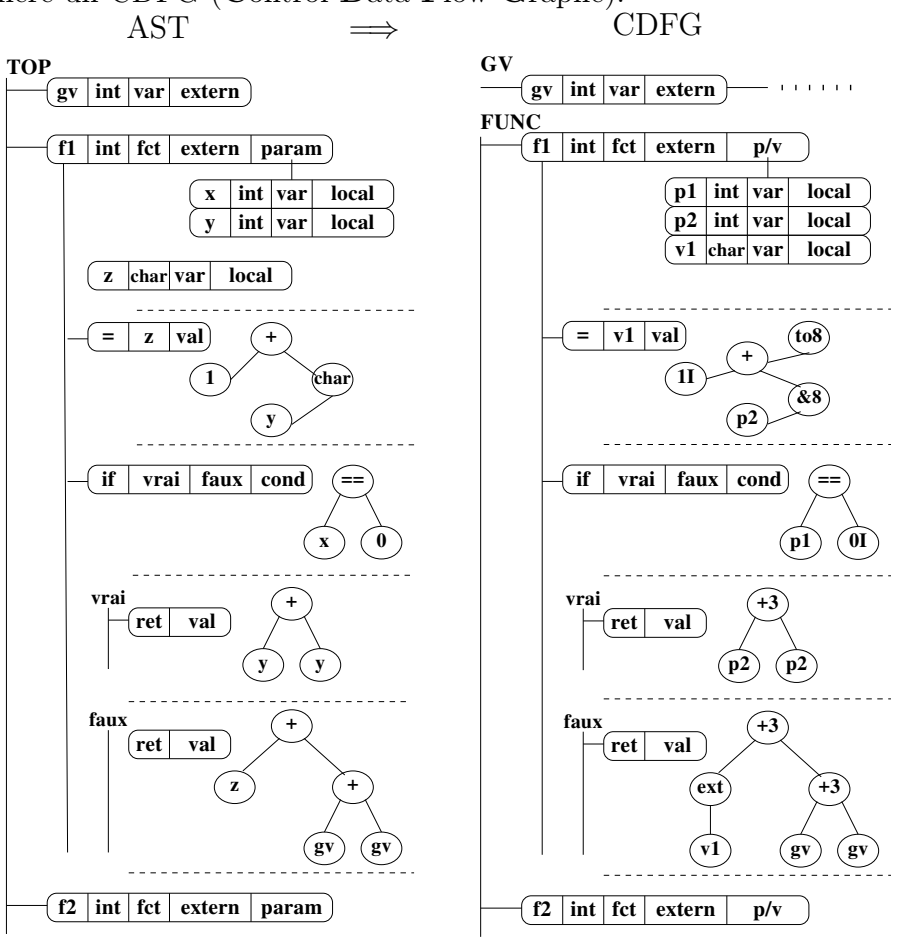
```
IF i ELSE IF I
IF IF i ELSE i
IF IF i ELSE IF IF i ELSE i
```

2. Proposez une grammaire permettant de lever ce conflit.

2.3 Analyse sémantique

2.3.1 Entrée/Sortie

L'analyse sémantique part d'un arbre syntaxique abstrait (AST) et génère un CDFG (**C**ontrol **D**ata **F**low **G**raphe).



Le CDFG n'est généré que si l'AST suit la sémantique du langage source.

2.3.2 Entrée/Sortie

AST n'est qu'une description syntaxique du code source. Au niveau syntaxique "$x = y$;" est correct pour le langage C même si x ou y ne sont pas déclarées.

$\Longrightarrow$ AST est propre à un code dans un langage.

CDFG est une description interne au compilateur du code source. Il est indépendant du langage source. Les opérateurs du CDFG n'ont plus d'ambigüité. Par exemple la sémantique du C (documentation) indique précisément si "$y + z$;" est correcte et quelle opération faire.

- "long x,y;" $\Longrightarrow$ long r=x +long y.
- "char x,y;" $\Longrightarrow$ int r=char2int(x) +int char2int(y).
- "T*x; char y;" $\Longrightarrow$ T* r= x +addr (sizeof(T) *int char2int(y));
- ...

$\Longrightarrow$'+' du langage C donne une petite dizaine d'opérations addition.

2.3.3 Quelques analyses

2.3.3.1 Type

L'analyse sémantique d'un type ajoute à la table des types de l'analyseur une entrée

{nom-type taille-type (n_1, T_1, d_1) (n_2, T_2, d_2) ...}

avec n_i, T_i et d_i le nom, le type et le déplacement en octets depuis le haut du type du $i^{ième}$ champ.

Un type ne génère pas de code directement, ses informations sont utilisées pour les instructions contenant "sizeof(T)", "p.n_i", ...

Pour les langages OO, une classe génère des données (vtable, table des casts dynamiques) et éventuellement du code (constructeurs, destructeur).

2.3.3.2 Élément de tableau

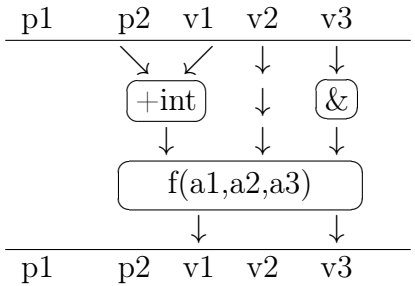
$TAB1 : (t\ expr)$ si "T t[N];" ou "T*t;"

$\Longrightarrow MEM_{SZT}(t\ +_{addr}\ (SZT\ *\ expr))$

$TAB2 : (t\ e_1\ e_2)$

- $T\ t[N][M]$
  $\Longrightarrow MEM_{SZT}(t\ +_{addr}\ (SZT\ *\ (e_1\ +_{int}\ (M\ *\ e_2))))$
- $T\ *^{k+2} t$
  $\Longrightarrow TAB1 : (TAB1 : (t\ e_1)\ e_2)$
  $\Longrightarrow MEM_{SZp^kT}($
  $MEM_{SZp^{k+1}T}(t\ +_{addr}\ (SZp^{k+1}T\ *\ e_1))$
  $+_{addr}\ (SZp^kT\ *\ e_2)$
)

2.3.3.3 Appel de fonction



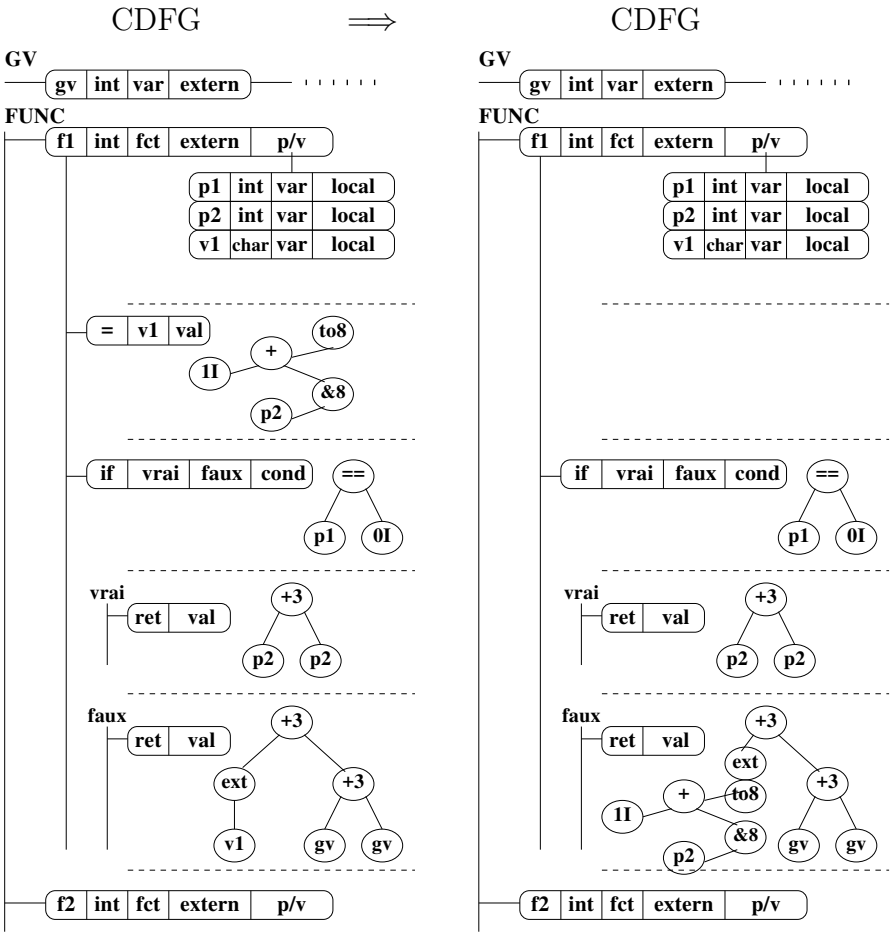
v1 stocke le retour de la fonction et v3 est passé par référence.

2.4 Optimisation générale

2.4.1 Entrée/Sortie

L'optimisation globale effectue des transformations sur le CDFG pour

1. d'abord favoriser la vitesse d'exécution du code cible,
2. ensuite diminuer la taille du code cible.



2.4.2 Principales optimisations

L'optimisation globale est indépendante du langage. La plupart des optimisations sont indépendantes de l'architecture cible. Cependant certaines optimisations sont activées ou désactivées suivant l'architecture cible.

- Propagation des constantes
  $\Longrightarrow$ simplification du CDFG.
- Déplacement de code et suppression des variables
  $\Longrightarrow$ gros DFG,
  $\Longrightarrow$ suppression de jump,
  $\Longrightarrow$ suppression du code redondant.
- Traitement des fonctions
  $\Longrightarrow$ inline (vitesse),
  $\Longrightarrow$ regroupement (taille).
- Traitement des multiplications
  $\Longrightarrow$ remplacement par des additions.
- Calcul formel
  $\Longrightarrow$ remplacement d'un calcul par une formule.
- Reconnaissances de fonctions standard
  $\Longrightarrow$ remplacement par un code optimisé.
- Recherche de parallélisme vectoriel.

2.4.3 Quelques optimisations

2.4.3.1 Propagation des constantes

On propage les constantes puis

- Calcul des expressions constantes ($5I + 4I \Longrightarrow 9I$).
- Simplifications usuelles des expressions ($expr + 0I \Longrightarrow expr$
- Suppression de code mort.
  if x==0 then $bloc_1$ else $bloc_2$ fsi
  Si on trouve "x==0", on peut réduire cette partie du CDFG à $block_1$.

2.4.3.2 Récursivité terminale

Une fonction f est récursive terminale si la dernière instruction exécutée est un appel à f. Dans ce cas, la récusivité peut être remplacée par une boucle qui est un code plus rapide.

```
int fact(int n, int r) {
  if (n<=0)
    return r;
  else
    return fact(n-1,n*r);
}
```

```
int fact(int n, int r) {
  while ( !(n<=0) ) {
    r = n * r;
    n = n - 1;
  }
  return r;
}
```

Cette optimisation est particulièrement utile dans les langages fonctionnels.

2.4.3.3 Suppression de calculs inutiles

L'optimisation détecte certains calculs inutiles comme

```
x = 5;
... // x non utilisé
x = 6;
```

```
x++;
... // x non utilisé
x--;
```

Peut on faire l'optimisation suivante?

```
p = malloc(5);
... // p non utilisé
p = NULL;
```

```
//p = malloc(5);
... // p non utilisé
p = NULL;
```

2.4.3.4 Suppression des fois et jump

Deux instructions sont coûteuses, la multiplication car elle dure au moins 2 cycles et le jump car il introduit au moins une bulle dans le pipeline du processeur cible.

Multiplication Elles apparaissent dans les accès aux tableaux (T[i]).

```
for (i=0; i<n ;i++) {
  T[i] = ...;
}
```

```
for (p=t,pf=p+n ; p<pf
                ; p++) {
  *p = ...;
}
```

Jump L'idée est de déplacer du code pour supprimer des jump.

```
if ( cond )
  x = ...;
else
  x = ...;
return x;
```

```
if ( cond )
  return ...;
else
  return ...;
```

Jump L'idée est d'utiliser le "move conditionnel" si il existe (demi si).

