Génération de code

Brique ASC

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

Plan

Code

Définition

Nous utiliserons la définition suivante de **code** :

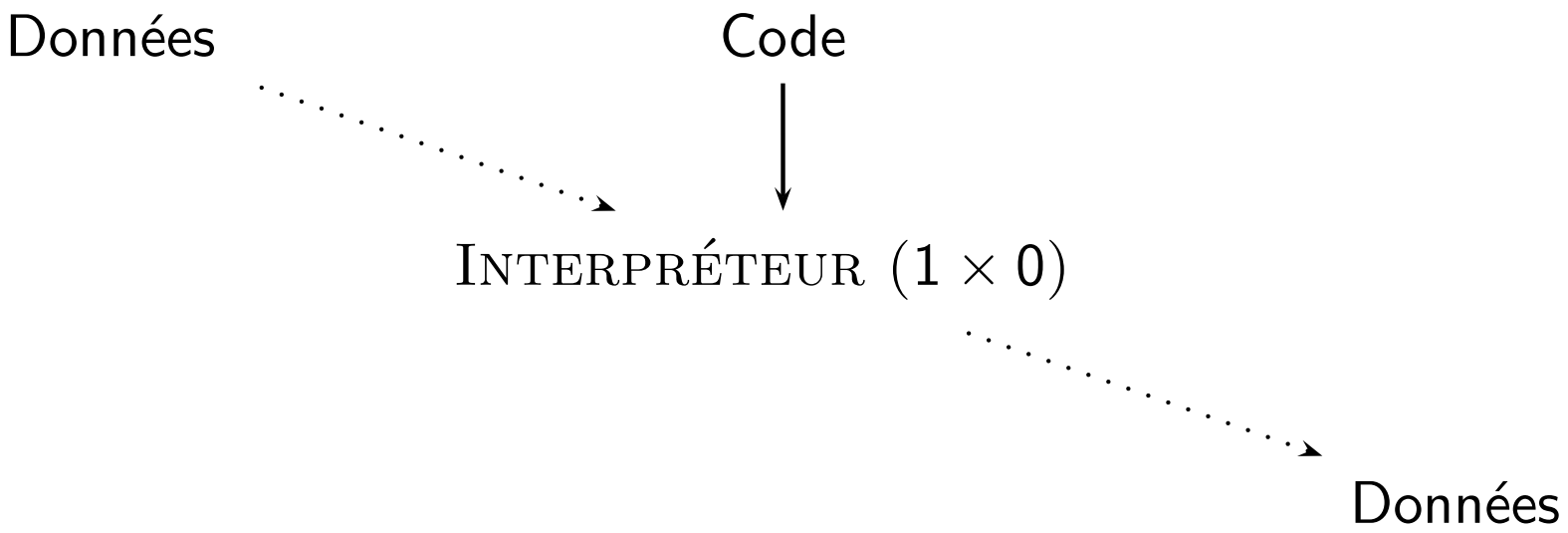
Suite d'instructions destinée à un ordinateur

Exemples de code :

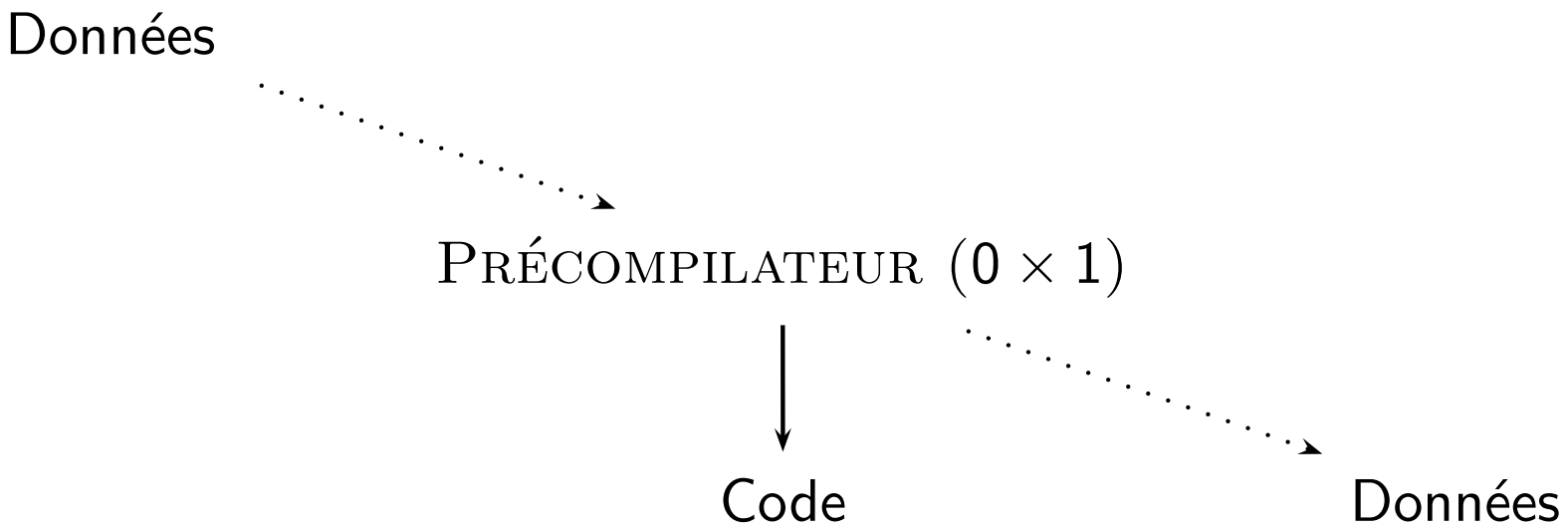
- Code C : programme en langage C
- Code machine : programme en langage machine
- Pseudocode : suite d'instructions à effectuer

Niveaux de programmation

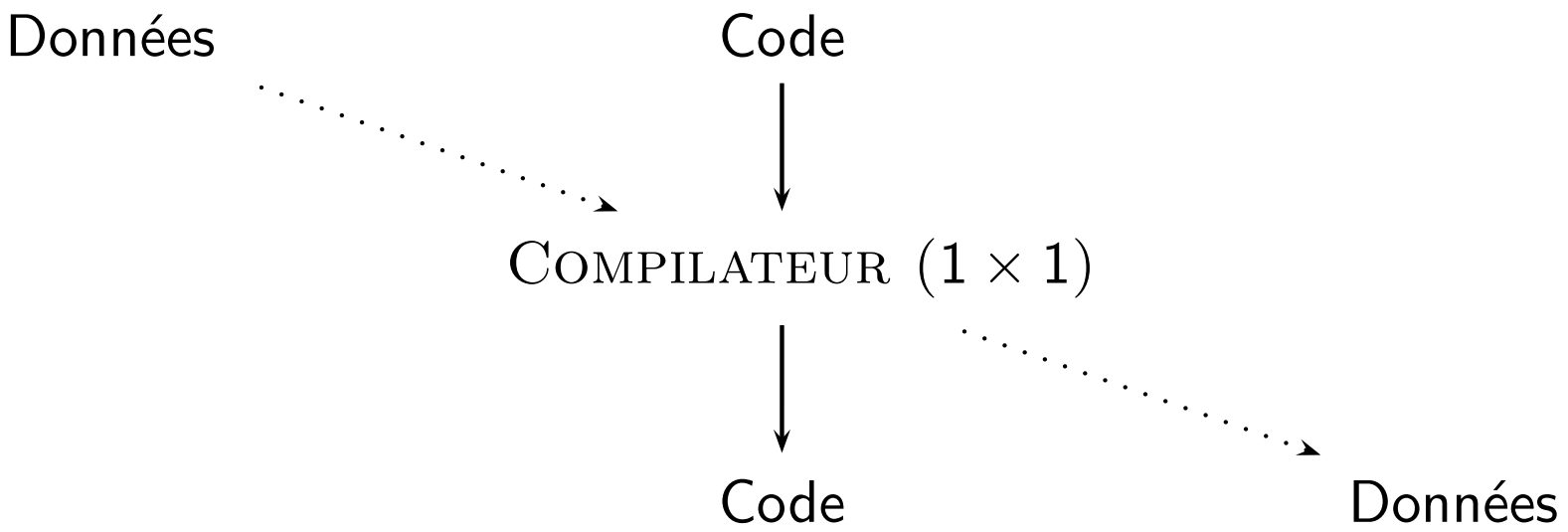
Niveaux de programmation



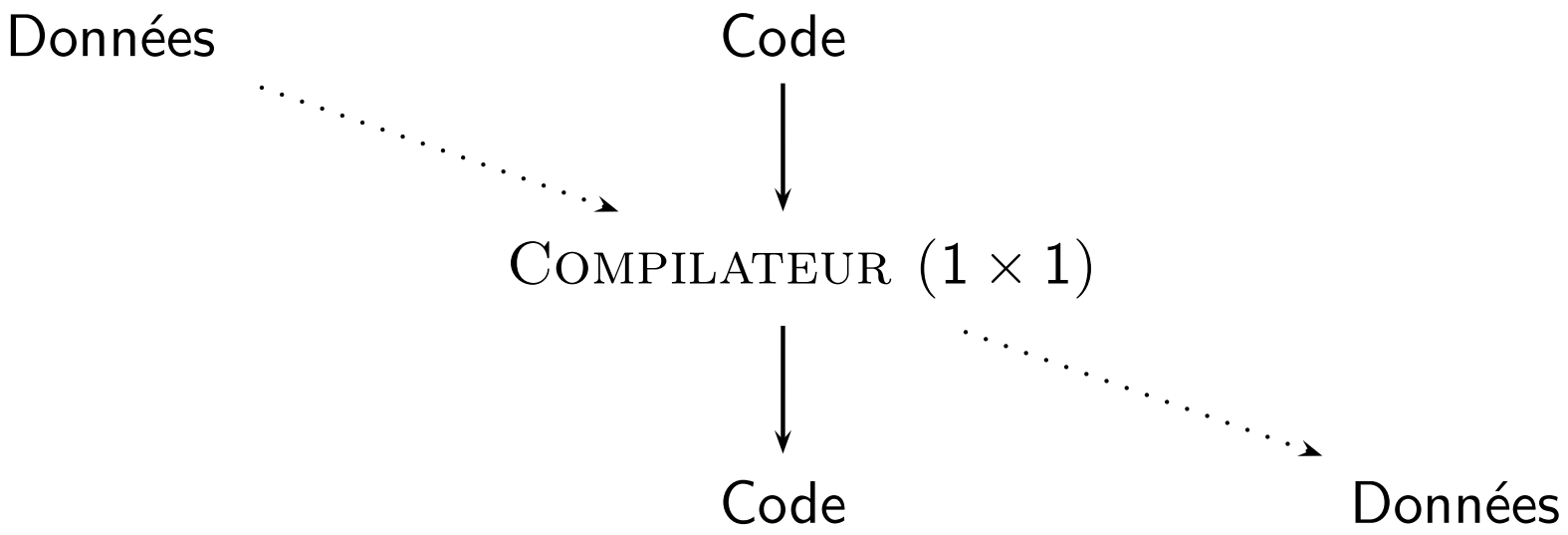
Niveaux de programmation



Niveaux de programmation



Niveaux de programmation



On pourrait continuer : vérificateur (2×0), métacompilateur (2×1), ...

