



# Structure interne d'un compilateur et représentation intermédiaire

---

Patrick Carribault

*patrick.carribault@cea.fr*



# Plan du cours

---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation



# Structure générale d'un compilateur

---



# Plan du cours

---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation

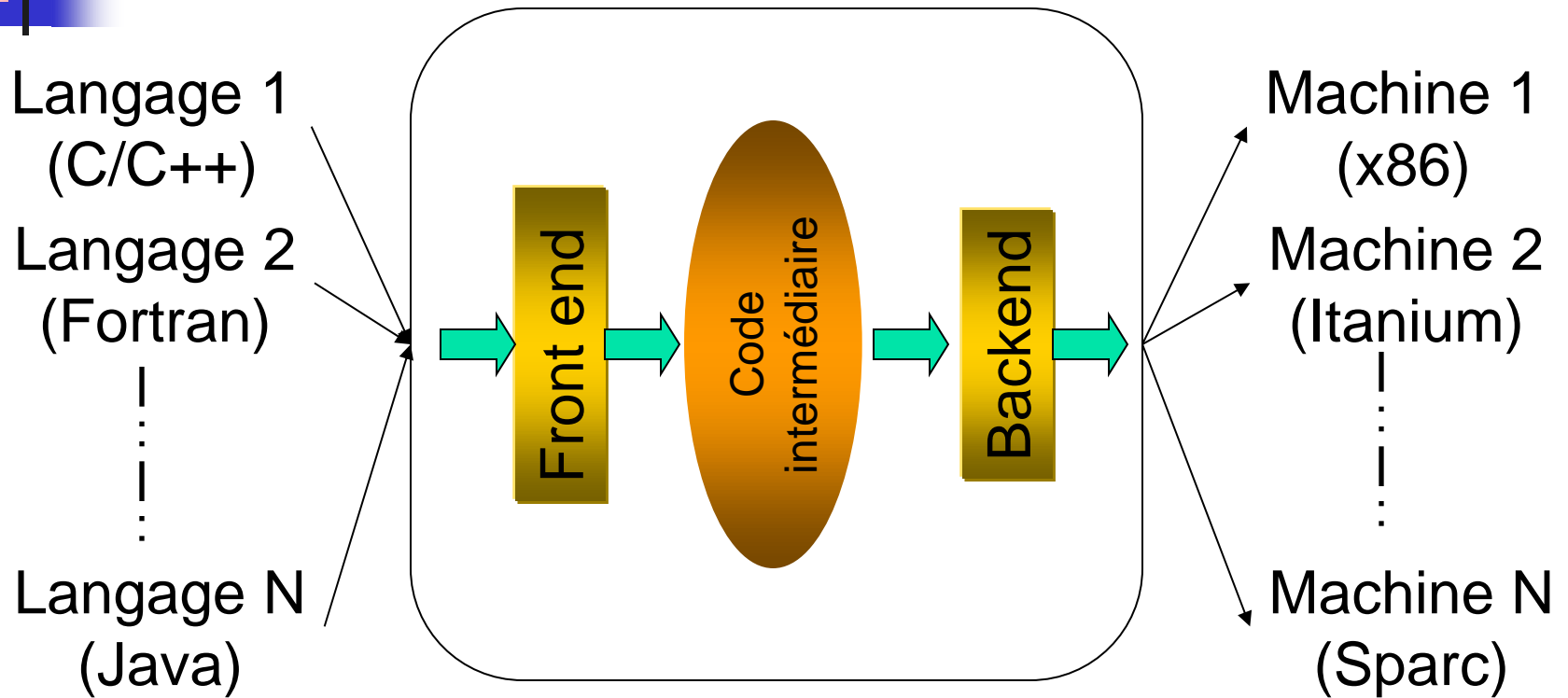


# Compilateur standard

---

- Définition :
  - Traducteur de langages
  - Extensions : analyseur / optimiseur
- Vision boîte noire :
  - Entrée : un langage de programmation
  - Sortie : un langage de programmation
- Principales parties
  - Préprocesseur
  - Cœur du compilateur
  - Assembleur
  - Linker
  - Loader (exécution)
- Utilisation de bibliothèques/outils annexes

# Compilateur standard



Vision idéaliste



# Plan du cours

---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation



# Représentation intermédiaire

---

- *Code intermédiaire* ou langage intermédiaire (IR ou IL en anglais)
- Définition : réécriture d'un programme P1 d'un langage L1 vers un autre programme P2 d'un langage L2 tel que
- Contraintes
  - Conservation de la sémantique : P1 calcule la même chose que P2
  - Baisse du niveau d'abstraction : L2 est plus "près" de la machine cible (L2 est un langage "simplifié" par rapport à L1)





# Sémantique

---

- Définition : le sens du programme, son but, son algorithme
- Changer la sémantique : c'est modifier le but ultime du programmeur
  - A tort ou à raison...
- Exemple : un code C faux

```
int *ptr = NULL ;  
*ptr = 5 ;
```
- Vis-à-vis du compilateur
  - un compilateur peut déterminer que ceci va planter
  - mais il se doit de faire ce que demande le programmeur