2.4.3.5 Traitements des boucles

- Suppression de calculs redondants.

```
for (i=0 ; i<n ; i++)
  t[i] = i + (x+y+y+5);
```

```
tmp = x+y+y+5;
for (i=0 ; i<n ; i++)
  t[i] = i + tmp;
```

- Déroulement.

```
for (i=0 ; i<2 ; i++) {
  t[i] = i;
}
```

```
t{0] = 0;
t[1] = 1;
// i est supprimée
```

- Calcul formel.

```
for (i=0 ; i<n ; i++)
  s += 1;
```

```
if (n>0 )
  s = n;
```

- Reconnaissance de fonctions standard.

```
for ( i=0 ; i<n ; i++)
  t[i] = 0;
```

MemSet	t	0	n

2.4.4 Barrière d'optimisation

Sur les variables locales l'optimisation maximale est possible. Sur les variables globales l'optimisation doit suivre quelques règles:

Appel fonction code inconnu Ce sont des barrières d'optimisation pour les affectations et les lectures de variables globales.

Appel fonction code connu et n'utilisant pas la variable gv Ce ne sont pas des barrières d'optimisation pour la variable gv.

Appel fonction code connu et utilisant la variable gv Ce sont des barrières d'optimisation pour la variable gv.

Si ces règles ne sont pas suffisantes, les compilateurs définissent en général des builtins pour poser des barrières.

La séquence "p=0x...; *p='h'; *p='e'; *p='l'; ... *p='\n';" est un autre cas où l'optimisation de suppression de code inutile n'est pas souhaitable si p pointe sur le registre d'un contrôleur d'E/S. Les langages ou les compilateurs proposent des directives pour désactiver cette optimisation. Par exemple en langage C, il faut déclarer p de la manière suivante: "volatile char *p;"

2.5 Génération de code

C'est le backend du compilateur. À partir de cette étape, il faut tenir compte des caractéristiques du processeur cible.

- Registres disponibles et fonction des registres.
- Jeu d'instructions (fonction + efficacité).
- Delayed slots

Cependant, si on veut pouvoir supporter différentes architectures cibles (exemple: gcc cible une quarantaine d'architectures) sans avoir à réécrites la majeur partie du backend du compilateur, il faut utiliser une approche systématique.

Cette étape est découpée en 2 sous-étapes majeur

la sélection d'instructions Elle est basée sur un moteur de réécriture. Ce dernier est paramétré par des règles de réécriture spécifiques au processeur cible.

l'allocation de registres Elle est basée sur un moteur d'allocation. Ce dernier est paramétré par des le nombre et les types des registres disponibles sur l'architecture cible.

2.5.1 Projection

Cette phase traduit le graphe de contrôle en un graphe de contrôle adapté à l'assembleur cible (voir la figure 9). Les principales caractéristiques de ce dernier graphe sont:

- Les nœuds opérationnels sont des opérateurs de l'assembleur cible.
- Les variables ne sont pas allouées et notées %0, %1, %2, ...

Cette phase est particulièrement sensible aux options de compilation de bas niveau comme l'option PIC.

2.5.1.1 Principe

La phase prend chaque nœud du graphe de contrôle et le traduit. Ce travail est illustré sur la figure 9 en considérant que l'opérateur "+" du graphe se traduit par l'instruction "add x y" dont la sémantique est "y += x".

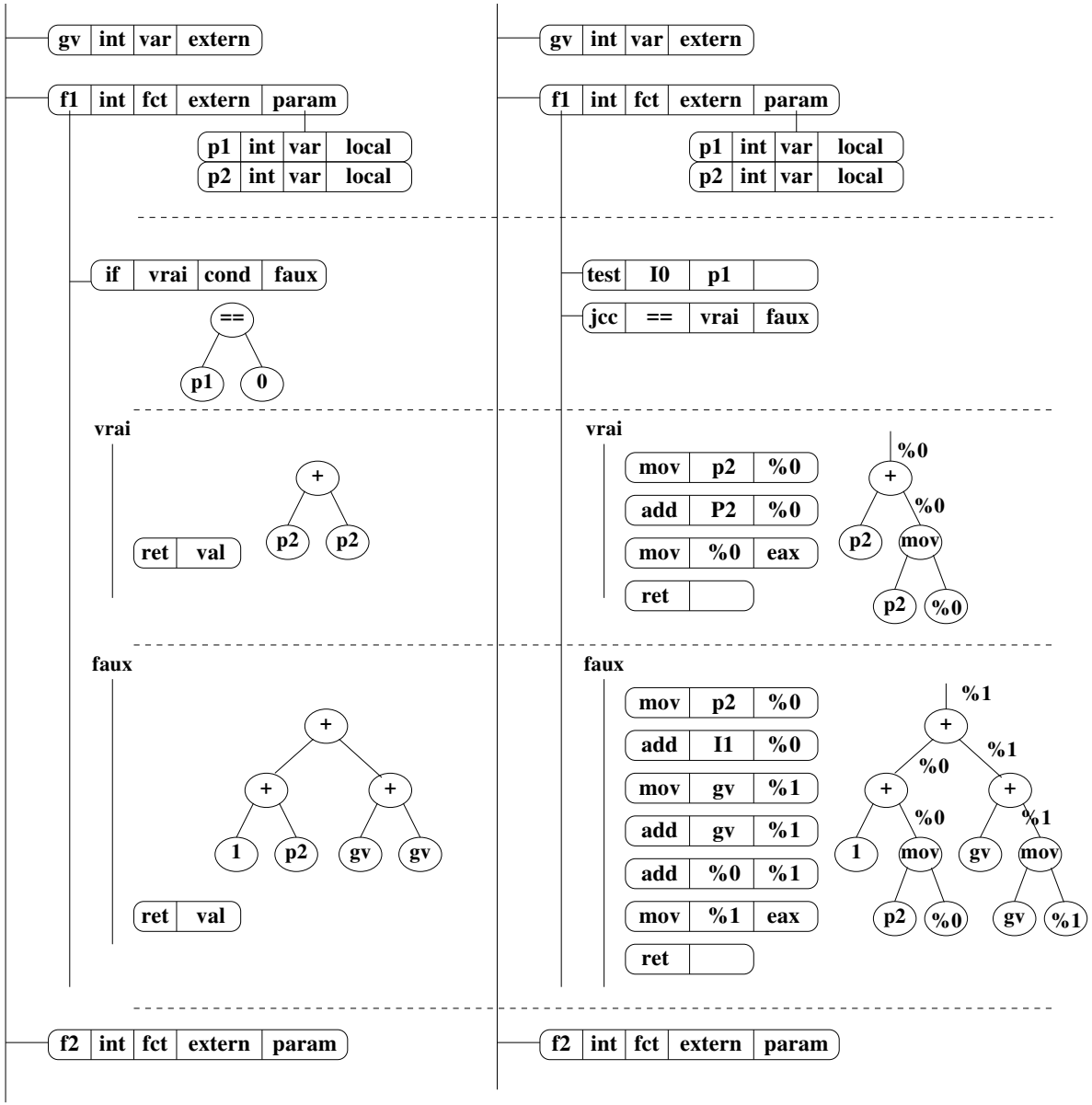


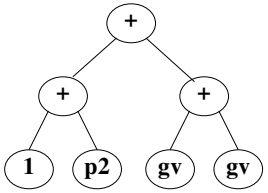
Figure 9: Projection d'un graphe de contrôle sur un assembleur cible.

2.5.1.2 Problème

Ce genre de traduction simple est celle qui est faite sans option d'optimisation par gcc. Elle n'est pas optimale car les processeurs ont plusieurs instructions qui implémentent l'addition.

Si l'architecture est un X86, une projection plus efficace de l'arbre de l'exemple est:

```
1   movl  p1 , %0
2   movl  gv , %1
3   lea   1(%0,%2,2),%1
```



Pour cette phase, le problème principal est:

- une expression arithmétique
- un ensemble d'instructions assembleur

$\Longrightarrow$ une séquence d'instructions assembleur optimale traduisant l'expression.

Un système de réécriture est une solution à ce problème.

2.5.1.3 Système de réécriture

Termes du premier ordre

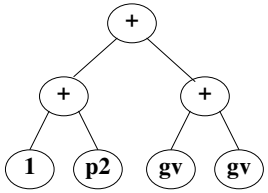
Les objets sont des termes du premier ordre. Un terme est une expression construite de variables et d'opérateurs avec leur arité. Les opérateurs nullaire représentent les constantes. Les représentations classiques et équivalentes sont:

mathématique $(1 + p1) + (gc * gv)$

polonaise $++ +1(p1)(gv)(gv)$

fonctionnelle $plus(plus(1, p1), plus(ev, ev)$

arborescente voir ci-contre



Substitution

Une substitution simple "$\sigma : v \rightarrow t_v$" associe le terme t_v à une variable v. On peut appliquer une substitution "$\sigma : v \rightarrow t_v$ à une terme s (noté $\sigma(s)$) en remplaçant dans s toute les occurrences de la variable v_i par t_i. Par exemple

avec $\sigma : x \rightarrow add(1, x)$

on a $\sigma(add(x, add(y, x))) = add(add(1, x), add(y, add(1, x)))$

Une substitution (générale) est un ensemble de substitutions simples $\{\sigma : v_i \rightarrow t_i\}$ tel que les v_i soient des variables différentes. On peut appliquer cette substitution à une terme s (noté $\sigma(s)$) en remplaçant dans s **uniquement** toute les occurrences des variables v_i par t_i (aucune substitution n'est faite dans les t_i). Par exemple:

$\sigma : \{y \rightarrow add(y, x),\ x \rightarrow add(1, x)\}$

$\sigma(add(x, add(y, x))) = add(add(1, x), add(add(y, x), add(1, x)))$

Règle de réécriture

Une règle "$\rho : l \rightarrow r$" est constituée de deux termes l et g avec (1) l n'est pas une méta-variable, (2) toutes les meta-variables du terme r sont présentes dans le terme l. Voici quelques règles:

valides : 2 → plus(1,1), plus(a,plus(1,a)) → inc(inc(a)).

invalides: x → plus(1,1), plus(a,1) → inc(b).

Soit une règle "$\rho : l \rightarrow r$" et t un terme. Une réécriture "$t \xrightarrow{\rho} t'$" consiste à remplacer une occurrence de l dans t par r pour donner t'. Par exemple considérons la règle "$\rho : plus(1, a) \rightarrow inc(a)$" et le terme t "plus(1,plus(1,inc(x)))". Dans t on a

une $1^{ère}$ occurrence de l : $plus(1, plus(1, inc(x)))$

et une 2^{nde} occurrence de l : $plus(1, plus(1, inc(x)))$

Donc pour réécrire t avec la règle ρ, il faut préciser la position p. Dans notre exemple pour $p1$ et $p2$ étant respectivement la $1^{ère}$ et la 2^{nde} position:

$t \xrightarrow{\rho\ en\ p1} t'_1 = inc(plus(1, inc(x)))$

$t \xrightarrow{\rho\ en\ p2} t'_2 = plus(1, inc(inc(x)))$

De manière plus formelle, $t \xrightarrow{\rho\ en\ p} t'$ si s'il existe une substitution σ telle que $\sigma(t|_p) = l$ et $t' = t[\sigma(r)]_p$ ($t|_p$ est le sous terme de t à la position p, $t[s]_p$ est le terme t dont le sous terme $t|_p$ a été remplacé par le terme s).

Soit la règle $\rho : plus(e, 1) \rightarrow inc(e)$ indiquez quelles sont les réécritures valides:

plus(1,plus(1,1)) $\xrightarrow{\rho}$ plus(1,inc(1))	
plus(1,plus(1,1)) $\xrightarrow{\rho}$ inc(plus(1,1))	
plus(plus(1,1),1) $\xrightarrow{\rho}$ inc(plus(1,1))	

plus(1,x)$\xrightarrow{\rho}$inc(x)	
plus(y,1)$\xrightarrow{\rho}$inc(y)	

Système de réécriture

Un système de réécriture $\mathcal{R}$ est un ensemble de règles de réécriture par exemple:

ρ_1: plus(a,1) $\rightarrow$ inc(a)

ρ_2: plus(a,b) $\rightarrow$ plus(b,a)

Il permet de chainer les substitutions:

plus(1,plus(1,1)) $\xrightarrow{\rho_1}$plus(1,inc(1)) $\xrightarrow{\rho_2}$ plus(inc(1),1) $\xrightarrow{\rho_1}$ inc(inc(1))

Un terme t qui ne peut plus être réécrit par aucune règle du système de réécriture $\mathcal{R}$ est dit en forme normale ou irréductible pour $\mathcal{R}$.

Soit un terme t et $\{t'_0\ t'_1,\ldots\ \}$ l'ensemble des termes irréductibles atteignables à partir de t par une suite de réécritures de $\mathcal{R}$. ou { t } quand t est irréductible. Si le cardinal de cet ensemble est 1 alors son terme unique est appelé le terme canonique ou la forme canonique de t pour le système $\mathcal{R}$.