Compilation

- Un compilateur prend du code et renvoie du code
- Un traducteur de C vers Ada est un compilateur
- La machine cible n'est pas obligatoirement celle sur laquelle tourne le compilateur (l'hôte)
- Certains systèmes sont à la fois des interpréteurs et des compilateurs

Compilation

- Un compilateur prend du code et renvoie du code
- Un traducteur de C vers Ada est un compilateur
- La machine cible n'est pas obligatoirement celle sur laquelle tourne le compilateur (l'hôte)
- Certains systèmes sont à la fois des interpréteurs et des compilateurs

Compilation

- Un compilateur prend du code et renvoie du code
- Un traducteur de C vers Ada est un compilateur
- La machine cible n'est pas obligatoirement celle sur laquelle tourne le compilateur (l'hôte)
- Certains systèmes sont à la fois des interpréteurs et des compilateurs

Compilation

- Un compilateur prend du code et renvoie du code
- Un traducteur de C vers Ada est un compilateur
- La machine cible n'est pas obligatoirement celle sur laquelle tourne le compilateur (l'hôte)
- Certains systèmes sont à la fois des interpréteurs et des compilateurs

Configurations typiques

- Configuration native : la cible T utilise la même architecture et le même système d'exploitation que l'hôte B
- Configuration croisée : la cible T n'utilise pas la même architecture ou le même système d'exploitation que l'hôte B (*cross compilation*)
- Configuration canadienne : l'hôte B génère un compilateur croisé du futur hôte H vers la cible T

Configurations typiques

- Configuration native : la cible T utilise la même architecture et le même système d'exploitation que l'hôte B
- Configuration croisée : la cible T n'utilise pas la même architecture ou le même système d'exploitation que l'hôte B (*cross compilation*)
- Configuration canadienne : l'hôte B génère un compilateur croisé du futur hôte H vers la cible T

Configurations typiques

- Configuration native : la cible T utilise la même architecture et le même système d'exploitation que l'hôte B
- Configuration croisée : la cible T n'utilise pas la même architecture ou le même système d'exploitation que l'hôte B (*cross compilation*)
- Configuration canadienne : l'hôte B génère un compilateur croisé du futur hôte H vers la cible T

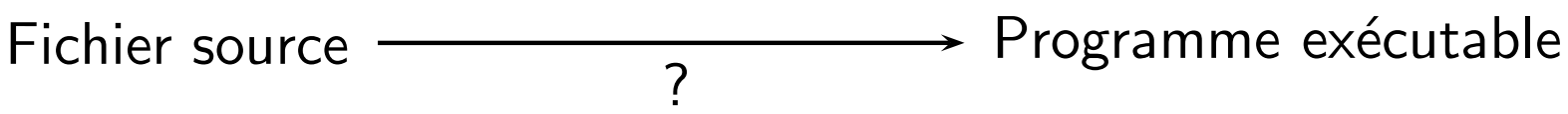
Configurations typiques

- Configuration native : la cible T utilise la même architecture et le même système d'exploitation que l'hôte B
- Configuration croisée : la cible T n'utilise pas la même architecture ou le même système d'exploitation que l'hôte B (*cross compilation*)
- Configuration canadienne : l'hôte B génère un compilateur croisé du futur hôte H vers la cible T

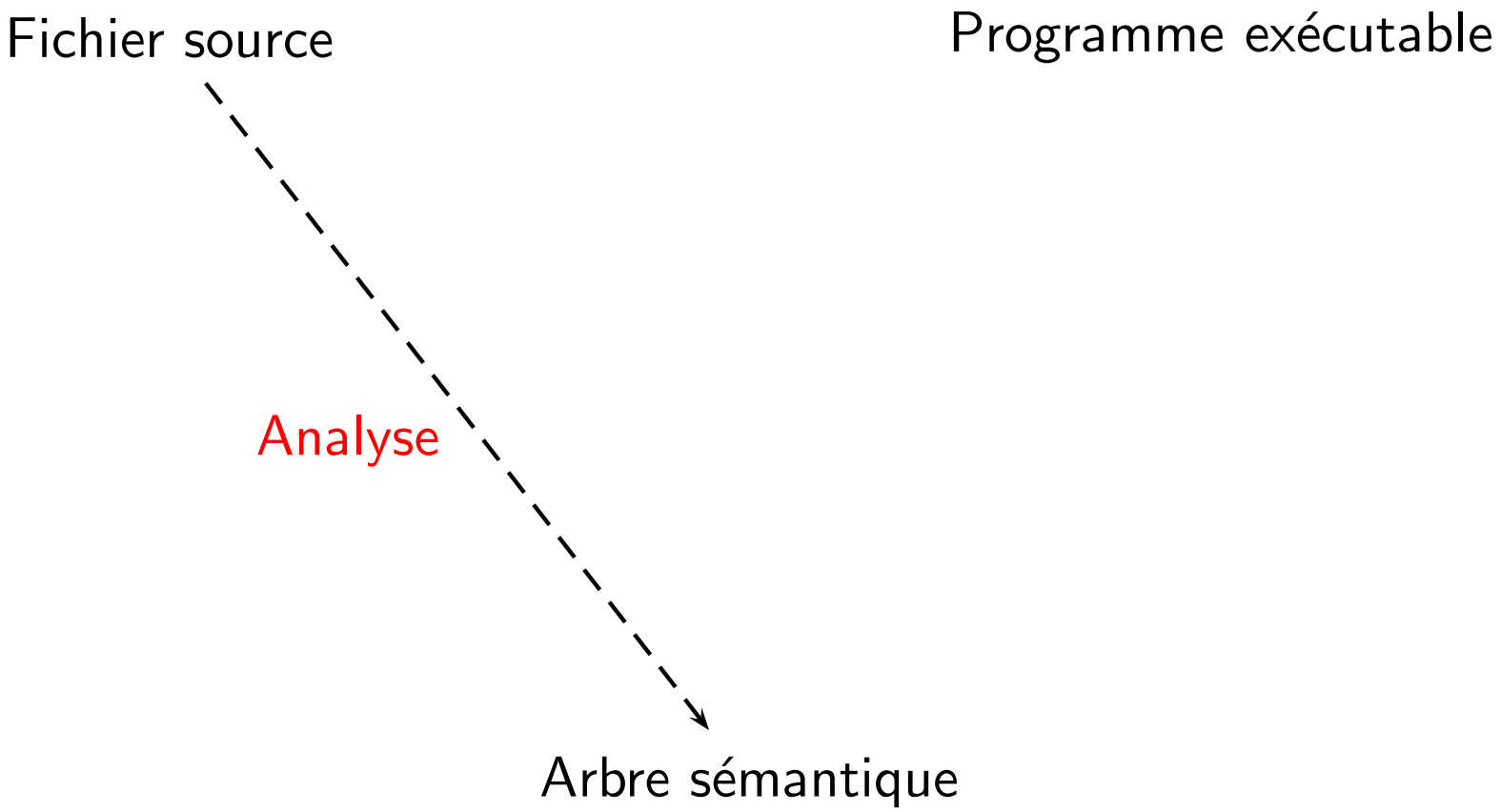
Exemple

Une société développant un compilateur Ada écrit en Ada n'utilise que des machines Intel sous GNU/Linux (B). Un client lui demande une version du compilateur pour leurs machines Sparc/Solaris (H) générant du code pour une carte ARM (T). Combien de compilateurs entrent en jeu ?

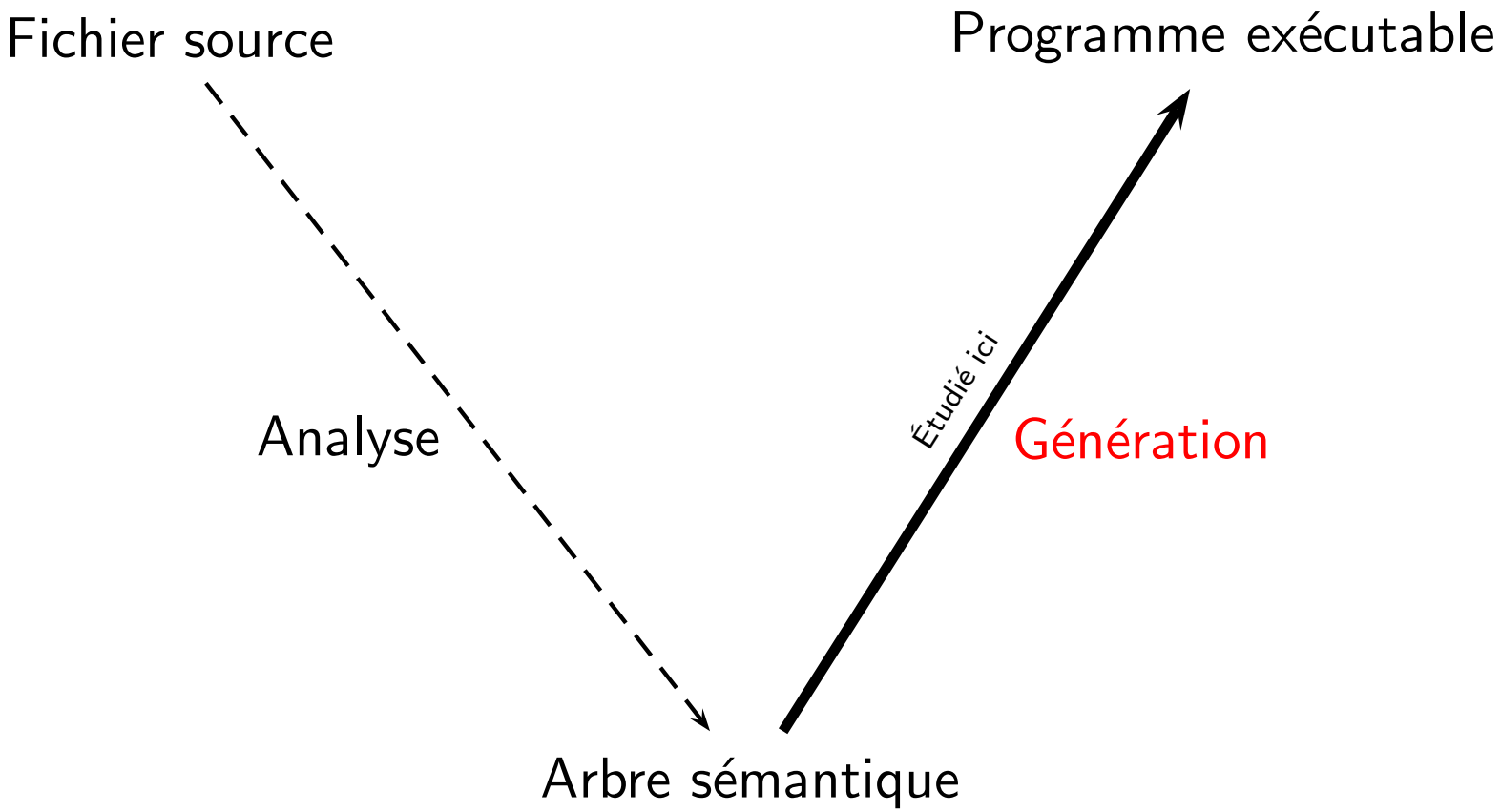
Processus de compilation



Processus de compilation



Processus de compilation



Rappels : frontal

La partie frontale d'un compilateur

- transforme le code en un arbre syntaxique
- construit les associations sémantiques
- vérifie la syntaxe et la sémantique du code

Elle peut également

- opérer des transformations sur l'arbre (optimisation, simplification)
- générer des informations de haut-niveau (nombre de lignes de code, présence de code mort, ...)