2.5.1.4 Propriétés

Système de réécriture terminant

Un système de réécriture $\mathcal{R}$ est dit terminant si et seulement si il n'existe pas de suite infinie de réécritures:

$t_0 \underset{\mathcal{R}}{\rightarrow} t_1 \underset{\mathcal{R}}{\rightarrow} t_2 \ldots \underset{\mathcal{R}}{\rightarrow} t_n \underset{\mathcal{R}}{\rightarrow} \ldots$

Une suite finie de réécritures conduit à un terme irréductible.

Système de réécriture confluent

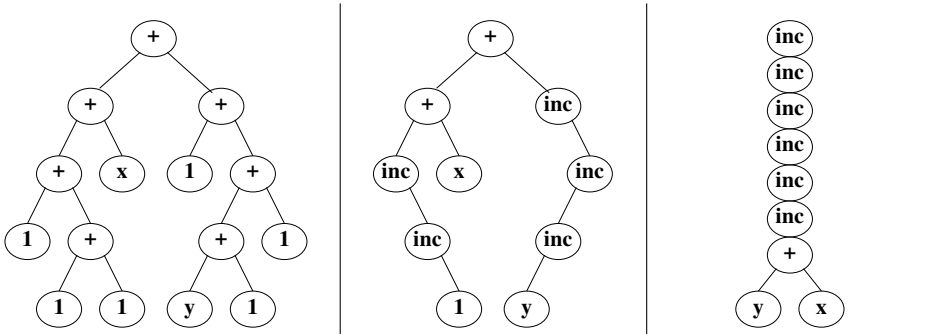
Un système de réécriture $\mathcal{R}$ est dit confluent si et seulement si pout tout terme t, t' et t'':

$t \overset{*}{\underset{\mathcal{R}}{\rightarrow}} t'$ et $t \overset{*}{\underset{\mathcal{R}}{\rightarrow}} t'' \Longrightarrow \exists v,\ t' \overset{*}{\underset{\mathcal{R}}{\rightarrow}} v$ et $t'' \overset{*}{\underset{\mathcal{R}}{\rightarrow}} v$

Si une suite de réécriture partant d'un terme t conduit à un terme irréductible t' alors t' est le terme canonique de t.

Système de réécriture convergent

Un système $\mathcal{R}$ de réécriture est convergent si il est à la fois terminant et confluent alors



(a) terme initial (b) terme solution (c) terme solution optimal

terme software l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), 1 (nullaire) et des variables x, y, z.

terme harware l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), inc (unaire), 1 (nullaire) et des variables x, y, z.

Figure 10: Exemple de projection.

- tout terme t à un terme canonique t',
- et toute suite de réécritures partant de t est finie,
- et toutes ces suites conduisent au terme canonique t'.

Ces caractéristiques font que que ce type de système de réécriture est généralement le paramètre d'un moteur de réécriture.

Quelques exemples

Au départ (voir figure 10.a), on a l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), 1 (nullaire) et des variables x, y, z. Notre architecture nous propose les instructions inc qui ajoute 1 à 1 ou à une variable, et add qui ajoute 2 variables dans une troisième. On ajoute donc l'opérateur unaire inc à nos opérateurs. Notre processeur cible exécute l'instruction inc en 1 cycle et l'instruction add en 4 cycles. Une projection consiste donc à transformer un terme initial (figure 10.a) en un terme solution (figure 10.b) et si possible en un terme solution optimal (figure 10.c).

add(e,1)$\rightarrow$inc(e)	Terminant oui. Confluent oui. Résolvant non.

add(e,1) →inc(e) add(e,e')→add(e',e)	Terminant non. Confluent oui. Résolvant oui. Optimal non car add(add(1,1),add(1,1))..
add(e,1)→inc(e) add(1,e)→inc(e)	Terminant oui. Confluent oui. Résolvant oui. Optimal non car add(add(1,1),add(1,1))..
add(e,1) →inc(e) add(1,e) →inc(e) add(inc(1),inc(1))→inc(add(1, inc(e,1)))	Terminant oui. Confluent oui. Résolvant oui. Optimal non car add(add(e,1),add(1,1))..
add(e,1) →inc(e) add(1,e) →inc(e) add(e,inc(e'))→inc(add(e,e')) add(inc(e),e')→inc(add(e,e'))	Terminant oui. Confluent oui. Résolvant oui. Optimal oui.

2.5.1.5 Un peu de théorie

Terminaison

Montrer qu'un système de réécriture $\mathcal{R}$ est terminant est un problème indécidable. Cependant, il existe de nombreuse méthodes pour prouver que $\mathcal{R}$ est terminant. Beaucoup d'entre elles sont basées sur un ordre de réduction.

Un ordre de réduction est une relation $<$ sur l'ensemble des termes. Dans un ordre de réduction, $t_1 < t_2$ signifie que le terme t_1 est plus simple que le terme t_2. On a alors la propriété:

$\forall\, (l \to r) \in \mathcal{R},\ l > r \Longrightarrow \mathcal{R}$ est terminant.

Un ordre de réduction sur l'ensemble des termes $\mathcal{T}$ du système est un ordre strict qui est:

- monotone pour les opérateurs. $\forall s_1, s_2, ...s_n, u, v \in \mathcal{T}$, $1 \leq i \leq n$, $u < v \Longrightarrow f(s_1, ..., s_{i-1}, u, s_{i+1}, ..., s_n) < f(s_1, ..., s_{i-1}, v, s_{i+1}, ..., s_n)$
- stable par substitution. $\forall u, v \in \mathcal{T}$, $\forall \sigma$ du système, $u < v \Longrightarrow \sigma(u) < \sigma(v)$
- bien fondé ($\forall E \in \mathcal{T}$, E a un plus petit élément).

Soit $\mathcal{P}$ l' ensemble des polynômes à n variables de degré 1 à coefficient dans $\mathcal{N}$ ($P_a = a_0 + \Sigma_{i=1}^n a_i x_i$ avec $a_i \in \mathcal{N}$) .
Soit $<_\mathcal{P}$ la relation de $\mathcal{P}$: $P_a <_\mathcal{P} P_b \iff \forall i \geq 1$, $a_0 <_\mathcal{N} b_0$ et $a_i \leq_\mathcal{N} b_i$.

Irréflexivité ($\forall P \in \mathcal{P}$, $\neg P <_\mathcal{P} P$)
$P_a <_\mathcal{P} P_a \Longrightarrow a_0 <_\mathcal{N} a_0$
Comme $a_0 <_\mathcal{N} a_0$ est impossible dans $\mathcal{N}$, il n'existe donc pas d'élément de $\mathcal{P}$ tel que $P_a <_\mathcal{P} P_a$.

Transitivité ($\forall P_a, P_b, P_c \in \mathcal{P}$, $P_a <_\mathcal{P} P_b$ et $P_b <_\mathcal{P} P_c \Longrightarrow P_a <_\mathcal{P} P_c$)
La transitivité de $<_\mathcal{P}$ se déduit directement de la transitivité de $\mathcal{N}$.

Bien fondé S'il existe une suite décroissante infinie de polynômes $...<_\mathcal{P} P_i <_\mathcal{P} P_{i-1} ... <_\mathcal{P} P_1 <_\mathcal{P} P_0$ alors il existe une suite décroissante infinie de naturels $... <_\mathcal{N} a_0^i <_\mathcal{N} a_0^{i-1} ... <_\mathcal{N} a_0^1 <_\mathcal{N} a_0^0$. Or $(\mathcal{N}, <_\mathcal{N})$ est bien fondé, donc $(\mathcal{P}, <_\mathcal{P})$ l'est aussi.

Stabilité
Soit les polynômes:

$$\begin{array}{lll} P_{e,i}\ (x_1, ..., x_n) & = & e_{i,0} + \Sigma_{j=n}^n e_{i,j} x_j, \\ P_a\ (x_1, ..., x_n) & = & a_0 + \Sigma_{i=1}^n a_i x_i, \\ P'_a\ (x_1, ..., x_n) & = & P_a(P_{e,1}(...), ..., P_{e,i}(...), ..., P_{e,n}(...)) \end{array}$$

On a:

$$\begin{array}{ll} P'_a & = a_0 + \Sigma_{i=1}^n a_i P_{e,i}(x_1, ..., x_n) \\ & = a_0 + \Sigma_{i=1}^n a_i (e_{i,0} + \Sigma_{j=1}^n e_{i,j} x_j) \\ & = a_0 + \Sigma_{i=1}^n a_i e_{i,0} + \Sigma_{i=1}^n \Sigma_{j=1}^n a_i e_{i,j} x_j \\ & = a_0 + \Sigma_{i=1}^n a_i e_{i,0} + \Sigma_{i=1}^n \Sigma_{j=1}^n a_j e_{j,i} x_i \\ & = a_0 + \Sigma_{i=1}^n a_i e_{i,0} + \Sigma_{i=1}^n (\Sigma_{j=1}^n a_j e_{j,i}) x_i \\ P'_b & = b_0 + \Sigma_{i=1}^n b_i e_{i,0} + \Sigma_{i=1}^n (\Sigma_{j=1}^n b_j e_{j,i}) x_i \end{array}$$

Si $P_a <_\mathcal{P} P_b$ alors $\forall i \geq 1$, $a_0 <_\mathcal{N} b_0$ et $a_i \leq_\mathcal{N} b_i$ d'où

$$\begin{array}{lllll} a_0 + \Sigma_{i=1}^n a_i e_{i,0} & <_\mathcal{N} & b_0 + \Sigma_{i=1}^n b_i e_{i,0} & \Longrightarrow a'_0 & <_\mathcal{N}\ b'_0 \\ \Sigma_{j=1}^n a_j e_{j,i} & <_\mathcal{N} & \Sigma_{j=1}^n b_j e_{j,i} & \Longrightarrow a'_i & <_\mathcal{N}\ b'_i \end{array}$$

donc $P_a <_\mathcal{P} P_b \Longrightarrow P'_a <_\mathcal{P} P'_b$.

Figure 11: Démonstration de quelques propriétés de $(\mathcal{P}, <_\mathcal{P})$.

Soit $\mathcal{T}$ l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), inc (unaire), 1 (nullaire) et des variables x, y, z.
Soit $\mathcal{P}$ l' ensemble des polynômes à n variables de degré 1 à coefficient dans $\mathcal{N}$ ($P_a = a_0 + \Sigma_{i=1}^{n} a_i x_i$ avec $a_i \in \mathcal{N}$) .
Soit $<_\mathcal{P}$ la relation de $\mathcal{P}$: $P_a <_\mathcal{P} P_b \iff \forall i \geq 1,\ a_0 <_\mathcal{N} b_0$ et $a_i \leq_\mathcal{N} b_i$.
Soit Φ le morphisme de $\mathcal{T}$ dans $\mathcal{P}$:

Φ(1)	=	0
Φ(inc(e))	=	1 + Φ(e)
Φ(add(e,e'))	=	2 + 2Φ(e)+2Φ(e')
Φ(v)	=	v (si v est une variable)

Soit $<$ la relation de $\mathcal{T}$: $\forall t, s \in \mathcal{T},\ s < t \iff \Phi(s) <_\mathcal{P} \Phi(t)$.

Soit t un terme de $\mathcal{T}$, notons: $\Phi(t) = P_t(x_1, ..., x_b) = t_0 + \Sigma_{i=1}^{n} t_i x_i$
La relation $<$ de $\mathcal{T}$ est monotone si les 3 propopositions suivantes sont vraies pour tous les termes t, a, b tel que a<b.

- **inc(a) < inc(b)** par définition de Φ on a:
  $\Phi(inc(a)) = 1 + \Phi(a) = 1 + a_0 + \Sigma_1^n a_i x_i$
  $\Phi(inc(b)) = 1 + \Phi(b) = 1 + b_0 + \Sigma_1^n b_i x_i$
  Comme a<b on a: $\forall i \geq 1$,
  $a_0 <_\mathcal{N} b_0$ et $a_i \leq_\mathcal{N} b_i \implies (1 + a_0) <_\mathcal{N} (1 + b_0)$ et $a_i \leq_\mathcal{N} b_i$
  $\implies \Phi(inc(a)) <_\mathcal{P} \Phi(inc(b))$
  On a donc bien $inc(a) < inc(b)$.