Rappels : frontal

La partie frontale d'un compilateur

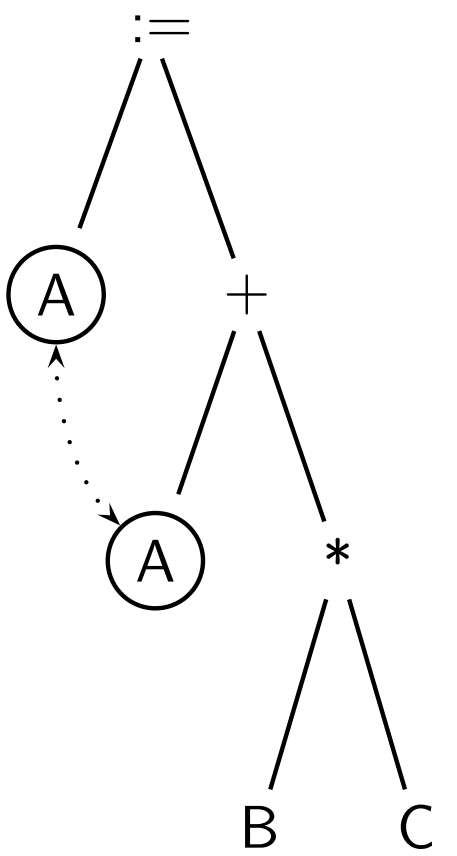
- transforme le code en un arbre syntaxique
- construit les associations sémantiques
- vérifie la syntaxe et la sémantique du code

Elle peut également

- opérer des transformations sur l'arbre (optimisation, simplification)
- générer des informations de haut-niveau (nombre de lignes de code, présence de code mort, ...)

Frontal : un exemple

```
A := A + B * C
```



Étapes

- Plusieurs phases de transformation peuvent être invoquées successivement
- La définition d'un langage intermédiaire facilite la réutilisation : M parties frontales et N générateurs de code permettent d'écrire $M + N + 1$ fragments plutôt que $M \times N$
- Le code machine peut être généré directement ou en langage d'assemblage (possibilité d'optimisations par l'assembleur, calcul de déplacement, déportation, etc.)

Un exemple : GCC

- GCC (GNU Compiler Collection) utilise un langage intermédiaire
- GCC génère un fichier assembleur temporaire
- L'optimisation se fait à chaque niveau
- Il est facile de rajouter un nouveau langage
- Il est facile de rajouter une nouvelle cible
- GCC peut être configuré en n'importe quelle configuration (native, croisée, canadienne)

Génération de code

But : permettre à chaque sous-programme présent dans l'arbre d'être appelé

Moyens :

- pour chaque instruction, générer du code effectuant les bonnes opérations
- générer du code pour l'entrée (prologue) et la sortie (épilogue) du sous-programme

Génération de code

But : permettre à chaque sous-programme présent dans l'arbre d'être appelé

Moyens :

- pour chaque instruction, générer du code effectuant les bonnes opérations
- générer du code pour l'entrée (prologue) et la sortie (épilogue) du sous-programme

Code machine

Le code machine

- est simple : x^y n'existe pas
- a peu d'arguments : $f(u, v, w, x, y, z)$ est impossible
- est peu structuré : `for (i=0; i<10; i++)` n'est pas représentable simplement

Il faut transformer l'arbre en instructions élémentaires.

Constructions ternaires

Classiquement, chaque opération est transformée en une suite de constructions au plus ternaires, car cela

- correspond en général aux possibilités d'un microprocesseur classique
- permet d'optimiser indépendamment chaque instruction
- permet d'unifier les sous-expressions communes
- correspond à une base $R \leftarrow M \otimes N$

Opérations ternaires

- Affectation : $x \leftarrow y \otimes z$
- Affectation : $x \leftarrow \otimes y$
- Copie : $x \leftarrow y$
- Saut : goto L
- Saut conditionnel : if $x \otimes y$ goto L
- Appel : param x_i et call p, n
- Affectations indexées : $x[i] \leftarrow y$ et $x \leftarrow y[i]$
- Références : $x \leftarrow \&y$, $x \leftarrow *y$ et $*x \leftarrow y$

Décomposition ternaire

`a = a + b * c;` peut devenir :

$$
\begin{aligned}
r_1 &\leftarrow (b) \\
r_2 &\leftarrow r_1 \times (c) \\
r_3 &\leftarrow (a) \\
r_4 &\leftarrow r_3 + r_2 \\
(a) &\leftarrow r_4
\end{aligned}
$$

En général, on ne peut avoir qu'une adresse (ou constante) manipulée à la fois, plus des registres.

Décomposition ternaire

`*x++ = 3;` peut devenir :

$$
\begin{aligned}
r_1 &\leftarrow (x) \\
(r_1) &\leftarrow 3 \\
r_2 &\leftarrow r_1 + 4 \\
(x) &\leftarrow r_2
\end{aligned}
$$

Architecture IA32

- Intel 32 bits (386, 486, Pentium, PII, PIII, P4)
- Architecture CISC
- Très peu de registres généraux (`%eax`, `%ebx`, `%ecx` et `%edx` principalement)
- Versions réduites de ces registres disponibles (`%ax` sur 16 bits et `%ah` et `%al` sur 8 bits)

Exemple sur IA32

`a = a + b * c;` devient :

```
        movl b,%eax
        imull c,%eax
        addl %eax,a
```

`*x++ = 3;` devient :

```
        movl x,%eax
        movl $3,(%eax)
        addl $4,x
```

GCC et assembleur

- Pour générer le fichier assembleur *fichier.S*, il faut d'utiliser :
  `gcc -S` *fichier.c*
- Rajouter `-O` permet de rendre le code plus lisible (le défaut est de ne mettre **aucune** optimisation)
- Rajouter `-o -` pour visualiser le résultat sur la sortie standard

Exemple

`gcc -S -o - -O` *fichier.c*

Restrictions

Sur un grand nombre de microprocesseurs, les restrictions suivantes existent :

- la plupart des opérations ne peuvent faire référence qu'à **un seul** non-registre de manière basique (pas d'indirection, etc.) et à des registres
- les registres sont en nombre **très limité** (CISC) ou **limité** (RISC)
- tous les registres ne sont pas utilisables pour toutes les opérations

Plan

Allocation des registres

- Le nombre de registres est restreint
- Les registres sont utilisés comme variables temporaires
- La durée de vie d'une variable temporaire est... temporaire
- Un registre inutilisé est réutilisable
- On cherche à minimiser la durée de vie de chaque registre

Durée de vie

- La vie commence à la **définition** d'un registre (première écriture ou registre existant lors de l'entrée de la fonction)
- Elle dure jusqu'à la dernière utilisation sans définition intermédiaire (dernière lecture avant une écriture, ou survie après la fonction)
- Un registre physique a potentiellement plusieurs vies
- Ces vies successives ne doivent pas (ne peuvent pas) interférer

Durée de vie : exemple 1

Le code C :

```
a = b + c;
d = e + f;
```

génère

```
movl c,%eax
addl b,%eax
movl %eax,a
movl f,%eax
addl e,%eax
movl %eax,d
```

Le registre %eax a été réutilisé.

Durée de vie : exemple 2

Le code C :

```
a = b + c;
d = e + f;
b = a;
```

génère

```
movl c,%eax
addl b,%eax
movl %eax,a
movl f,%edx
addl e,%edx
movl %edx,d
movl %eax,b
```

Le compilateur a choisi deux registres différents (%eax et %edx) pour conserver la valeur de a dans le premier.

Techniques d'allocation

Les étapes d'allocation de registre sont :

- découpage par instructions ternaires
- affectation de registres virtuels en nombre infini
- calcul des durées de vie
- affectation sur des registres physiques

Cette dernière phase requiert des calculs d'interférence et une coloration.

Interférence entre registres

Un calcul d'interférence est fait entre les registres virtuels t_i.

- On construit un graphe dont les sommets sont les t_i
- On ajoute des arêtes lorsque deux registres interfèrent :
  - Ils sont tous les deux *vivants* en même temps, ou
  - Un registre est incompatible avec un registre physique (qu'on ajoute alors à la liste des registres)

Coloration du graphe

On colore le graphe d'inférence avec au plus N couleurs, où N est le nombre de registres physiques disponibles.

- Deux sommets reliés par une arête doivent être de couleur différente (ils interfèrent)
- Un registre physique est affecté à chaque couleur

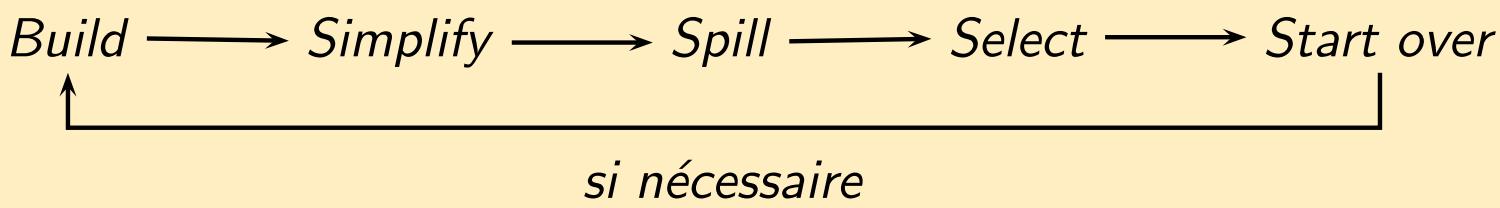
Approximation

Problème

Le coloriage d'un graphe en N couleurs est un problème NP-complet

Approche choisie

On connaît une approximation de la coloration dont le temps d'exécution est linéaire par rapport au nombre de sommets.



Exemple

Chaque lettre représente ici un registre virtuel.

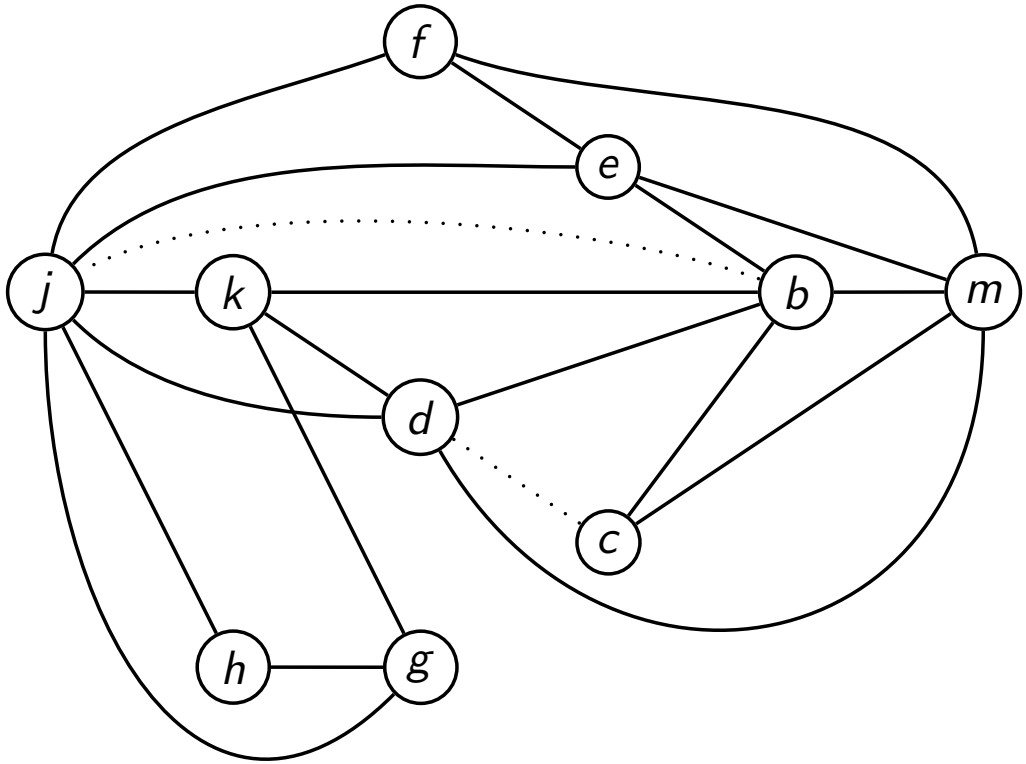
$$
\begin{aligned}
g &\leftarrow *(j+12) \\
h &\leftarrow k-1 \\
f &\leftarrow g \times h \\
e &\leftarrow *(j+8) \\
m &\leftarrow *(j+16) \\
b &\leftarrow *f \\
c &\leftarrow e+8 \\
d &\leftarrow c \\
k &\leftarrow m+4 \\
j &\leftarrow b
\end{aligned}
$$

Problématique

- k et j sont vivants à l'entrée et à la sortie du bloc
- d est vivant à la sortie du bloc
- On ne dispose que de 4 registres physiques
- Première étape : construction du graphe contenant
  - les interférences
  - les copies – initialement, une copie n'est pas considérée *per se* comme une interférence

Construction

Les arêtes en pointillés représentent des copies, les autres des interférences.