- **add(a, t) < add(b, t)** par définition de Φ on a:
  $\Phi(add(a,t)) = 2 + 2\Phi(a) + 2\Phi(t) = 2 + 2(a_0 + \Sigma_1^n a_i x_i) + 2(t_0 + \Sigma_1^n t_i x_i)$
  $= 2(1 + a_0 + t_0) + \Sigma_1^n 2(a_i + t_i) x_i$
  $\Phi(add(b,t)) = 2 + 2\Phi(b) + 2\Phi(t) = 2 + 2(b_0 + \Sigma_1^n b_i x_i) + 2(t_0 + \Sigma_1^n t_i x_i)$
  $= 2(1 + b_0 + t_0) + \Sigma_1^n 2(b_i + t_i) x_i$
  Comme a<b on a: $\forall i \geq 1,\ a_0 <_\mathcal{N} b_0$ et $a_i \leq_\mathcal{N} b_i$
  $\implies 2(1 + a_0 + t_0) <_\mathcal{N} 2(1 + b_0 + t_0)$ et $2(a_i + t_i) \leq_\mathcal{N} 2(b_i + t_i)$
  $\implies \Phi(add(a,t)) <_\mathcal{P} \Phi(add(b,t))$
  On a donc bien add(a,t)<add(b,t).

- **add(t, a) < add(t, b)** par définition de Φ on a:
  $\Phi(add(t,a)) = 2 + 2\Phi(t) + 2\Phi(a) = \Phi(add(a,t))$
  $\Phi(add(t,b)) = 2 + 2\Phi(t) + 2\Phi(b) = \Phi(add(b,t))$
  Donc comme précédemment, on a donc bien $add(t,a) < add(t,b)$.

Figure 12: Démonstration de la monotonie de ($\mathcal{T}$,<).

Reprenons notre exemple avec l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), inc (unaire), 1 (nullaire) et des variables x, y, zet des règles de réécriture ci-contre.

(1)	add(1,e)	→ inc(e)
(2)	add(e,1)	→ inc(e)
(3)	add(inc(e),e')	→ inc(add(e,e'))
(4)	add(e,inc(e'))	→ inc(add(e,e'))

Considérons $\mathcal{P}$ l' ensemble des polynômes à n variables de degré 1 à coefficient dans $\mathcal{N}$ ($P_a = a_0 + \Sigma_{i=1}^{n} a_i x_i$ avec $a_i \in \mathcal{N}$) .
On définit alors la relation $<_\mathcal{P}$ entre 2 éléments de $\mathcal{P}$ par:
$P_a <_\mathcal{P} P_b \iff \forall i \geq 1,\ a_0 <_\mathcal{N} b_0$ et $a_i \leq_\mathcal{N} b_i$
$<_\mathcal{P}$ est un ordre strict bien fondé et stable sur $\mathcal{P}$ (voir la démonstration sur la figure 11).

Maintenant, on définit le morphisme Φ de ($\mathcal{T}$,<) dans ($\mathcal{P}$,$<_\mathcal{P}$):

Φ(1)	=	0
Φ(inc(e))	=	1 + Φ(e)
Φ(add(e,e'))	=	2 + 2Φ(e)+2Φ(e')
Φ(v)	=	v (si v est une variable)

avec
$\forall t, s \in \mathcal{T},\ s < t \iff \Phi(s) <_\mathcal{P} \Phi(t)$
Comme $<_\mathcal{P}$ est un ordre strict bien fondé et stable sur $\mathcal{P}$, La relation $<$ est aussi sur un ordre strict bien fondé et stable sur $\mathcal{T}$. De plus La relation $<$ est aussi monotone (voir démonstration sur la figure 12). donc la relation $<$ est un ordre de réduction sur $\mathcal{T}$.

Comparons les membres gauches (l) et droits (r) des règles de réécriture de notre système $\mathcal{R}$.

	l	Φ(l)	?	Φ(r)	r
(1)	add(1,x)	2+2Φ(1)+2Φ(x) 2+2x	>	1+Φ(x) 1+x	inc(x)
(2)	add(x,1)	2+2Φ(x)+2Φ(1) 2+2x	>	1+Φ(x) 1+x	inc(x)
(3)	add(inc(x),y)	2+2Φ(inc(x))+2Φ(y) 2+2(1+x)+2y 4+2x+2y	>	1+Φ(add(x,y)) 1+2+2x+2y 3+2x+2y	inc(add(x,y))
(4)	add(x,inc(y))	2+2Φ(x)+2Φ(inc(y)) 2+2x+2(1+y) 4+2x+2y	>	1+Φ(add(x,y)) 1+2+2x+2y 3+2x+2y	inc(add(x,y))

Comme pour toutes les règles ($\rho : l \to r$), on a l>r, le système de réécriture $\mathcal{R}$ est terminant.

ρ1:$add(x,1) \to inc(1)$ ρ2:$add(1,x) \to inc(1)$	add(1,1)	inc(1) inc(1)
ρ :$add(add(x,1),1)) \to add(x,2)$	add(add(add(x,1),1),1)	add(add(x,2),1) add(add(x,1),2)
ρ1:$f(f(x)) \to f(x)$ ρ2:$f(a) \to b$	f(f(a))	f(a) f(b)
ρ1:$add(add(x,1),1)) \to add(x,2)$ ρ2:$add(x,0) \to x$		

Figure 13: Exemple de paires critiques et de confluence locale

Confluence

Montrer qu'un système de réécriture $\mathcal{R}$ est confluent est un problème indécidable.

On peut cependant montrer la confluence locale. Pour cela, on prend les règles du système $\mathcal{R}$ 2 par deux et on détermine si elles sont en conflit. Si aucun couple de règles n'est en conflit, le système $\mathcal{R}$ est localement confluent.

Si $\mathcal{R}$ est terminant, alors: confluent $\iff$ localement confluent

Soit deux règles de réécriture de $\mathcal{R}$ (pas forcément différentes) $\rho 1 : g \to d$ et $\rho 2 : l \to r$ et un terme t tels que $t \xrightarrow{\rho 1 \ en \ p1} t1'$ et $t \xrightarrow{\rho 2 \ en \ p2} t2'$.

De manière intuitive, elles sont en conflit si p1 est dans l ou p2 est dans g. On appelle $(t1', t2')$ une paire critique, en effet appliquer la $1^{ère}$ réécriture interdit la 2^{nde} et vice versa. On part donc dans des séquences de réécriture différentes.

Cependant si cette paire critique est joignable, c'est à dire qu'il existe u tel que $t1' \xrightarrow[\mathcal{R}]{*} u \xleftarrow[\mathcal{R}]{*} t2'$ alors ces séquences confluent à nouveau.

Donc un système $\mathcal{R}$ conflue localement si toutes ses paires critiques sont joignables. Quelques exemples sont présentés sur la figure 13.

Reprenons notre exemple avec l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), inc (unaire), 1 (nullaire) et des variables x, y, z et des règles de réécriture:

(1)	add(1,e)	$\to$ inc(e)
(2)	add(e,1)	$\to$ inc(e)
(3)	add(inc(e),e')	$\to$ inc(add(e,e'))
(4)	add(e,inc(e'))	$\to$ inc(add(e,e'))

1 & 1 pas de paire critique

1 & 2 t=$add(1,1)$, PC=$(inc(1), inc(1))$, joignable car $inc(1) \xrightarrow[\mathcal{R}]{*} u = inc(1) \xleftarrow[\mathcal{R}]{*} inc(1)$

1 & 3 pas de paire critique

1 & 4 t=$add(1, inc(e))$, PC=$(inc(inc(e)), inc(add(1,e)))$, joignable car $inc(inc(e)) \xrightarrow[\mathcal{R}]{*} u = inc(inc(e)) \xleftarrow[\mathcal{R}]{1} inc(add(1,e)))$

2 & 2 pas de paire critique

2 & 3 t=$add(inc(e),1)$, PC=$(inc(inc(e)), inc(add(e,1)))$, joignable car $inc(inc(e)) \xrightarrow[\mathcal{R}]{*} u = inc(inc(e)) \xleftarrow[\mathcal{R}]{2} inc(add(e,1)))$

2 & 4 pas de paire critique

3 & 3 pas de paire critique

3 & 4 t=$add(inc(e), inc(e')$, PC=$(inc(add(e, inc(e'))), inc(add(inc(e), e')))$, joignable car $inc(add(e, inc(e'))) \xrightarrow[\mathcal{R}]{4}$u=inc(inc(add(e,e'))) $\xleftarrow[\mathcal{R}]{3} inc(add(inc(e), e'))$

4 & 4 pas de paire critique

Notre système de réécriture est donc localement confluent et comme il est terminant, il est convergent.

2.5.2 Allocation des registres

Cette phase (voir la figure 14) prend en entrée un graphe de contrôle d'une fonction au niveau macro assembleur cible mais

- Les variables ne sont pas allouées et notées %0, %1, %2, ...
- Le prologue et l'épilogue de la fonction ne sont pas écrits.

Elle génère le code complet de la fonction:

- À chaque %i est associé soit un registre soit une variable en pile.

Entrée

```
1  exemple:
2    prologue
3    movl %p1      , %0
4    movl %p2      , %1
5    lea  (%0,%1)  , %2
6    movl %p3      , %3
7    addl %0       , %2
8    addl %2       , %3
9    movl %3       , %eax
10   épilogue
11   ret
```

Sortie

```
1  exemple:
2    enter $12,$0
3    pushl %ebx
4    movl 12(%EBP)    , %ebx
5    movl  8(%EBP)    , %ecx
6    lea  (%ecx,%ebx) , %ebx
7    movl 16(%EBP)    , %eax
8    addl  8(%EBP)    , %ebx
9    addl %ebx        , %eax
10   popl %ebx
11   leave
12   ret
```

Figure 14: Entrée/sortie de la phase "allocation de registes".

- Chaque instruction est réécrite en tenant compte de l'affectation.
- Le prologue et l'épilogue sont écrits.
- Le code mort est supprimé.

2.5.2.1 Durée de vie des variables

On calcule pour chaque variable et pour tous les nœuds du graphe son état:

- * La valeur de la variable est utilisée en lecture plus loin dans le graphe.
- R La variable est utilisée en lecture dans le nœud.
- W La variable est écrite dans le nœud.
- "■" La valeur de la variable n'est pas utilisée plus loin.

La figure 15 illustre la durée de vie des variables.

2.5.2.2 Code mort

Les durées de vie des variables permettent de détecter les codes morts du graphe de contrôle.

```
1  exemple:              // |eax|0|1| 2| 3||p1|p2|p3|
2    prologue            // |   | | |  |  ||* |* |* |
3    movl %p1     , %0   // |   |W| |  |  ||R |* |* |
4    movl %p2     , %1   // |   |*|W|  |  ||  |R |* |
5    lea  (%0,%1) , %2   // |   |R|R| W|  ||  |  |* |
6    movl %p3     , %3   // |   |*| |*| W||  |  |R |
7    addl %0      , %2   // |   |R| |RW| *||  |  |  |
8    addl %2      , %3   // |   | | | R|RW||  |  |  |
9    movl %3      , %eax // |W  | | |  | R||  |  |  |
10   épilogue            // |*  | | |  |  ||  |  |  |
11   ret                 // |R  | | |  |  ||  |  |  |
```

Figure 15: Durée de vie des variables.

- Soit v_0, v_1, ... les variables écrites dans le nœud N (ou l'instruction I). L'état de ces variables est W ou WR.
- Soit NS=$\{NS_0, NS_1, \dots\}$ l'ensemble des nœuds successeurs du nœud N dans le graphe de contrôle.
- Soit $ES_{i,j}$ les états des variables v_i dans les nœuds NS_j.

Le nœud N est mort si NS est vide ou si tous les états $ES_{i,j}$ sont soit inutilisés (■), soit écriture (W).

Un nœud mort est donc un nœud dont le résultat (l'écriture) n'est pas utilisé. Un nœud mort peut être supprimé.

Supprimer le nœud mort N peut rendre mort les nœuds prédécesseurs de N.

2.5.2.3 Interférence de variables

La durée de vie des variables permet de déterminer si 2 variables interfèrent, c'est à dire si elles ne peuvent pas être regroupées dans la même location mémoire.

Dans un graphe de contrôle, 2 variables v_0 et v_1 interfèrent si il existe un nœud du graphe où les états E_0 et E_1 des variables sont égaux à une des combinaisons suivantes:

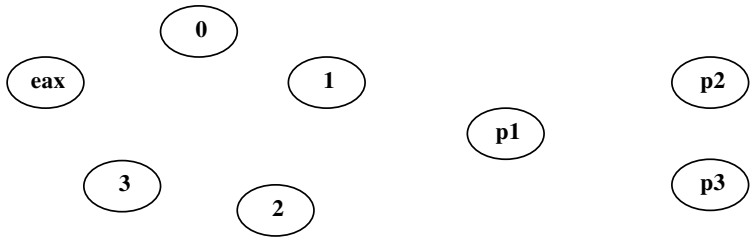


Figure 16: Graphe d'inférence du graphe de contrôle de la figure 15.