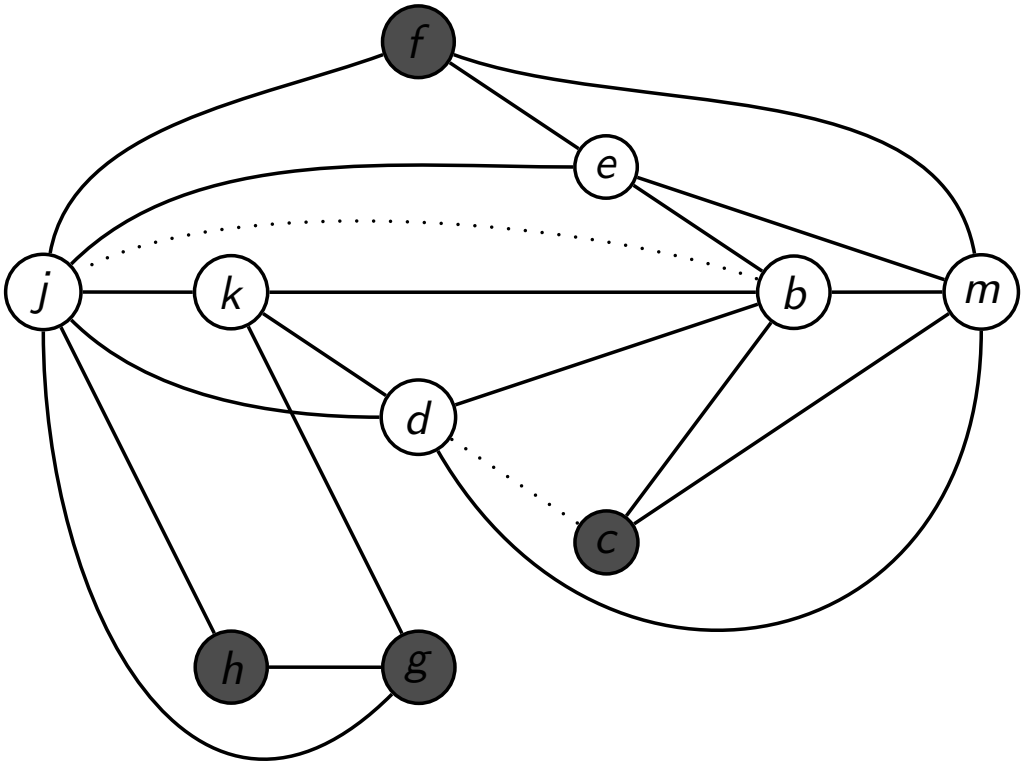


Simplification : principes

Soit K le nombre de registres physiques disponibles, et G le graphe d'interférences. Pour chaque sommet S relié à n_S arêtes, si $n_S < K$, alors s'il existe une solution au problème de coloriage pour le graphe $G \backslash S$, alors il existe une solution au problème de coloriage pour le graphe G.
On met donc de côté, sur une pile, les sommets qu'on peut enlever.

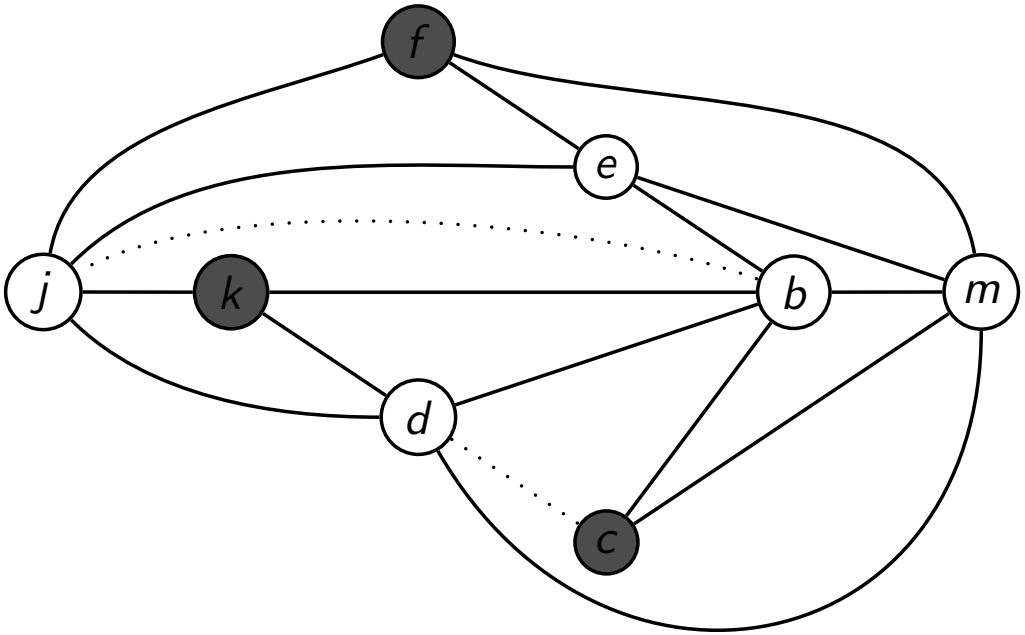
Recherche de simplification

Les sommets g, h, c et f sont candidats pour la simplification. On enlèvera ici h et g.



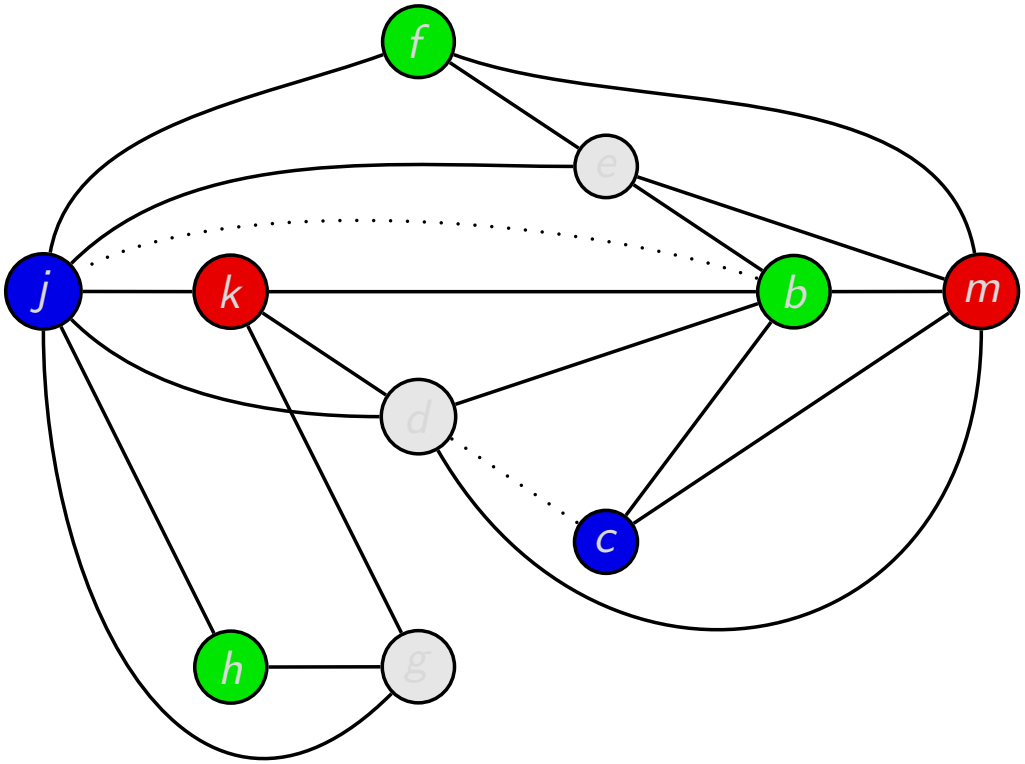
Résultat de la simplification

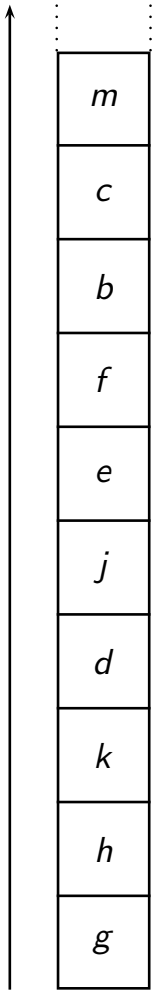
Le fait d'avoir supprimé des sommets permet d'en éliminer d'autres et de les ajouter sur la pile.



Après la simplification

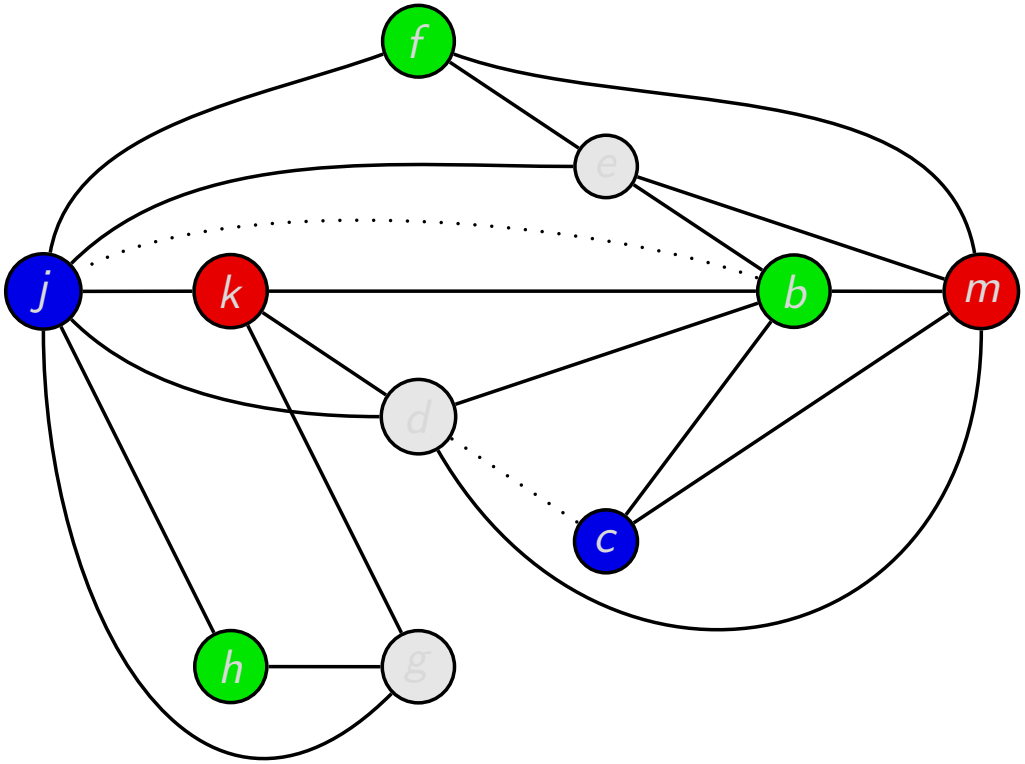
On a pu supprimer tous les sommets. L'état de la pile est indiqué ci-contre.

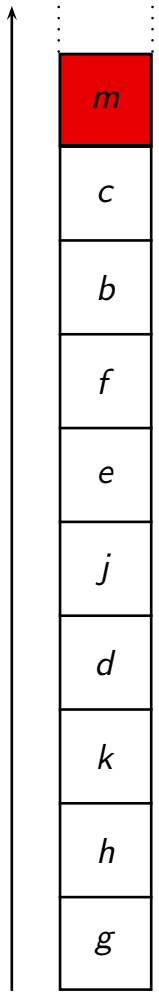




Après la simplification

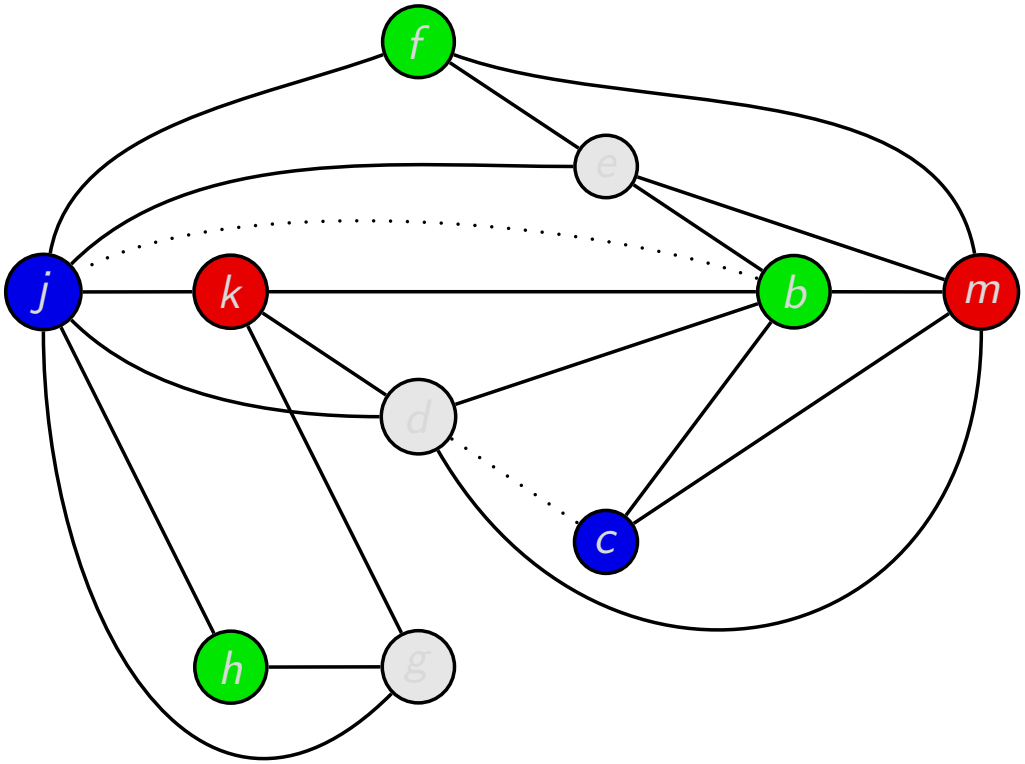
Sélection : on dépile les sommets un par un et on reconstruit le graphe, en choisissant une couleur que les voisins n'ont pas.

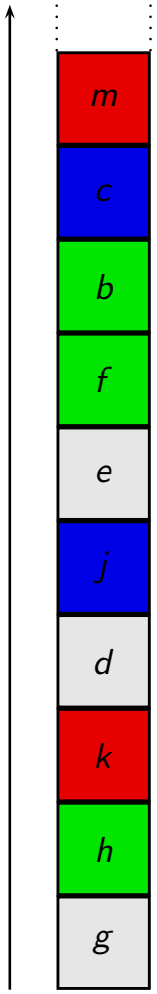




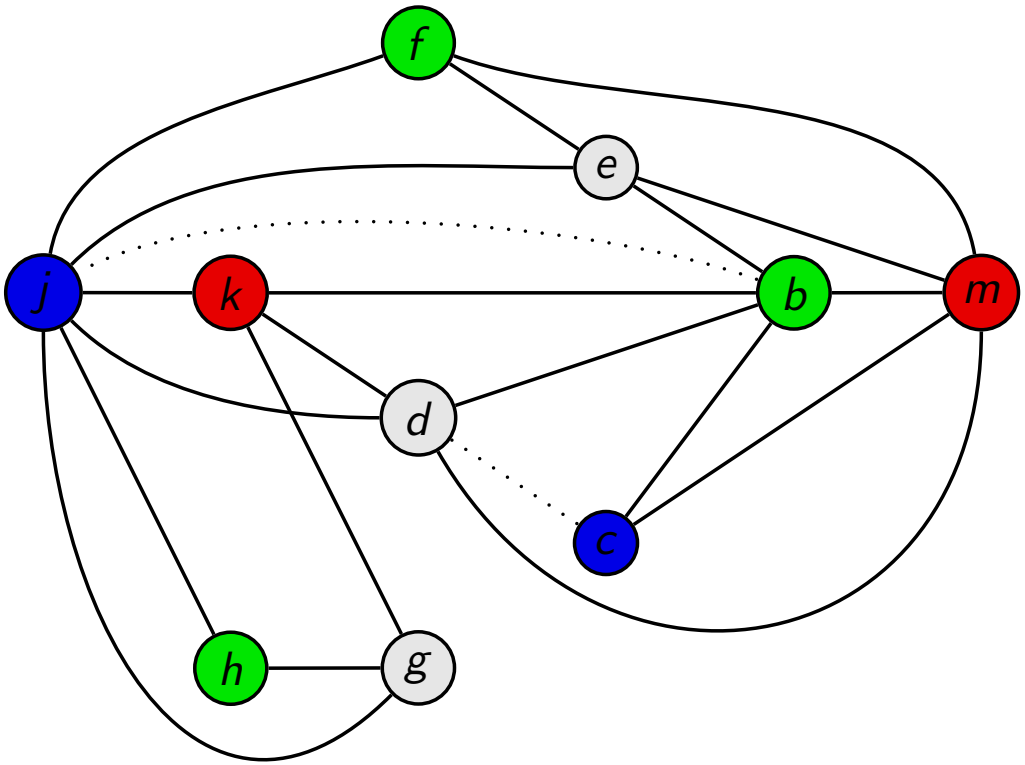
Après la simplification

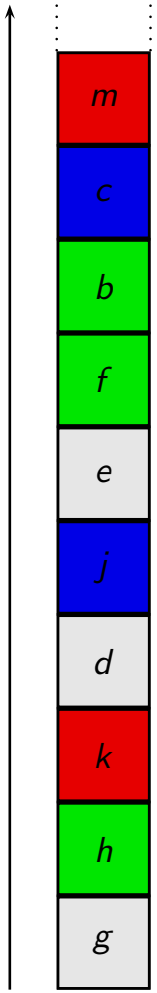
De proche en proche, on peut colorier l'ensemble des sommets.



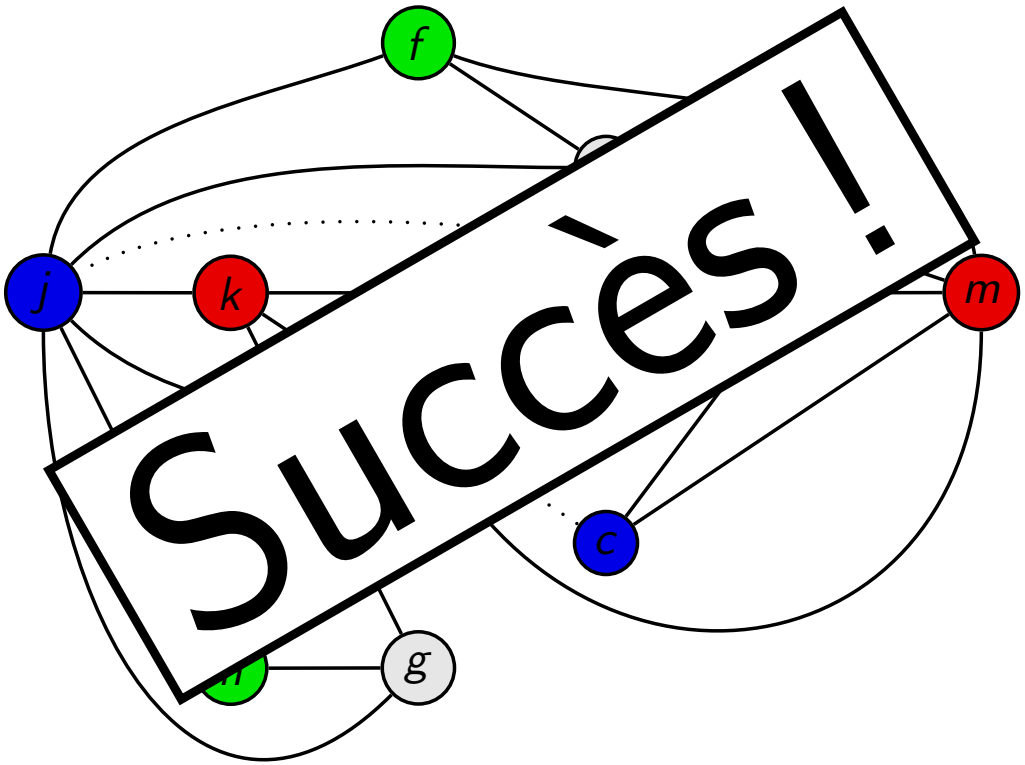


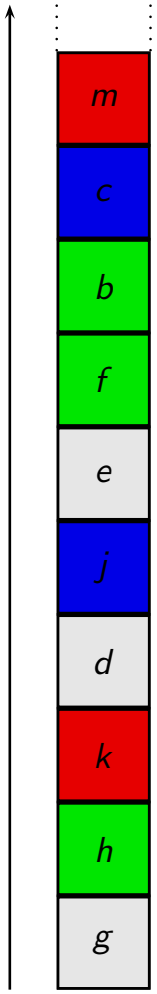
Après la simplification





Après la simplification





Spill

La phase de simplification ne fonctionne pas toujours, et certains sommets ne peuvent pas être supprimés.
Dans ce cas, on :

- enlève quand même le sommet
- le marque comme *spilled* dans la pile
- essaye de lui trouver une couleur lorsqu'on le dépile (c'est parfois possible, lorsque plusieurs voisins ont la même couleur)

Start over

Si la phase de sélection n'a pas pu trouver de registre pour un sommet, on :

- réécrit le programme pour lire/écrire en mémoire avant/après chaque usage du registre virtuel (plus de registres, durées de vies courtes)
- recommence l'algorithme au début

En pratique, une ou deux passes suffisent quasiment toujours, malgré les nouvelles interactions.