*** et *** .....

R et * .....

W et * .....

RW et * .....

les cas symétriques (* et R) et (* et W) et (* et RW)

2.5.2.4 Graphe d'interférence

Le graphe d'interférence des variables est construit de la manière suivante:

- Les nœuds du graphe sont les variables.
- Une arête entre 2 nœuds indique que les variables interfèrent.

Dans la figure 16 construisez le graphe d'interférence du graphe de contrôle de la figure 15.

2.5.2.5 Graphe d'interférence et de préférence

L'ABI de l'assembleur utilisée dédie des registres pour les échanges de valeur entre les fonctions.

- ABI I686: valeur de retour dans `%EAX`
- ABI X86_64 Linux & RISC: premiers paramètres transmis dans des registres.

Dans le code "`movl %0,%EAX; ret`", choisir `%EAX` pour le registre 0, permet de supprimer l'instruction movl.

Dans le graphe de préférence, on ajoute des arêtes de préférence. On les détecte dans les instructions "move r_s, r_d" où un des registres est une pseudo-variable et l'autre un registre spécifié. Cette arête entre les deux nœuds n'est ajoutée que si il n'y a pas déjà un arc d'interférence entre eux.

Dans notre exemple figure 15, on a une telle instructions à la fin (`movl %3,%EAX;`), on ajoute donc une arête de préférence (en pointillé) entre les nœuds 3 et eax sur le graphe d'interférence de la figure 16.

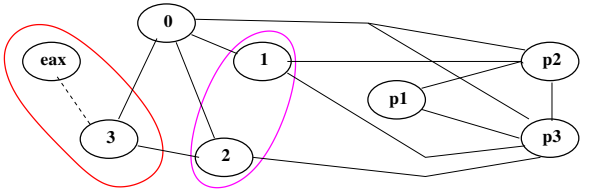
2.5.2.6 Résolution

L'allocation de registres se résout en coloriant le graphe en k couleurs, k étant le nombre de registres disponibles.

- 2 nœuds du graphe peuvent avoir la même couleur si ils n'ont pas d'arête d'interférence.
- Si possible donner la même couleur aux nœuds ayant une arête de préférence.
- Les nœuds de la même couleur partagent le même registre.
- Les nœuds non coloriés devront être mis en variable dans la pile.

Coloriez le graphe de la figure 16 en 3 couleurs (le nombre de registres temporaire pour l'ABI I386 de Linux). On trouve alors l'allocation: (%3) dans %eax ; (%1, %2) dans %ecx ; (%0) dans %edx.

Si on colorie le graphe avec 2 couleurs, on est bloqué à cette étape,



%0 doit donc être stocké dans la pile. Une fois que l'allocation des registres est déterminée, on peut recommencer le coloriage pour diminuer le nombre de variables en pile. Dans l'exemple précédent, on peut regrouper %0 et p1. On n'a donc pas de variable à créer. L'allocattion est: (%1, %3) dans %eax ; (%1, %2) dans %ecx ; (%0) dans %p1.

Au niveau théorique, le coloriage d'un graphe en k couleurs est NP-complet pour k supérieur ou égal à 3. Les algorithmes ne peuvent donc pas garantir une solution optimale mais une solution acceptable.

2.5.2.7 Génération du code

Pour la génération de code, on a le graphe de contrôle initial et la table d'allocation des variables (les %i). Les étapes sont:

Création de la fenêtre de pile On décide ici si on utilise FP ou SP pour accéder aux variables dans la pile. En général:

pas de variables locales on utilise SP pour référencer les paramètres, le prologue et l'épilogue sont vides.

variables locales On crée dans le prologue l'espace dans la pile pour les variables, et on restaure la pile initiale dans l'épilogue. Les variables et les paramètres sont référencés par FP.

Traduction des instructions On prend les instructions une à une et on remplace les %i par le registre donné par l'allocation ou par la référence à la variable donnée dans l'étape précédente. L'instruction peut être supprimée, ou modifiée dans le cas où elle n'est pas supportée.

Exemple 1

En partant du graphe de la figure 15 et de l'allocation sur 3 registres ((%3) dans %eax ; (%1, %2) dans %ecx ; (%0) dans %edx), on obtient:

- Il n'y a pas de variables, on va utiliser SP,
  $\Longrightarrow$ prologue et épilogue vides
  $\Longrightarrow$ %p1 est 4(%ESP), %p2 est 8(%ESP), %p3 est 12(%ESP).
- On remplace les %i par leur allocation dans le source. On n'obtient pas d'instructions invalides, et un "movl %eax,%eax" que l'on peut supprimer.

Le résultat est présenté sur la figure 17.a

Exemple 2

En partant du graphe de la figure 15 et de l'allocation sur 2 registres ((%1, %3) dans %eax ; (%1, %2) dans %ecx ; (%0) dans pile), on obtient le résultat présenté sur la figure 17.

a) (%3) dans %eax ; (%1, %2) dans %ecx ; (%0) dans %edx

```
exemple:
//prologue
  movl 4(%esp)      , %edx
  movl 8(%esp)      , %ecx
  lea  (%edx,%ecx), %ecx
  movl 12(%esp)     , %eax
  addl %edx         , %ecx
  addl %ecx         , %eax
//movl %eax       , %eax
//épilogue
  ret
```

b) (%1, %3) dans %eax ; (%1, %2) dans %ecx ; (%0) dans pile

```
exemple:
  enter $4, $0
??movl  8(%ebp), -4(%ebp)
  movl 12(%ebp), %eax
??lea (-4(%ebp),%eax),%ecx
  movl 16(%ebp), %eax
  addl -4(%ebp), %ecx
  addl %ecx     , %eax
//movl %eax     , %eax
  leave
  ret
```

Figure 17: Deux générations du graphe de contrôle de la figure 15.

- Il y a une variables, on a utilisé FP,
  $\Longrightarrow$ prologue: enter $4,$0
  $\Longrightarrow$ épilogue: leave
  $\Longrightarrow$ %p1 est 8(%EBP), %p2 est 12(%EBP), %p3 est 16(%EBP), %0 est -4(%EBP).
- On remplace les %i par leur allocation dans le graphe. On n'obtient 2 instructions invalides, elles devront être réécrites.

La réécriture des instructions invalides ne pose pas de problème, si on a un registre libre (EDX ici).

`movl 8(%ebp),-4(%ebp)` $\Longrightarrow$ .....

`movl (-4(%ebp),%eax),ecx` $\Longrightarrow$ .....

2.5.3 Exercices

Exercice 1

Soit $\mathcal{R} = \{\ f(x,b) \rightarrow g(x)\ \}$, réécrivez les termes ci-contre (quand c'est possible).

(a) $f(a,b)$
(b) $g(b)$
(c) $f(y,b)$
(d) $f(g(a),c)$
(e) $f(f(x,b),b)$
(f) $f(f(x,y),b)$

Exercice 2

Soit $\mathcal{R} = \{\ f(f(x)) \to f(x),\ f(a) \to b\ \}$, Indiquez si les assertions ci-dessous sont vraies.
$\xrightarrow{+}$ est la fermeture transitive de $\to$. $\xrightarrow{*}$ est la fermeture réflexive et transitive de $\to$.

(a) $f(f(a)) \to f(b)$
(b) $f(f(a)) \to f(a)$
(c) $f(f(a)) \to f(f(a))$
(d) $f(f(a)) \xrightarrow{+} f(f(a))$
(e) $f(f(a)) \xrightarrow{*} f(f(a))$
(f) $f(f(a)) \xrightarrow{+} b$
(g) $f(f(a)) \xrightarrow{*} b$
(h) $f(a) \xrightarrow{*} f(b)$
(i) $f(b) \xrightarrow{*} f(a)$

Exercice 3

Soit $\mathcal{T}$ l'ensemble des termes constitués de l'opérateur + (binaire) et des constantes entières (nullaire) et de variables. On notera les constantes en majuscule et les variables en minuscule.

1. Déterminez un système de réécriture $\mathcal{R}$ qui simplifie les constantes.
2. Le système $\mathcal{R}$ est-il localement confluent?

Exercice 4

Soit un ordre $\succ$ total sur les symboles des opérateurs, appelé précédence. On note $|t|$ la taille d'un terme t.

L'ordre de réduction de Knuth et Bendix ($>_{KBO}$) est défini de la manière suivante:

Soit 2 termes s et t, $s >_{KBO} t$ si:

- pour chaque variable x, le nombre d'occurrences de x dans t est inférieur à celui dans s ;
- **et** si au moins une des conditions ci-dessous est vraie
  - $|s| >_{\mathcal{N}} |t|$ ($>_{\mathcal{N}}$ est l'ordre usuel sur les entiers)
  - $s = f(s_1, ..., s_n)$, $t = g(t_1, ..., t_m)$ et f $\succ$ g ;
  - $s = f(s_1, ..., s_n)$, $t = f(t_1, ..., t_n)$ et il existe i tel que pour tout $j < i$ on a $s_j = t_j$ et $s_i >_{KBO} t_i$

Soit l'ensemble $\mathcal{T}$ des termes constitués des opérateurs add (binaire), inc (unaire), 1 (nullaire) et des variables x, y, z et les règles $\mathcal{R}$ ci-contre.

(1) add(1,e) → inc(e)
(2) add(e,1) → inc(e)
(3) add(inc(e),e') → inc(add(e,e'))
(4) add(e,inc(e')) → inc(add(e,e'))

1. Choisissez une précédence pour l'ordre de réduction de Knuth et Bendix.
2. Prouvez que le système de réécriture $\mathcal{R}$ termine.

Exercice 5

Pour le graphe ci-contre:

1. Donnez les durées de vie des variables.
2. Donnez le graphe d'interférence et de préférence.
3. Donnez une allocation et générez le code.

```
1  exemple:
2    prologue
3    movl %p1      , %0
4    movl %p2      , %1
5    lea  (%0,%1), %2
6    movl %p3      , %3
7    addl %0       , %2
8    movl %3       , %eax
9    épilogue
10   ret
```

3 Lex & Yacc

3.1 Lex

3.1.1 Fonction

Fonction générale En fonction d'un ensemble de (motif,traitement) génère un programme qui:

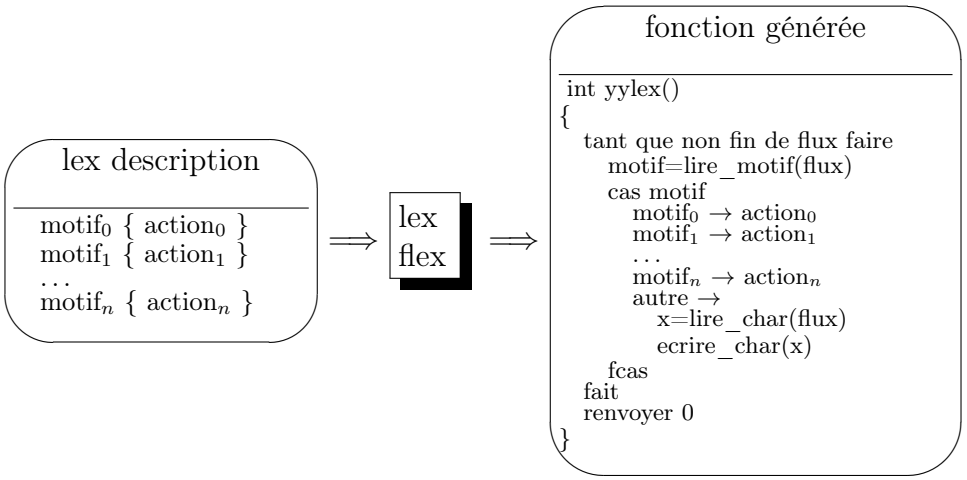
1. lit un flux d'entrée
2. reconnaît les motifs
3. effectue les traitements associés aux motifs

Exemples:

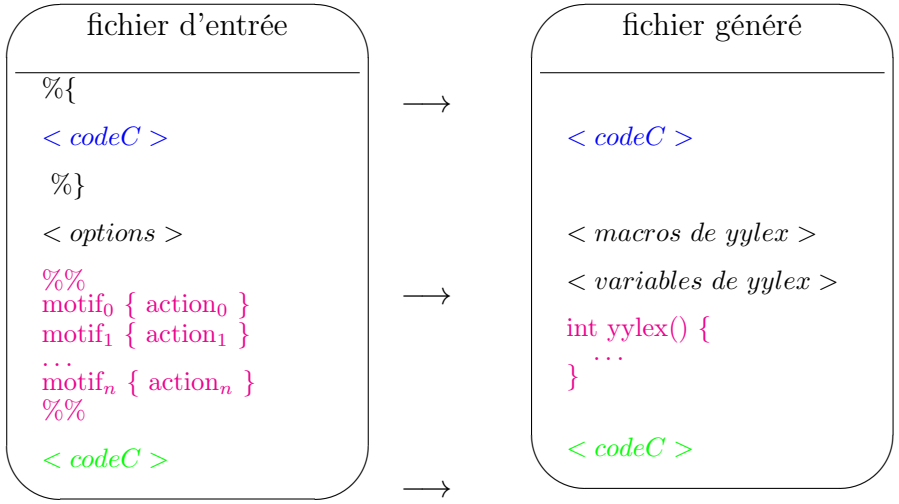
- transformer des fichiers unix en fichiers windows
- transformer tous les caractères d'un fichier en majuscule
- correcteur orthographique

Différence avec sed, perl, ... le traitement n'est pas réduit à une substitution, le filtre est plus rapide.