Amélioration

On peut fusionner les sommets si :

- l'un est une copie de l'autre
- et ils n'interfèrent pas

Dans ce cas :

- les arêtes du nouveau sommet sont les unions des arêtes des anciens sommets
- l'instruction de copie est supprimée du graphe et les références réécrites

Quand fusionner ?

But : ne pas transformer un graphe K-coloriable en un graphe non K-coloriable

Deux stratégies sûres (mais non parfaites) :

Briggs : a et b peuvent fusionner si le nouveau sommet a moins de K sommets de degré au moins égal à K

George : a et b peuvent fusionner si, pour chaque voisin t de a, t interfère déjà avec b ou bien t est de degré inférieur à K

Quand fusionner ?

But : ne pas transformer un graphe K-coloriable en un graphe non K-coloriable

Deux stratégies sûres (mais non parfaites) :

Briggs : a et b peuvent fusionner si le nouveau sommet a moins de K sommets de degré au moins égal à K

George : a et b peuvent fusionner si, pour chaque voisin t de a, t interfère déjà avec b ou bien t est de degré inférieur à K

Adaptation du coloriage

Avec la fusion, le coloriage est modifié :

- La simplification ne considère pour sortir un sommet que les sommets non impliqués dans une copie
- La fusion intervient après la simplification, le plus tard possible ; en cas de fusion, on continue la simplification
- Si on ne peut ni simplifier ni fusionner, on marque un sommet de faible degré impliqué dans une copie comme étant hors-copie et on continue

Adaptation du coloriage

Avec la fusion, le coloriage est modifié :

- La simplification ne considère pour sortir un sommet que les sommets non impliqués dans une copie
- La fusion intervient après la simplification, le plus tard possible ; en cas de fusion, on continue la simplification
- Si on ne peut ni simplifier ni fusionner, on marque un sommet de faible degré impliqué dans une copie comme étant hors-copie et on continue

Adaptation du coloriage

Avec la fusion, le coloriage est modifié :

- La simplification ne considère pour sortir un sommet que les sommets non impliqués dans une copie
- La fusion intervient après la simplification, le plus tard possible ; en cas de fusion, on continue la simplification
- Si on ne peut ni simplifier ni fusionner, on marque un sommet de faible degré impliqué dans une copie comme étant hors-copie et on continue

Sommets précoloriés

- Les registres physiques sont précoloriés
- Cela permet d'épargner certains registres prédéfinis (%pc ou pointeur de pile par exemple)
- Tous ces sommets interfèrent avec les autres
- Un sommet précolorié :
  - a un degré infini
  - n'est jamais simplifié
- La simplification s'arrête lorsque il ne reste que des sommets précoloriés

Sauvegarde de registres

- Certains registres doivent être sauvés à l'entrée d'une fonction et restaurés à la fin *(callee-save)*
- Si ces registres ont une durée de vie permanente, ils ne seront jamais utilisés
- On fait une affectation et une restauration dans une variable temporaire pour réduire leurs durées de vie

Supposons que r_7 soit un registre à sauvegarder.

Sauvegarde de registres (2)

$$\begin{array}{c} \mathrm{def}(r_7) \\ \vdots \\ \mathrm{use}(r_7) \end{array} \quad \text{devient} \quad \begin{array}{c} \mathrm{def}(r_7) \\ t_{231} \leftarrow r_7 \\ \vdots \\ r_7 \leftarrow t_{231} \\ \mathrm{use}(r_7) \end{array}$$

S'il n'y a pas de pénurie de registres, t_{231} sera fusionné avec r_7 et aucun code ne sera généré.

Choix des registres

Certains registres sont

- ***callee-save :*** à sauver par l'appelé
- ***caller-save :*** à sauver par l'appelant

On veut privilégier

- les *caller-save* pour les variables temporaires éphémères
- les *callee-save* pour les variables temporaires vivantes autour d'appels de sous-programmes

Règles et heuristiques

- Une instruction `call` redéfinit tous les *caller-save*
- On choisit pour le *spill* les sommets de fort degré et de peu d'utilisations

Par conséquence,

- Une variable temporaire éphémère sera allouée dans un registre *caller-save*
- Une variable de longue durée provoquera le *spill* de la copie d'un registre *callee-save*

Allocation : conclusions

- L'allocation de registres n'est pas un problème simple
- Les langages à pile (Forth) n'en ont pas besoin
- Les heuristiques ne sont pas parfaites
- Les algorithmes ne sont que des approximations

Plan

Gestion de la pile

La ou les piles permettent de gérer les appels de sous programme, le passage d'arguments, les variables locales et la récursivité.

Appel de sous-programme

L'appel de sous-programme utilise la pile :

- l'instruction d'appel de sous-programme (`call` sur IA32) place l'adresse suivant l'instruction courante sur la pile et branche sur le sous-programme
- l'instruction de retour (`ret` sur IA32) dépile l'adresse de retour et y saute

Passage de paramètres

- L'appel à un sous-programme nécessite
  - de lui passer des paramètres
  - de récupérer une valeur de retour
- Une convention d'appel doit être établie entre les modules

Traditionnellement, la pile et les registres servent à passer les paramètres et un registre à récupérer la valeur de retour.

Paramètres en C

En C,

- les paramètres sont empilés en l'ordre inverse
- l'adresse de retour est la dernière empilée (automatiquement)

Conséquences :

- on peut passer, sans conséquences, trop d'arguments à une fonction C
- si on en passe moins, des valeurs fantaisistes seront récupérées

C : Exemple

Considérons le code C suivant :

```
int f (int x, int y, int z)
{
   return x+y*z;
}

void g ()
{
   t = f (x, y, z);
}
```

(x, y, z et t sont déclarés autre part)

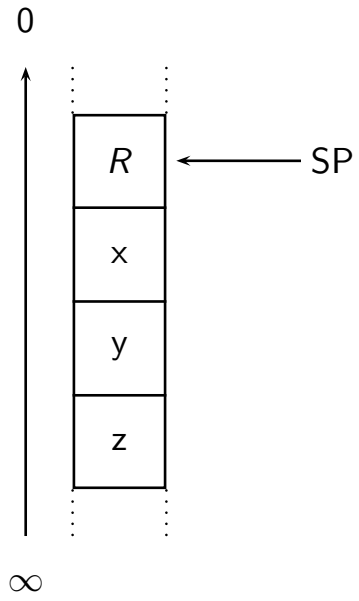
Architecture IA32

Sur architecture Intel,

- le registre de pile est `%esp`
- le retour d'une fonction est dans `%eax`
- un entier est codé sur 32 bits (4 octets)
- la pile va des adresses hautes vers les adresses basses

C : Arrivée dans f

À l'arrivée dans `f`, la pile contiendra :



Le $n^{\text{è}}$ argument est à l'adresse $\text{SP} + 4n$, indépendamment du nombre d'arguments.

Code IA32

Appel de f :

```
        pushl z
        pushl y
        pushl x
        call f
        movl %eax,t
        addl $12,%esp
```

f :

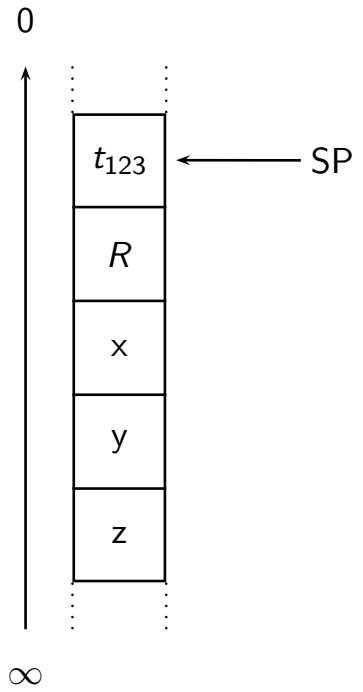
```
        movl 8(%esp),%eax
        imull 12(%esp),%eax
        addl 4(%esp),%eax
        ret
```

Pile et spilling

Lorsque du *spilling* intervient, on peut sauver et recharger la valeur d'un registre depuis :

- une adresse fixe en mémoire
- une adresse relative, sur la pile

La première méthode empêche les appels récursifs. On utilise donc la seconde.



Frame pointer

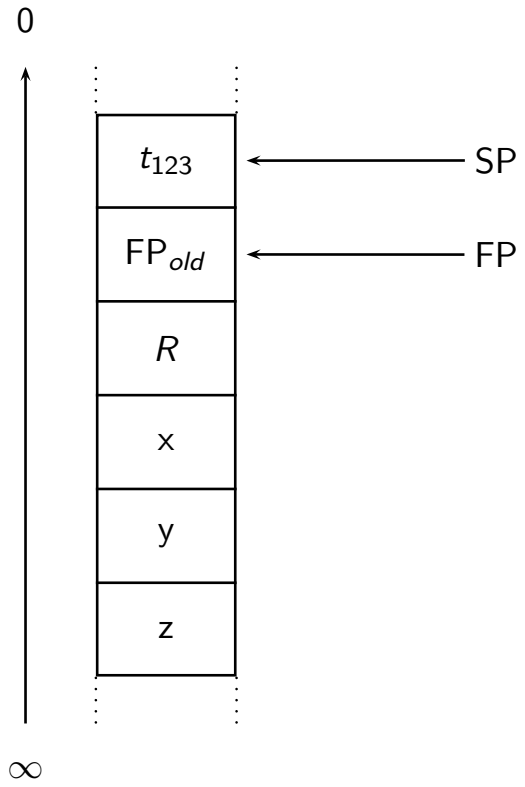
Le *spilling* sur la pile complique les choses :

- les adresses des arguments sont décalés
- les informations de *spilling* sont connues très tard dans le processus de compilation

On utilise un registre supplémentaire, le *frame pointer* (`%ebp` sur IA32) :

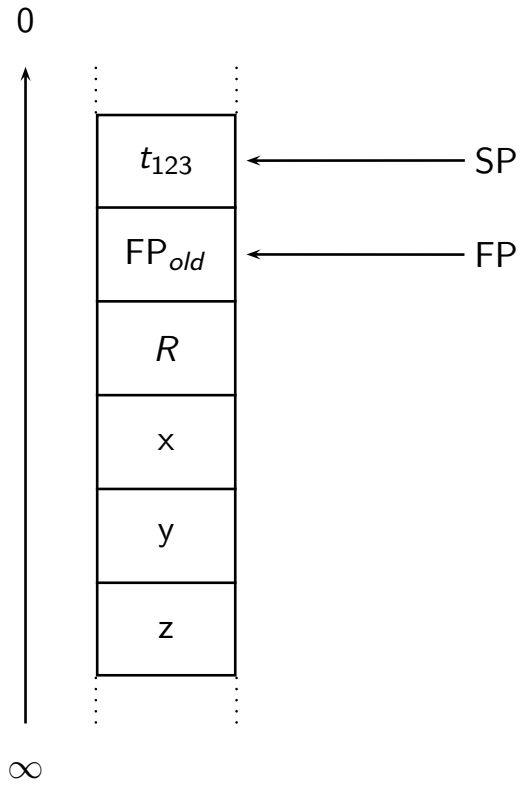
- il est sauvé sur la pile à l'entrée de la fonction
- il est restauré depuis la pile à la sortie de la fonction

Utilisation du FP



Le $n^{\text{è}}$ argument est maintenant en $FP + 4(n + 1)$

Utilisation du FP



Le $n^{\text{è}}$ argument est maintenant en $FP + 4(n + 1)$

Exemple et FP

Le code de l'exemple

```
int f (int x, int y, int z)
{
   return x+y*z;
}
```

devient :

```
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
imull 16(%ebp),%eax
addl 8(%ebp),%eax
leave
ret
```

IA32, FP et GCC

Sur architecture IA32, on dispose d'une instruction `leave` équivalente à :

```
movl %ebp, %esp
popl %ebp
```

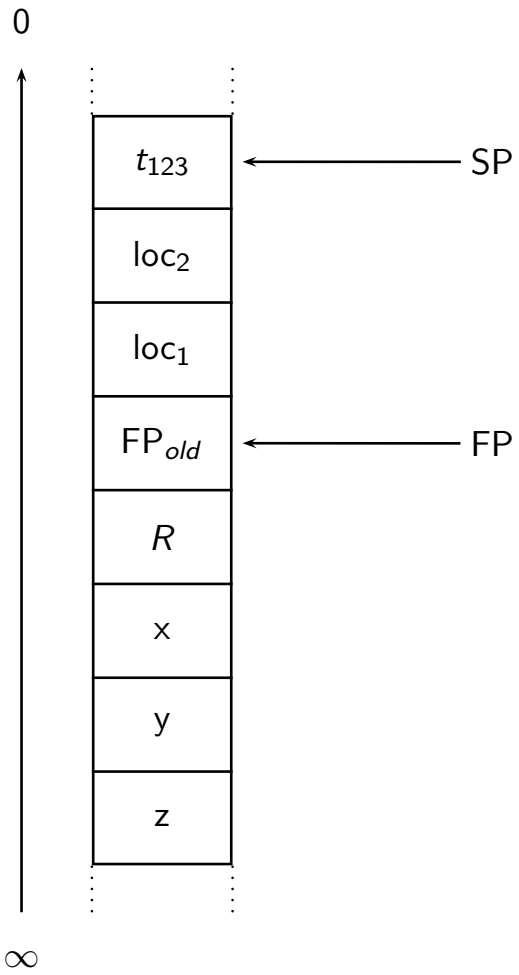
L'option `-fomit-frame-pointer` demande à GCC d'omettre, si possible, l'utilisation du *frame pointer*.

Variables locales

Les variables locales :

- sont stockées sur la pile
- voient leur place réservée lors de l'**activation** de la fonction (mise en place du *frame pointer* et décalage du pointeur de pile)
- sont référencées par le *frame pointer* : la $n^{\text{ème}}$ variable locale est en $\text{FP} - 4n$

Structure de la pile



Architectures modernes

Sur les architectures modernes :

- les premiers arguments sont passés dans des registres
- l'adresse de retour est passée dans un registre

Seuls les sous-programmes non terminaux (qui en appellent d'autres) auront besoin de les sauvegarder : on évite des opérations en mémoire.

Pile et interruptions

Pile et loi de Murphy

Il est nécessaire de ne jamais écrire au-delà du sommet de la pile, même si le pointeur de pile est modifié par la suite.

```
movl %eax,-4(%esp)
% Loi de Murphy: *BOUM* (int)
addl $-4,%esp
```

- La sauvegarde du contexte avant l'interruption utilise la pile
- Le problème est apparu dans Linux/gcc il y a quelques années

Problèmes non détaillés

- les sous-programmes imbriqués et les liens statiques
- les fermetures transitives
- l'utilisation du *frame pointer* par le débogueur

Inconvénients du modèle

Le mélange des adresses de retour et des arguments impose une activation et une sortie compliquées.

Plusieurs systèmes utilisent deux piles :

- Forth (pile de données et pile de retour)
- Stackless Python (pile C et pile Python)

Pile : conclusions

- La pile sacrifie au moins un registre (deux sur PowerPC)
- La pile nécessite des accès mémoire (accélérés grâce aux caches)
- La pile permet une récursion facile
- La pile et les frames permettent le déboguage
- La pile est trop utilisée en général

Plan

Contrôle de flux

Le contrôle de flux comprend :

- les sauts simples (`goto`)
- les appels de sous-programmes
- les tests de condition
- les boucles
- les sauts non locaux (interruptions, exceptions)

Cas simples (IA32)

Saut simple en C :

```
a:
   ...
   goto a;
```

devient :

```
.L3:
        ...
        jmp .L3
```

Appel de sous-programme en C :

```
f();
```

devient :

```
        call f
```

Tests de condition

La forme

$$\text{if } A \otimes B \text{ goto } L$$

se traduit en général par une séquence du type :

- évaluation d'une partie de la condition $A \otimes B$
- saut conditionnel à L à l'aide d'un opérateur spécialisé

Les conditions peuvent être inversées ou réarrangées.

Tests en assembleur

En assembleur, des instructions (spécifiques au tests ou généralistes) positionnent des bits dans un octet de status. Exemple :

- **Z** : zéro (égalité ou résultat nul)
- **C** : carry (débordement ou résultat négatif)

Des instructions permettent ensuite de sauter à un autre endroit en fonction de l'état d'un ou plusieurs bits.

Tests : exemple

Le test `if (a == b) f();` donne, sur IA32 :

```
        movl a,%eax
        cmpl b,%eax
        jne .L6
        call f
.L6:
```

La condition a été traduite en « si *a* et *b* **ne sont pas** égaux, **sauter par dessus** l'appel de *f* ».

Tests : if/then

D'une manière générale, `if (c) {thenpart}` devient :

évaluer c
si c est faux : sauter en *finlabel*
thenpart
finlabel :
...

Tests : if/then/else

D'une manière générale, `if (c) {thenpart} else {elsepart}` devient :

évaluer *c*
si *c* est faux : sauter en *elselabel*
thenpart
sauter en *finlabel*

elselabel :

elsepart

finlabel :

...

Switch

Une construction telle que

```
switch (a) {
  case 0:   f0 (); break;
  case 1:   f1 (); break;
  ...
  case 9:   f9 (); break;
}
```

pourrait être traduite par un ensemble de
`if (c == ...) ... else if ...`, mais serait inefficace.

Switch : amélioration

On utilise un tableau de pointeurs sur sous-programmes pour améliorer les résultats du switch :

```
typedef void (*noparam) (void);
static noparam dispatch [] =
  {f0, ..., f9};
if (a >= 0 && a <= 9)
  (dispatch[a])();
```

Résultat : exécution en temps quasiment constant

Boucles

Toutes les boucles sont transformées dans la structure suivante :

- 1- Initialisation des paramètres de la boucle
- 2- Saut par-dessus l'étape 3 si le test est en fin de boucle
- 3- Test de sortie, saut après 6 si positif
- 4- Corps de la boucle
- 5- Exécution de la partie finale de la boucle
- 6- Saut inconditionnel en 3

Exemple : boucle for

```
for (i=0;i<10;i++) {...}
```

- 1- $i \leftarrow 0$
- 2-
- 3- if $\neg(i < 10)$ goto 7
- 4- ...
- 5- $i \leftarrow i + 1$
- 6- goto 3

Exemple : boucle while

```
while (c) {...}
```

- 1-
- 2-
- 3- if $\neg(c)$ goto 7
- 4- ...
- 5-
- 6- goto 3

Exemple : boucle do/while

`do {...} while (c)`

- 1-
- 2- goto 4
- 3- if $\neg(c)$ goto 7
- 4- ...
- 5-
- 6- goto 3

Réécriture

Toutes ces constructions peuvent être réécrites à un plus haut niveau. Par exemple,

```
while (n < 10) {n = u (n);}
```

peut être réécrit comme

```
whilelabel:
  if (n < 10) {
    n = u (n);
    goto whilelabel;
  }
```

Réécriture et simplification

```
if (c) ok(); else notok();
```

peut se réécrire avec uniquement `if...goto` et `goto` :

```
  if (c) goto thenlabel;
  notok();
  goto endiflabel;
thenlabel:
  ok();
endiflabel:
```

Architectures différentes

- Sur les processeurs PIC, les instructions conditionnelles sont de la forme *sauter l'instruction suivante si un bit de statut est positionné ou non*
- Sur les processeurs PowerPC, toutes les instructions peuvent être conditionnées par la présence ou l'absence d'un bit de statut

Architectures différentes

- Sur les processeurs PIC, les instructions conditionnelles sont de la forme *sauter l'instruction suivante si un bit de statut est positionné ou non*
- Sur les processeurs PowerPC, toutes les instructions peuvent être conditionnées par la présence ou l'absence d'un bit de statut

Flux : conclusion

Le contrôle de flux

- peut s'exprimer de manière complexe dans un langage
- se traduit par des instructions simples en assembleur
- utilise des bits de statut
- peut prendre une forme différente sur certaines architectures, ce qui peut obliger à repenser la logique de la génération de code

Plan

Interface avec l'extérieur

Ce terme regroupe à la fois :

- l'échange de références entre les fichiers objet
- l'importation de symboles
- le traitement des paramètres initiaux (`argc`, `argv`, `environ`)
- l'appel au système d'exploitation
  - entrées/sorties
  - gestion des ressources

Interface avec l'extérieur

Ce terme regroupe à la fois :

- l'échange de références entre les fichiers objet
- l'importation de symboles
- le traitement des paramètres initiaux (`argc`, `argv`, `environ`)
- l'appel au système d'exploitation
  - entrées/sorties
  - gestion des ressources

Interface avec l'extérieur

Ce terme regroupe à la fois :

- l'échange de références entre les fichiers objet
- l'importation de symboles
- le traitement des paramètres initiaux (`argc`, `argv`, `environ`)
- l'appel au système d'exploitation
  - entrées/sorties
  - gestion des ressources

Interface avec l'extérieur

Ce terme regroupe à la fois :

- l'échange de références entre les fichiers objet
- l'importation de symboles
- le traitement des paramètres initiaux (`argc`, `argv`, `environ`)
- l'appel au système d'exploitation
  - entrées/sorties
  - gestion des ressources

Interface avec l'extérieur

Ce terme regroupe à la fois :

- l'échange de références entre les fichiers objet
- l'importation de symboles
- le traitement des paramètres initiaux (`argc`, `argv`, `environ`)
- l'appel au système d'exploitation
  - entrées/sorties
  - gestion des ressources

Interface avec l'extérieur

Ce terme regroupe à la fois :

- l'échange de références entre les fichiers objet
- l'importation de symboles
- le traitement des paramètres initiaux (`argc`, `argv`, `environ`)
- l'appel au système d'exploitation
  - entrées/sorties
  - gestion des ressources

Édition de liens

La compilation

- fait généralement référence à des entités externes
- ne connaît pas toutes ces entités

Le fichier assembleur

- exporte les symboles exportables
- fait référence à des symboles inconnus

L'éditeur de liens recolle les morceaux.

Assembleur et symboles

Le fichier

```
void main() { return f(); }
```

donne

```
.globl main
        .type    main,@function
main:
        subl $12,%esp
        call f
        addl $12,%esp
        ret
```

Stockage dans le fichier objet

L'utilitaire `nm` permet de visualiser les valeurs des symboles :

```
% nm t.o
         U f
00000000 t gcc2_compiled.
00000000 T main
```

Le symbole `f` est indéfini, et le symbole `main` est exporté depuis l'adresse relative 0 de la section *text*.