3.1.2 Principe de lex/flex



3.1.3 Format de lex/flex



Attention: les commandes %... et les motifs doivent commencer en première colonne.

3.1.4 Motifs

Expressions régulières (base)

`x`	match the character 'x'
`.`	any character (byte) except newline
`\X`	if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \X. Otherwise, a literal 'X' (used to escape operators such as '*')
`[xyz]`	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
`[abj-oZ]`	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
`[↑A-Z]`	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

`[↑A-Z\n]`	any character EXCEPT an uppercase letter or a newline
`"[xyz]\"foo"`	the literal string: [xyz]"foo
`\0`	a NUL character (ASCII code 0)
`\123`	the character with octal value 123
`\x2a`	the character with hexadecimal value 2a
`«EOF»`	an end-of-file

Expressions régulières (composition)

`(r)`	match an r; parentheses are used to override precedence
`r*`	zero or more r's, where r is any regular expression
`r+`	one or more r's
`r?`	zero or one r's (that is, "an optional r")
`r{2,5}`	anywhere from two to five r's
`r{2,}`	two or more r's
`r{4}`	exactly 4 r's
`rs`	the regular expression r followed by the regular expression s; called "concatenation"
`r|s`	either an r or an s
`r/s`	an r but only if it is followed by an s.
`↑r`	an r, but only at the beginning of a line.

Caractères à "échapper" + ∗ ? . / etc

Exemples

- un mot
- un mot commençant par une majuscule
- un entier
- une séquence sans blanc ni tabulation.
- un identifiant C
- un commentaire shell

3.1.5 Actions

- pour les sections "codes C" et les actions tout le langage C ou C++.
- la variable **yytext** (char*) contient le motif reconnu.
- la variable **yyleng** (int) contient le nombre de caractères de yytext.
- action par défaut: écriture du motif
- action ";" permet d'ignorer le motif

Attention: yytext est une zone mémoire fixe et donc réécrasée dans chaque action.

3.1.6 Exemple

Source

```
%option noyywrap
%%
[A-Z] { printf("%c",*yytext-'A'+'a'); }
```

Compilation & exécution

```
shell> flex fich.lex
shell> cc lex.yy.c -lfl
shell> ./a.out
abcd<return>
abcd
AbCd<return>
abcd
<CTL-D>
shell>
```

Remarques

- Par défaut le flux d'entrée est le "standard input"
- Par défaut le flux de sortie est le "standard output"
- Par défaut les motifs non reconnus sont écrits sur le flux de sortie.
- libfl.a contient le main

3.1.7 Quelques extensions

3.1.7.1 Commentaires et variables

Les lignes commençant par un blanc ou une tabulation sont copiées telles quelles dans le fichier généré.

```
%option noyywrap

%%
   int cnt=0;
bon  cnt++;
   /* ceci est un commentaire */
(.|\n) ;
«EOF» { printf("%d\n",cnt); return 0; }
```

3.1.7.2 Définition de macro de motifs

```
ENTIER (\+|\-)?[0-9]+
VAR (v|V)ENTIER

%option noyywrap

%%
{ENTIER} ...
{VAR} ...
.|\n ;
```

3.1.7.3 Changement du flux d'entrée

Par défaut lex/flex lit sur yyin et écrit sur yyout. Ce sont des FILE* et ils sont initialisés à stdin et stdout.

```
int main(int argc,char**argv)
{
   if (argc!=3) { ... ; return 1; }
   if ((yyin=fopen(argv[1],"r"))==0) { ... ; return 1; }
   if ((yyout=fopen(argv[2],"w"))==0) { ... ; return 1; }
   yylex();
   return 0;
}
```

De plus lex/flex utilise la macro `"YY_INPUT(buf,result,max_size)"` pour lire le flux d'entrée. Son défaut est:

```
result=fread(buf,1,max_size,yyin)
```

Il suffit de la redéfinir pour lire où on veut.

```
%{
static char* inputbuffer;

#define YY_INPUT(buf,res,nb) \
   if (inputbuffer && *inputbuffer) {\
      *buf=*inputbuffer++; res=1;\
   } else res=0;
%}
```

3.1.7.4 Plusieurs lex

Avec l'option -P de flex

```
bash> flex -Pmonyy fich.l
bash>
```

Le fichier générés s'appelle "lex.monyy.c", la fonction générée s'appelle "monyylex()", les variables des actions "monyytext" et "monyyleng".

Dans fich.l on peut toujours utiliser "yylex()", "yytext", et "yyleng".

Avec les "start-conditions"

```
%option noyywrap

%x code

%%
<INITIAL>BCODE { BEGIN code; printf("%s",yytext); }
<code>ECODE { BEGIN INITIAL; printf("%s",yytext); }
<code>[A-Z] { printf("%c",*yytext-'A'+'a'); }
```

3.2 Yacc

3.2.1 Fonction

Fonction générale

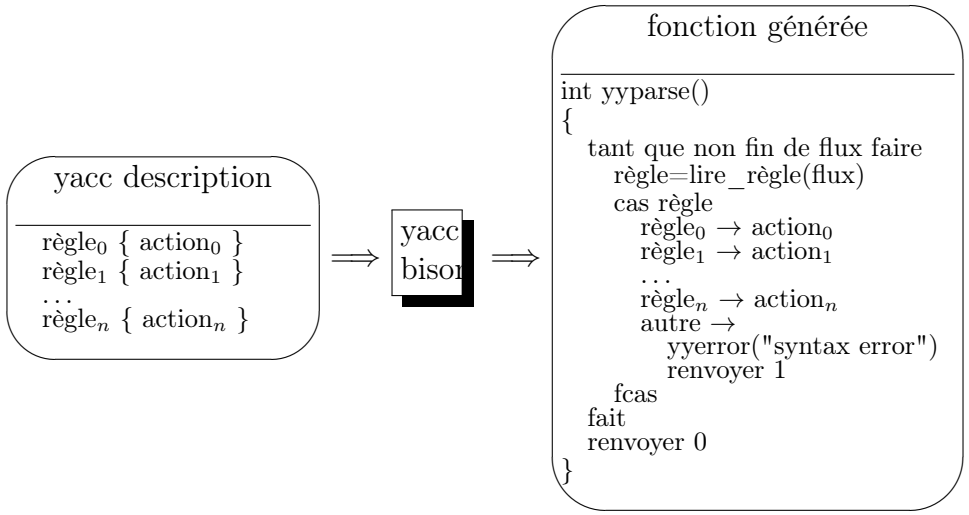
En fonction d'un ensemble de (règle de BNF,traitement) génère un programme qui:

1. lit un flux d'entrée
2. si une règle de BNF est reconnue, effectue le traitement associé
3. si aucune règle n'est reconnue, génère un message d'erreur

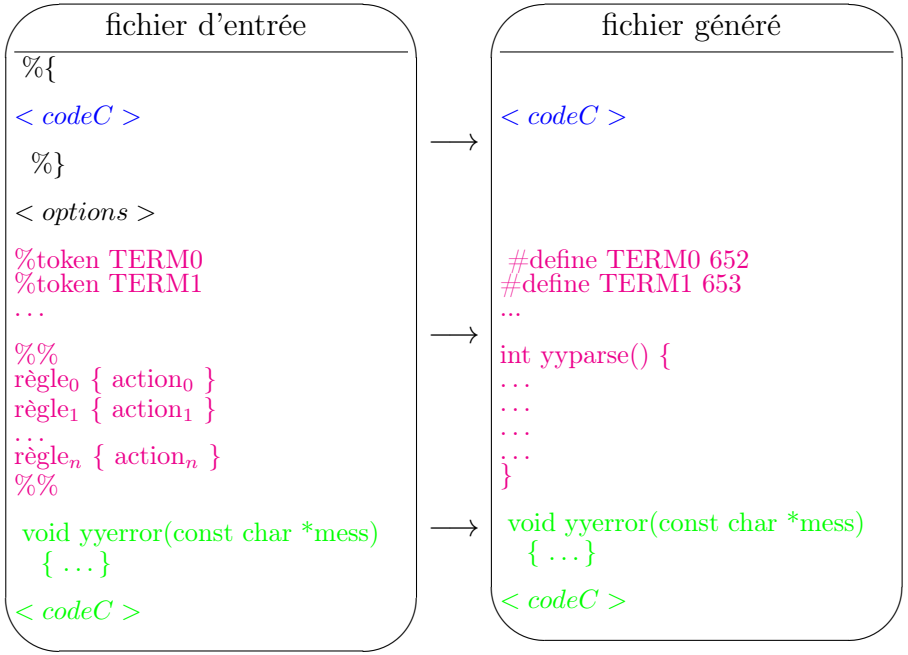
Exemples:

- Un fichier de configuration:
  `config := (variable)*`
  `variable := IDENT EQUAL STRING`
- Langages de programmation usuels.
- Toutes grammaire LR(1).

3.2.2 Principe de yacc/bison



3.2.3 Format de yacc/bison



3.2.4 Syntaxe

3.2.4.1 Syntaxe des règles

Format général

```
< nom regle >
   : < nom_regle_ou_terminal > ...
   | < nom_regle_ou_terminal > ...
   ...
   | < nom_regle_ou_terminal > ...
;
```

Exemple

```
 /* définition des terminaux */
%token ENTIER

/* règle BNF principale */
%start debut

%%
debut: liste_entiers ;
liste_entiers
```

```
    : liste_entiers ENTIER
    | ENTIER
;
%%
```

3.2.4.2 *Syntaxe des actions*

Format général

r : rt_1 rt_2 ... rt_n { < *codeC* > } ...

- tout le langage C ou C++.
- Les éléments d'une règle ont une valeur associée, celle ci peut être référencée par l'opérateur $.
  $i la valeur associée à rt_i (lecture)
  $$ la valeur associée à r (écriture)
- le rôle de l'action est en général de calculer $$ en fonction des $i.
- Les valeurs associées sont par défaut des "int".
- L'action par défaut est: $$=$1
- L'action { } permet de désactiver l'action par défaut.

Exemple

```
/* définition des terminaux */
%token ENTIER

/* règle BNF principale */
%start debut

%%
debut: liste_entiers          { printf("total=%d\n",$1); } ;

liste_entiers
   : liste_entiers ENTIER     { $$=$1+$2; }
   | ENTIER                   { $$=$1; }
;
%%
```

3.2.5 Lecture du flux

flux $(t_0, v_0)(t_1, v_1)...(t_n, v_n)$

acquisition Chaque fois que Yacc/Bison a besoin d'un (t_i, v_i) il appelle la fonction "`int yylex()`".

- elle renvoie l'identifiant du terminal t_i
- elle met dans la variable globale `yylval` la valeur associée v_i.

Le type par défaut de `yylval` est un entier.

Exemple en utilisant lex

```
%%
   /* un entier */
[0-9+] { yylval=atoi(yytext); return ENTIER; }
   /* saute les espaces, les tabs, les LFs */
[\t\n] ;
   /* renvoie un terminal inconnu -> yyerror() */
. { return TK_ERROR; }
```

Remarque Il n'y a aucune obligation d'utiliser lex/flex, "`int yylex()`" peut être écrite à la main.

3.2.6 Extension

3.2.6.1 *Terminal implicite*

Dans une règle: un caractère entre quote est considéré comme un terminal. Le code du caractère est alors le code du terminal.

Yacc

```
%token ENTIER
...
%%
somme
   :  somme '+' ENTIER
   |
   ;
```

Lex

```
%%
\+ { return *yytext; } // ou '+'
...
```

3.2.6.2 *Changer le type des valeurs associées et de yylval*

1. Définition du type dans le préambule

```
%union {
   double    reel;
   char*     str;
   int       entier;
}
```

2. Associer un champ à chaque terminal et règle utilisés dans les actions

```
%token<entier>   NB0
%token<reel>     NB1
%type<reel>      somme
%type<str>       chaine
```

3. Utiliser l'opérateur $ dans les actions, yacc/bison ajoutera automatiquement le nom du champ.

```
%%
somme
   : somme '+' NB0  { $$=$1+$3; }
   | somme '+' NB1  { $$=$1+$3; }
   | chaine         { $$=atoi($1); }
;
...
```

4. Utiliser explicitement le champ dans le lex/flex.

```
%%
[0-9]+           { yylval.entier=atoi(yytext); return NB0; }
[0-9]+\.[0-9]*   { yylval.reel   =atof(yytext); return NB1; }
...
```

Attention: yyparse manipule des piles de yylval $\Longrightarrow$ évitez:

```
%union {
   double reels[1000];
   ...
}
```

3.2.7 Mixer lex/yacc

par inclusion

exemple.y	exemple.l	compilation
%{ ... %} ... %% ... %% #include "lex.yy.c" int yyerror(char*s)	%{ ... %} ... %% ... %% ...	bash> ls exemple.l exemple.y bash> flex exemple.l bash> ls exemple.l exemple.y lex.yy.c bash> bison exemple.y bash> ls exemple.l exemple.tab.c exemple.y lex.yy.c bash> cc exemple.tab.c -lfl bash>

par compilation séparée

exemple.y	exemple.l	compilation

%{
...
%}
...
%%
...
%%
...
	%{
	...
	#include "example.tab.h"
	...
	%}
	...
	%%
	...
	%%
	...
	int yyerror(char*s)
	  ...
	...
		bash> ls
		exemple.l exemple.y
		bash> flex exemple.l
		bash> ls
		exemple.l exemple.y lex.yy.c
		bash> bison -d exemple.y
		bash> ls
		exemple.l exemple.tab.c
		exemple.tab.h exemple.y
		lex.yy.c
		bash> cc exemple.tab.c \\
		    lex.yy.c -lfl
		bash>

3.3 Exemple Complet

Ennoncé

Ecrire un programme qui évalue l'expression booléenne passée dans son premier argument.

L'expression est complètement parenthésée, les terminaux sont:

VRAI FAUX	constantes
& | +	opérateurs binaires: et, ou, ou exclusif
↑	opérateurs unaire: non
( )	parenthèses ouvrante et fermante

Quelques exemples:

VRAI

↑FAUX
((VRAI +FAUX) & ↑(FAUX|VRAI))

Analyse lexicale

```
%{ /* Fichier: expr.l */
#define YY_INPUT(buf,res,nb) \
    if (stream && *stream) {\
           *buf=*stream++; res=1;\
    } else res=0;
static char* stream;
%}

%option noyywrap
...

%%
...

%%

int yyerror(char* m) {
  fprintf(stderr,"%s (near %s)\n", m,yytext);
  exit(1);
}
```

Analyse grammaticale

```
%{
/* Fichier: expr.y */
#include <stdio.h>
static int result;
%}

%token TK_ERROR
%token ...
...

%%
...


%%

#include "lex.yy.c"

int main(int nb, char** arg) {
  int ret;
  if (nb!=2) {
    fprintf(stderr,"usage: %s expr\n",arg[0]);
    return 1;
  }
  stream=argv[1];
  if ( (ret=yyparse)!=0 )
    printf("%s\n",result?"VRAI":"FAUX");
  return ret;
}
```

Compilation & exécution

```
bash> ls
expr.l expr.y
bash> flex expr.l
bash> bison expr.y
bash> ls
expr.l  expr.tab.c  expr.y  lex.yy.c
bash> cc expr.tab.c
bash> ./a.out "VRAI + FAUX"
    syntax error (near +)
bash> ./a.out "(VRAI + FAUX)"
    VRAI
```

```
bash>
```

3.4 Travaux dirigés et pratiques

Exercice 1

Écrivez un filtre:

- qui s'arrête sur une chaîne de caractères incomplète,
- qui extrait et imprime les chaînes de caractères (une par ligne).

Exercice 2

Écrivez un filtre qui enlève les commentaires. Les commentaires sont de la forme:

- # jusqu'à la fin de la ligne (comme en shell),
- commençant par "/*" jusqu'à "*/" (comme en C).

Exercice 3

Écrire un programme qui lit un fichier passé en argument et qui écrit "oui" sur le fichier standard de sortie si il contient 4 lignes consécutives contenant le mot "prof" suivi du mot "eleve". Autrement, il écrit "non".

```
aaa prof aaa eleve aaa\n                          ⟶ oui
aaa\n aaa prof aaa\n\n\n eleve aaa\n aaa\n         ⟶ oui
aaa\n aaa eleve aaa\n\n\n prof aaa\n aaa\n         ⟶ non
aaa\n aaa prof aaa\n\n\n aaa\n eleve aaa\n         ⟶ non
aaa eleve aaa prof aaa\n                          ⟶ non
```

Exercice 4

Considérons le compilateur "/pub/ia/ico/cal1.y", "/pub/ia/ico/cal1.l" qui est basé sur la grammaire ci-dessous:

```
%token NUM

%%
expr
  : NUM
                  | expr '+' expr { $$=$1+$3; ... }
                  | expr '*' expr { $$=$1*$3; ... }
;
%%
```

Q1 Générez le compilateur

```
sh> yacc cal1.y
cal1.y: warning: 4 shift/reduce conflicts
sh> lex cal1.l
sh> gcc y.tab.c
```

Q2 On a 4 conflits. Générons le PDF de l'automate.

```
sh> yacc -g cal1.y
sh> dot -Tpdf cal1.dot > cal1.pdf
sh> xpdf cal.pdf
```

Les conflits se trouvent dans les états 6 et 7 pour '+' et '*' (losange rouge). Dans ce cas, l'analyseur préfère avancer (shift).

Q3 Comme l'analyseur avance, "2 * 3 + 4 + 5" doit être analysé par 2*(3+(4+5)). Vérifions:

```
sh> echo "2 * 3 + 4 + 5" ¦ ./a.out
```

Q4 Yacc/Bison permet de corriger ce type de grammaire ambigüe. Pour cela il suffit de préciser dans le préambule avant le %% l'associativité (gauche ou droite) et la priorité des opérateurs.

pragma	interprétation	pragma	interprétation
%left '+'	$a+b+c \to (a+b)+c$	%right '+'	$a+b+c \to a+(b+c)$
%left '+' '*'	$a+*b+c*d \to ((a*b)+c)*d$		
%right '+' '*'	$a+*b+c*d \to a*(b+(c*d))$		
%right '*' %left '+'	L'ordre de déclaration donne l'ordre de priorité. Ici '*' est moins prioritaire que '+' car déclaré après. $a*b+c+d*e \to a*(((b+c)+d)*e)$		

Modifiez cal1.y pour donner la priorité de '*' sur '+' et rendre ces opérateurs associatif à gauche puis vérifiez sur $1*2*3+4+5*6*7$

Q5 Modifiez la grammaire pour accepter les expressions parenthésées.

Exercice 5

Considérons le compilateur "/pub/ia/ico/cal2.y", "/pub/ia/ico/cal2.l" qui est basé sur la grammaire et l'analyseur lexical ci-dessous:

```
%token NUM
%left '+' '-'
%%
        too:  expr { ... } ;
expr: NUM
    | expr '+' expr { ... }
    | expr '-' expr { ... }
;
```

```
%%
[+\-][0-9]+ { ... }
[ \t\n]     ;
.           {     }
%%
```

Q1 Générez le compilateur et testez le

```
sh> bison cal2.y
sh> flex cal2.l
sh> gcc cal2.tab.c
sh> echo "2 + 3 - 4" | ./a.out 1
sh> echo "2+3-4" | ./a.out FATAL: syntax error (near +3)
```

Q2 Pour corriger ce dysfonctionnement on peut enlever le [+]? dans cal2.l ainsi l'analyse lexicale renvoie que des nombres positifs.

```
sh> flex cal2.l
sh> gcc cal2.tab.c
sh> echo "2 + 3 - 4" | ./a.out 1
sh> echo "2+3-4" | ./a.out 1
```

Q3 Mais maintenant, on a:

```
sh> echo "2+3+ -4" | ./a.out FATAL: syntax error (near -)
```

Pour corriger ce problème ajouter dans la syntaxe les règles:

: expr : '+' expr

: expr : '-' expr

Régénérer le compilateur puis tester le avec "2+3+ -4".

Q4 Essayez "1+++—1", ca doit fonctionner.

4 Projet

4.1 Sujet

L'objectif de ce projet est la réalisation d'un compilateur *labgen* qui lit une description de labyrinthe. Il génère le compilateur associé au labyrinthe *labres* qui lit un chemin et indique si il mène à la sortie.

4.1.1 *labgen*

Le compilateur *labgen* a ∅, 1 ou 2 arguments.

labgen Avec ∅ argument, il lit la description du labyrinthe dans le flux standard d'entrée et génère l'exécutable *labres*.

***labgen* if** Avec 1 argument, il lit la description du labyrinthe dans le fichier **if** et génère l'exécutable *labres*.

***labgen* if exe** Avec 2 arguments, il lit la description du labyrinthe dans le fichier **if** et génère l'exécutable **exe**.

Si l'argument **if** est "-" alors la description du labyrinthe est lue dans le flux standard d'entrée.

De plus si le labyrinthe n'a pas de chemin vers la sortie, il ne génère pas d'exécutable mais écrit sur le le flux standard d'erreur un message indiquant que c'est un labyrinthe sans solution.

4.1.2 *labres*

Le compilateur *labres* généré a ∅ ou 1 argument.

labres Avec ∅ argument, il lit un chemin dans le flux standard d'entrée.

***labres* if** Avec 1 argument, il lit un chemin dans le fichier *if*.

Le comportement du compilateur *labres* généré est le suivant:

le chemin est correct et termine sur une sortie Il écrit "gagné" sur le flux standard de sortie.

le chemin est syntaxiquement incorrecte Il écrit "syntax error" sur le flux standard d'erreur.

autres cas Il écrit "perdu" sur le le flux standard de sortie.

Pour les messages "perdu" et "syntax error", il indiquera aussi la ligne et le dernier terminal lu.

Enfin les compilateurs *labgen* et *labres* doivent fonctionner sur toutes les machines Unix et en particulier sur celles de l'école.

4.2 Description du labyrinthe

Un labyrinthe est une matrice à 2 dimensions, une case étant soit un mur (M), soit une case sur laquelle on peut se déplacer, soit un trou de vers (chiffre), soit une porte magique (lettre minuscule), soit l'entrée (E), soit une sortie (S).

Une description de labyrinthe est une suite d'instructions.

La figure 4.2 donne quelques labyrinthes et leurs descriptions.

4.2.1 Terminaux et commentaires

Tous les caractères qui sont précédés du caractère "#" sur la même ligne sont ignorés.

Dans cette description nous utilisons les mnémoniques suivant:

IDENT Identifiant. Il est constitué d'une lettre suivie de ∅ ou plusieurs lettres ou de chiffres.

CNUM Constante numérique entière de 32 bits. Le format est un signe optionnel, suivi de chiffres décimaux (ex: 67, +67, -67, 0, 000, -0123).

DIR Indique une direction, les valeurs possibles sont: N (nord), NE (nord-est), E (est), SE (sud-est), S (sud), SW (sud-ouest), W (ouest), BW (nord-ouest).