Lancement d'un exécutable

Le point d'entrée d'un exécutable est `_start`. Lors de l'édition de liens, le code de `_start` est ajouté à l'exécutable. Ce code :

- initialise les données
- charge les bibliothèques dynamiques
- positionne sur la pile les valeurs pour `environ`, `argv`, et `argc`
- appelle le sous-programme nommé `main`
- prend le retour de `main` comme code de sortie

Lancement d'un exécutable

Le point d'entrée d'un exécutable est `_start`. Lors de l'édition de liens, le code de `_start` est ajouté à l'exécutable. Ce code :

- initialise les données
- charge les bibliothèques dynamiques
- positionne sur la pile les valeurs pour `environ`, `argv`, et `argc`
- appelle le sous-programme nommé `main`
- prend le retour de `main` comme code de sortie

Lancement d'un exécutable

Le point d'entrée d'un exécutable est `_start`. Lors de l'édition de liens, le code de `_start` est ajouté à l'exécutable. Ce code :

- initialise les données
- charge les bibliothèques dynamiques
- positionne sur la pile les valeurs pour `environ`, `argv`, et `argc`
- appelle le sous-programme nommé `main`
- prend le retour de `main` comme code de sortie

Lancement d'un exécutable

Le point d'entrée d'un exécutable est `_start`. Lors de l'édition de liens, le code de `_start` est ajouté à l'exécutable. Ce code :

- initialise les données
- charge les bibliothèques dynamiques
- positionne sur la pile les valeurs pour `environ`, `argv`, et `argc`
- appelle le sous-programme nommé `main`
- prend le retour de `main` comme code de sortie

Lancement d'un exécutable

Le point d'entrée d'un exécutable est `_start`. Lors de l'édition de liens, le code de `_start` est ajouté à l'exécutable. Ce code :

- initialise les données
- charge les bibliothèques dynamiques
- positionne sur la pile les valeurs pour `environ`, `argv`, et `argc`
- appelle le sous-programme nommé `main`
- prend le retour de `main` comme code de sortie

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Notes sur le lancement

- La présence de `main` n'a rien d'obligatoire
  - c'est une convention en C et dans d'autres langages
  - cela permet d'automatiser des tâches pénibles
- `_start` récupère les informations transmises par le système d'exploitation depuis une structure
  - pointée par un registre, ou
  - à un endroit fixe par rapport à un registre
- `_start` se trouve au début de la zone *text* de l'exécutable (dans le cas du format ELF)

Invocation du système

- L'exécutable doit invoquer des services du système :
  - entrées/sorties
  - gestion des ressources
- Faire appel à des bibliothèques ne résoud rien : ces bibliothèques contiennent du code et des données, tout comme l'exécutable.
- Solution : utiliser un appel système (passage en mode noyau)

Appel système

- Il y a plusieurs moyens de commuter un processeur moderne dans un mode privilégié :
  - interruption physique
  - défaillance logicielle (instruction inconnue, instruction privilégiée, accès à une zone mémoire inexistante ou protégée)
  - trappe (déclenchement volontaire)
- Sous Unix, on utilisera typiquement une trappe (appel système)

Hello, World ! (FreeBSD)

Deux manières d'écrire *Hello, World !* pour FreeBSD.
Partie commune :

```
1:          .data                   # Data section
2:
3: msg:     .asciz "Hello, World!\n"
4:          len = . - msg - 1       # The length of the string.
5:
6:          .text                   # Code section.
```

Hello, World ! (libc)

```
 7:          .extern write
 8:          .extern exit
 9:          .global main
10:
11: main:
12:          pushl   $len
13:          pushl   $msg
14:          pushl   $1
15:          call    write
16:          addl    $12, %esp
17:
18:          pushl   $0
19:          call    exit
```

Hello, World ! (trappe)

```
 7:          .global _start
 8:
 9: _start:                          # Entry point.
10:         pushl   $len          # Arg 3 to write: length of string.
11:         pushl   $msg          # Arg 2: pointer to string.
12:         pushl   $1            # Arg 1: file descriptor.
13:         movl    $4, %eax      # Write.
14:         call    do_syscall
15:         addl    $12, %esp     # Clean stack.
16:
17:         pushl   $0            # Exit status.
18:         movl    $1, %eax      # Exit.
19:         call    do_syscall
20:
21: do_syscall:
22:         int     $0x80         # Call kernel.
23:         ret
```

Comparaisons inutiles

- La version avec bibliothèque C (statique) fait 12456 octets.
- La version avec bibliothèque C (dynamique) fait 2748 octets.
- La version sans bibliothèque C fait 464 octets.
- La version C (statique) fait 12528 octets.
- La version C (dynamique) fait 2840 octets.

Mandataires

Dans la bibliothèque C se trouve des fonctions mandataires pour les appels système, qui :

- positionnent dans `%eax` le numéro de l'appel système
- génèrent une trappe
- convertissent le retour (dans `%eax`) en `errno` si une erreur à eu lieu (en honorant la présence éventuelle de *threads*)

Quelle méthode choisir ?

- La génération directe des trappes
  - peut faire gagner de la place
  - peut accélérer l'exécution
- Le passage par une bibliothèque
  - peut faire gagner de la place (grace à la mutualisation du traitement des erreurs)
  - permet de réutiliser le même compilateur sur un autre OS (même architecture) si les ABI sont compatibles

Allocation mémoire

Qui alloue de la mémoire ?

- Le développeur d'application
- Le compilateur
  - pour des zones temporaires de taille importante
  - par le biais de l'expansion des constructions complexes

Qui gère la mémoire ?

- Le système d'exploitation
  - gère les tables de page et l'allocation de mémoire virtuelle
  - gère le chargement et le déchargement des pages
- Le programme, ou une bibliothèque
  - gère la mémoire allouée par le système
  - découpe la mémoire en zones de tailles fixes ou variables

Mémoire allouée

Sous Unix, deux appels système sont utilisés :

- `brk()` permet de spécifier la dernière adresse utilisable dans l'espace mémoire virtuel (contigü)
- `sbrk()` permet de déplacer la dernière adresse utilisée

Les fonctions C telles que `malloc()` (versions plus ou moins évoluées) utilisent `sbrk()`.

Interfaces : conclusion

- L'édition de liens permet l'interface avec les autres fichiers objets
- Les appels système permettent de demander au noyau d'agir
- La mémoire est allouée sous la forme d'un bloc extensible et est gérée en mode utilisateur (à l'exception des droits d'accès)

Plan

Langage orienté objet

La compilation d'un langage orienté objet pose de nouveaux problèmes (et apporte de nouvelles solutions) :

- aiguillage dynamique
- copies d'objets polymorphes
- déallocation d'objets polymorphes
- transtypage

Aiguillage dynamique

Problème

Sur quelle méthode diriger un appel de type `objet.methode()` ?

Solution

- Chaque classe possède une table de méthodes
- Chaque classe a un début de table identique à la classe parente
- Les surcharges occasionnent un remplacement dans la table adéquate
- Le même principe, sans surcharge, s'applique aux champs de l'objet

Aiguillage dynamique

Problème

Sur quelle méthode diriger un appel de type `objet.methode()` ?

Solution

- Chaque classe possède une table de méthodes
- Chaque classe a un début de table identique à la classe parente
- Les surcharges occasionnent un remplacement dans la table adéquate
- Le même principe, sans surcharge, s'applique aux champs de l'objet

Aiguillage et héritage

Problème

La méthode précédente n'est pas applicable dans le cas d'héritage multiple

Solution

- Chaque classe posède une table de méthodes
- Cette table est construite à partir de celle des classes parentes
- Un changement de vue provoque un décalage d'une entrée dans cette table
- Le même principe, sans surcharge, s'applique aux champs de l'objet

Aiguillage et héritage

Problème

La méthode précédente n'est pas applicable dans le cas d'héritage multiple

Solution

- Chaque classe posède une table de méthodes
- Cette table est construite à partir de celle des classes parentes
- Un changement de vue provoque un décalage d'une entrée dans cette table
- Le même principe, sans surcharge, s'applique aux champs de l'objet

Copie d'objets

Problème

Comment copier un objet polymorphe ? On ne connaît ni sa taille ni les opérations à effectuer sur d'éventuels objets inclus dans les champs du premier.

Solution

- Une méthode intrinsèque est ajoutée à chaque classe
- Cette méthode est appelée lorsqu'il faut réaliser une opération de copie
- En général, une telle méthode est surchargeable ou complétable (`Adjust` en Ada)

Copie d'objets

Problème

Comment copier un objet polymorphe ? On ne connaît ni sa taille ni les opérations à effectuer sur d'éventuels objets inclus dans les champs du premier.

Solution

- Une méthode intrinsèque est ajoutée à chaque classe
- Cette méthode est appelée lorsqu'il faut réaliser une opération de copie
- En général, une telle méthode est surchargeable ou complétable (`Adjust` en Ada)

Déallocation d'objets

Problème

Comment désallouer un objet polymorphe ? On ne connait pas sa taille.

Solution

- Une méthode intrinsèque est ajoutée à chaque classe
- Cette méthode est appelée lorsqu'il faut désallouer l'objet
- Pour certains langages, il suffit d'un champ (ou d'une méthode) renvoyant la taille d'un objet de la classe

Déallocation d'objets

Problème

Comment désallouer un objet polymorphe ? On ne connait pas sa taille.

Solution

- Une méthode intrinsèque est ajoutée à chaque classe
- Cette méthode est appelée lorsqu'il faut désallouer l'objet
- Pour certains langages, il suffit d'un champ (ou d'une méthode) renvoyant la taille d'un objet de la classe

Transtypage

Problème

Comment spécialiser le type d'un objet polymorphe ?

Solution

- Chaque objet possède un champ (ou une méthode) pointant vers un tableau d'ancêtres pour sa classe, et un pointant vers son niveau d'héritage (profondeur)
- Héritage simple : vérifier l'entrée correspondant à la profondeur de la cible
- Héritage dynamique : parcourir la table

Transtypage

Problème

Comment spécialiser le type d'un objet polymorphe ?

Solution

- Chaque objet possède un champ (ou une méthode) pointant vers un tableau d'ancêtres pour sa classe, et un pointant vers son niveau d'héritage (profondeur)
- Héritage simple : vérifier l'entrée correspondant à la profondeur de la cible
- Héritage dynamique : parcourir la table

Objets : conclusion

- Un langage orienté objet nécessite un traitement spécifique
- Des méthodes et/ou champs intrinsèques (éventuellement cachés) sont produits
- L'héritage simple permet de générer du code efficace
- L'héritage multiple complique et ralentit le processus

Plan

Extras : plan

- Compilation croisée
- Compilation ombilicale
- Programmes multi-tâches

Compilation croisée

- La cible est différente de l'hôte
- Sert :
  - lorsque la cible n'a pas les capacités nécessaires pour faire tourner un compilateur
  - pour porter un compilateur vers une machine n'en disposant d'aucun (GCC)
  - pour porter un compilateur se compilant avec lui-même (GNAT)

Compilation ombilicale

- L'hôte compile pour la cible
- L'hôte envoie le programme compilé à la cible au fur et à mesure
- La cible peut exécuter des commandes interactivement
- Exemple : Forth pour carte à puce

Programmes multi-tâches

- Les fonctions de bibliothèques sont redéfinies
- Les appels système
  - sont réentrants dans le cas de threads noyaux
  - doivent être surchargés dans le cas de threads utilisateurs
- Si le langage supporte le mode multi-tâches nativement, une phase d'expansion a généralement lieu

Question non abordées

- Code auto-modifiant
  - Code généré au vol par le programme lui-même
  - Utilisé dans GCC pour les trampolines
- Représentation des nombres en virgule flottante

Synthèse

On peut maintenant construire un premier compilateur :

- allocation des registres
- gestion de la pile
- gestion des flux
- interface avec le monde extérieur
- programmation orientée objet
- quelques extras