<**expr**> Constante étendue. Ce sont des expressions arithmétiques éventuellement parenthésées formées avec les opérateurs +, -, *, /, % et des constantes numériques (CNUM) et/ou des variables (IDENT). Voici des exemples de constantes: bee+12, (2*(gnu+-1)*3), a-+40, a - +40

Labyrinthe 1

```
    0   1   2
 +---+---+---+
0| E |   |   |
 +---+---+---+
1|   | * |   |
 +---+---+---+
2|   |   | O |
 +---+---+---+
```

Description 1

```
max = 2;
SIZE max;
IN (0,0);
OUT (max,max) ;
WALL PTA (1,1) ;
```

Description 2

```
SIZE 2,2;
IN  (0,0);
WALL ;
TOGGLE R (0,0) (2,2);
OUT (2,2) ;
```

Chemin 1:
S S E E

Chemin 2:
E SE S W NW N
E E S S

Labyrinthe 2

```
    0   1   2   3   4   5
 +---+---+---+---+---+---+
0| a6|   | E | * |   | O |
 +---+---+---+---+---+---+
1| W0| * |   | * |   |   |
 +---+---+---+---+---+---+
2| * | * |   | A | * | a2|
 +---+---+---+---+---+---+
3| w0| * |   |   | * |   |
 +---+---+---+---+---+---+
4|   | * | * | W1|   | * |
 +---+---+---+---+---+---+
5| O |   | w1| * | * |   |
 +---+---+---+---+---+---+
6|   |   |   |   |   |   |
 +---+---+---+---+---+---+
```

Description

```
max = 5;
SIZE max,max+1;
IN (2,0);
OUT (0,max) (max,0) ;
# les murs
WALL PTA (3,0) (3,1) (4,2)
        (4,3) (max,max-1);
WALL PTD (0,2) (1,-1) (0,1):3
         (1,0) (1,+1) (1,0);
# les trous de vers
WH (3,4) --> (2,max) ; # W0 w0
WH (0,1) --> (0,3)   ; # W1 w1
# la porte magique (A et ai)
MD (3,2) W (0,0) E (max,2) ;
```

Chemin 1: S SE E N N
Chemin 2: S SE W S S S

Figure 18: Exemples de labyrinthes

<xcst> Constante étendue. Ce sont des expressions arithmétiques (<expr>) dont toutes les variables sont définies. Elles donnent donc une constante.

<pt> Ce mnémonique définit un point. Son format est: "(<xcst$_x$> , <xcst$_y$>)".

<range> Ce mnémonique définit un ensemble de constantes numériques ses formats sont:

[<xcst$_d$>:<xcst$_f$>:<xcst$_i$>]
les $x_k = xcst_d + xcst_i * k$ tel que $xcst_d \leq x_k \leq xcst_f$.

[<xcst$_d$>:<xcst$_f$>:<xcst$_i$>[
les $x_k = xcst_d + xcst_i * k$ tel que $xcst_d \leq x_k < xcst_f$.

":<xcst$_i$>" est optionnel, sa valeur de défaut est 1. De plus il doit être strictement positif.

Enfin les espaces, les tabulations, les caractères de saut et retour de ligne sont ignorés.

4.2.2 Instructions générales

; Cette instruction ne fait rien.

<nom-var> = <xcst> ; Cette instruction affecte la valeur <xcst> à la variable "nom-var", "nom-var" étant un IDENT.

SIZE <xcst$_1$> ;
SIZE <xcst$_1$> , <xcst$_2$> ; Cette instruction donne la taille du labyrinthe comme une matrice de <xcst$_2$>+1 lignes et <xcst$_1$>+1 colonnes. Les lignes (colonnes) sont numérotées à partir de 0et vont du haut vers le bas (de gauche à droite).
. Si "xcst$_2$" est omis alors sa valeur par défaut est "xcst$_1$".

IN <pt> ; Cette instruction donne les coordonnées de l'entrée du labyrinthe.

OUT <pt$_0$> <pt$_1$> ...; Cette instruction donne les coordonnées des sorties du labyrinthe. <pt$_0$> est obligatoire.

SHOW Cette instruction imprime sur le flux standard de sortie la matrice du labyrinthe construite par les instructions précédentes.

IDENT op= <xcst> ; Cette instruction où op est un opérateur arithmétique binaire (+, - , *, /, %) est l'abréviation de
IDENT = IDENT op (<xcst>) ;

4.2.3 Instructions de tracé

Les instructions suivantes permettent de placer les murs, elles n'affectent pas l'entrée, ni les sorties, ni les trous de vers ni les portes magiques.

WALL ;
Cette instruction place un mur sur toutes les cases de la matrice.

WALL PTA <pt$_0$> <pt$_1$> <pt$_2$> ...;
Cette instruction place un mur sur les cases de la matrice de coordonnées pt$_i$. Le premier point est obligatoire.

WALL PTD <pt$_0$> <pt$_1$>:r_1 <pt$_2$>:r_2 ...;
Les ":r_i" sont optionnels. Sans ":r$_i$" cette instruction place un mur sur les cases de la matrice de coordonnées
pt_0, $pt_0 + pt_1$, $pt_0 + pt_1 + pt_2$, ...
Le premier point est obligatoire.

r_i est soit un <xcst> soit le caractère '*'. ":< $xcst$ >" indique de répéter le déplacement précédent <xcst> fois. ":*" indique de répéter le déplacement jusqu'à atteindre un bord du labyrinthe.
WALL PTD (0,0) (1,-1):2; $\Longrightarrow$ WALL PTD (0,0) (1,-1) (1,-1);
WALL PTD (0,0) (1,0):* (0,1):* (-1,0):* (0,-1):*; $\Longrightarrow$ les bords
Enfin le déplacement (0,0) est interdit.
WALL PTD (0,0) (1,0) (0,1); $\Longrightarrow$ valide
WALL PTD (0,0) (1,0) (0,0); $\Longrightarrow$ invalide

WALL R <pt$_0$> <pt$_1$> ;
WALL R F <pt$_0$> <pt$_1$> ;
Cette instruction place un mur sur le rectangle dont une diagonale est le segment allant de pt$_0$ à pt$_1$. Le F indique que l'intérieur du rectangle est aussi muré.

WALL FOR $v_1\ v_2 \ldots v_n$ **IN** $r_1\ r_2 \ldots\ r_n$ **(<expr$_x$>,<expr$_y$>) ;**
Cette instruction génère "WALL PTA <pt>" pour toutes les valeurs des variables v_i comprises dans les intervalles r_i.

Ainsi l'instruction suivante remplit un rectangle 6x3 en (0,1).

FOR i j IN [0:5] [1:3] (i,j);

Celle-ci un segment horizontal en pointillé.

FOR i IN [1:5:2] (i,4) ; Les variables de boucles (v_i) doivent être différentes et sont locales à la boucle.

Les mêmes instructions existent avec UNWALL à la place de WALL qui démurent les cases.

Les mêmes instructions existent aussi avec TOGGLE à la place de WALL qui démurent les cases si elles sont murées et les murent si elles sont démurées.

4.2.4 Instructions de passages secrets

Les instructions suivantes permettent de placer des trous de vers et des portes magiques, elles n'affectent pas l'entrée, ni les sorties.

WH <pt$_0$> –> <pt$_1$> –> <pt$_2$> . . . ;

Cette instruction indique un trou de vers entre pt_0 et pt_1, entre pt_1 et pt_1, ... L'instruction minimale est "[WH <pt_0> –> <pt_1>;". Le comportement d'un trou de vers d'un point pt_1 vers le point pt_2 est que le simple fait d'arriver sur le point pt_1 nous téléporte sur le point pt_2.

MD <pt> <dest-list>

Cette instruction indique une porte magique, "dest-list" est une suite de "DIR$_i$ <pt_i>". Ce denier indique que prendre la direction DIR$_i$ sur la case pt téléporte sur la case pt_i.

Dans cette liste, chaque direction ne peut être présente qu'une seule fois.

4.2.5 Règles syntaxiques et/ou sémantiques

RS-1 Le labyrinthe doit avoir au moins 2 lignes et au moins 2 colonnes.

RS-2 Le labyrinthe doit avoir une et une seule entrée. Elle ne peut pas être ni une sortie ni une entrée d'un trou de vers.

RS-3 Le labyrinthe doit avoir au moins une sortie. Elles ne peuvent pas être ni l'entrée ni une entrée d'un trou de vers.

RS-4 L'entrée et les sorties du labyrinthe doivent se situer sur la périphérie du labyrinthe (x=0 ou y=0 ou x=COLONNE$_{max}$ ou y=LIGNE$_{max}$)

RS-5 Tant que la taille du labyrinthe n'est pas définie, il est interdit:

- De définir l'entrée ou des sorties.
- D'utiliser des instructions de tracé.
- De définir des portes magiques ou des trous de vers.

RS-6 L'entrée et la sortie des trous de vers doivent être dans le labyrinthe.

RS-7 L'entrée et les sorties des portes magiques doivent être dans le labyrinthe.

RS-8 Pour les tracés de murs (WALL, UNWALL et TOGGLE)

- les points définis par les instructions PTA, PTD et R doivent être dans le labyrinthe.
- les points définis par les instructions FOR en dehors du labyrinthe sont ignorés.
- Si une instruction génère plusieurs occurrences du même point, celui-ci n'est considéré qu'une seule fois.

  TOGGLE PTD (0,0) (1,0) (-1,0); $\Longleftrightarrow$ TOGGLE PTD (0,0) (1,0);

RS-9 Une case ne peut pas être

- l'entrée d'un trou de vers et d'une porte magiques,
- l'entrée de 2 trous de vers,
- l'entrée de 2 portes magiques.

RS-10 Il ne doit pas y avoir de boucle infinie dans les trous de vers.

WH (0,0) –> (0,1) –> (0,0); # ou
WH (0,1) –> (0,0); WH (0,0) –> (0,1);

RS-11 Les instructions de définition de l'entrée, des sorties, des trous de vers et des portes magiques sont interprétées dans n'importe quel ordre après la dernière instruction de tracé.
Si une entrée ou une sortie de ces instructions est un mur, le mur est supprimé avec un message d'attention (warning).

4.3 Description des chemins

labres lit des chemins. Un chemin est une séquence de directions: N (nord), NE (nord-est), E (est), SE (sud-est), S (sud), SW (sud-ouest), W (ouest), NW (nord-ouest).
Le compilateur ignore les espaces, les tabulations, les caractères de saut et retour de ligne. Le chemin "NESW" sera interprèté par "NS SW" et non "N E S W".
Enfin, tous les caractères qui sont précédés d'un caractère "#" sur la même ligne sont ignorés.

4.4 Éléments à rendre

Livrable 1

Une archive tar à déposer dans le dépôt ico-l1 avant le 14/12/17 avec une soutenance pendant le TP du 15/12/17. Elle contiendra:

nom/labgen Les sources de l'analyseur syntaxique de labgen. Il écrira soit rien si la description de labyrinthe est syntaxiquement correcte soit "file:line: syntax error (near symbole)" dans le cas contraire. Aucun autre traitement n'est demandé.
Une commande shell "cmd" (ou un makefile) qui génère l'analyseur.

nom/res Les sources écrits à la main des compilateurs "labres" ex1 et ex2 des labyrinthes ci-dessous et une commande shell "cmd" qui génère ex1.

E		
	*	
		S

exemple 1

E		*
	*	
*		S

exemple 2

Pour l'exemple 2, bison doit échouer (renvoyer un statut non nul).
On peut se limiter à une rose des vents à 4 directions.
Le répertoire contiendra aussi au moins 6 fichiers chemins significatifs pour le labyrinthe ex1: perdu_1.dat perdu_1.dat gagne_1.dat gagne_2.dat serror_1.dat serror_2.dat

Livrable 2

Une archive tar à déposer dans le dépôt ico-l2 avant le 20/12/18 avec une soutenance soit pendant le TP du 12/01/18 (bonus), soit pendant le TP du 19/01/18. Elle contiendra:

nom/labgen Les sources du compilateur labgen.
Une commande shell "cmd" (ou un makefile) qui génère le compilateur.

nom/test Des labyrinthes de test.

4.5 Barème indicatif

Livrable 1: analyseur syntaxique de *labgen* (5 points)

- (∼ 4 points) syntaxe.
- (∼ 1 points) messages corrects et conformes.

Livrable 1: analyseur *labres* (2 points)

Livrable 2: *labgen* complet (13 points)

- (∼ 2 points) Le compilateur traite les variables et l'instruction SHOW sur des labyrinthes simples (similaires à labyrinthe 1 de la figure 4.2).
- (∼ 2 points) Le compilateur fonctionne sur des labyrinthes simples (similaires à labyrinthe 1 de la figure 4.2).
- (∼ 2 points) Le compilateur accepte et traite les trous de vers sur tout labyrinthe sans FOR.
- (∼ 2 points) Le compilateur accepte et traite les portes magiques. sur tout labyrinthe sans FOR.
- (∼ 3 points) Le compilateur accepte et traite les FOR et donc tout labyrinthe.
- (∼ 2 points) Divers.