# Sémantique

---

- Ce que fait le compilateur
  - Emet des avertissements lorsque le code est ambigu ou dangereux
- Exemple : un code C dangereux

```
if (x = 5)
    printf ("Hello world !") ;
```
- Vis-à-vis du compilateur
  - Rien ne dit que ceci n'est pas exactement voulu
  - Le compilateur va tout de même générer :

```
main.c :3 : warning : suggest
parentheses
```



# Représentation intermédiaire et architecture

---

- Cas général des architectures
  - Programmes impératifs
  - Ressemblance avec le modèle von-Neumann
- Conséquences
  - Représentations intermédiaires dans ce genre de machines doivent être de langage impératif.
- Si la machine est *data flow* par ex, on aurait choisi une représentation intermédiaire data flow ou fonctionnelle.

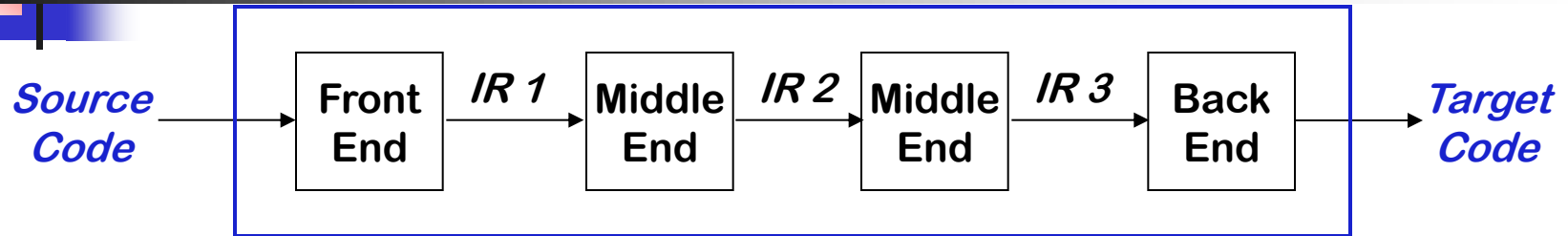


# Multiples représentations

---

- Complexité accrue des compilateurs
  - ➔ Plusieurs niveaux de représentations intermédiaires peuvent cohabiter
- But : procéder par étapes successives
  - Optimisation progressives
  - Plusieurs phases d'optimisations avant d'arriver au code binaire de la machine cible
- Compilateur  $\Leftrightarrow$  succession de compilateurs en cascade.

# Multiples représentations



- Abaisser à plusieurs reprises le niveau d'abstraction de la forme intermédiaire
  - Chaque représentation intermédiaire est appropriée pour certaines optimisations
- Ex: compilateur Open64
  - Forme intermédiaire appelé WHIRL
    - Consiste en cinq représentations intermédiaires progressivement détaillées



# Comment choisir ?

---

- Conception d'une forme intermédiaire
  - Affecte l'efficacité et la rapidité d'un compilateur
  - Affecte la qualité du programme généré
- Quelques critères de sélection
  - Facilité de génération
  - Facilité de manipulation
  - Taille de code induite
  - Liberté et puissance d'expression d'informations
  - Niveau d'abstraction
- L'importance de ces critères diffère selon les compilateurs
  - Sélectionner une forme intermédiaire pour un compilateur est une décision de conception importante !



# Type de représentation intermédiaire (RI)

---

- On peut les classer en trois catégories majeures

- RI Linéaire (code textuel)

- Pseudo-code pour une machine abstraite
- Le niveau d'abstraction varie
- Simple et de taille plus compacte
- Facile à réécrire et manipuler

Exemples :

Code 3 adresses,  
Code machine à pile

- RI structurée

- Utilise les graphes
- Beaucoup utilisée dans les traducteurs source à source
- Facilite une vision abstraite et globale d'un programme
- Nécessite une présence en mémoire qui peut être large

Exemples :

Arbres, DAGs

- Hybride

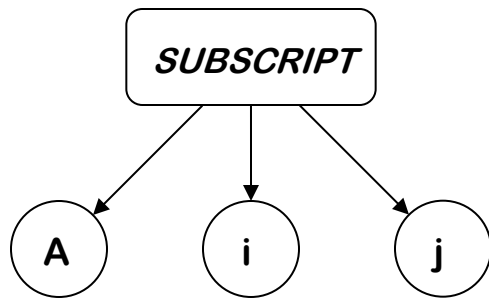
- Combinaison entre graphes et codes linéaires
- Cas le plus fréquent de nos jours

Exemple :

Graphe de flot de  
contrôle

# Niveau d'abstraction

- Le niveau de détails exposé dans une RI influence la profitabilité et la faisabilité de plusieurs optimisations.
- Ex : deux représentations possibles d'une référence à un élément de tableau  $A[i,j]$ .



Arbre syntaxique haut  
niveau :  
favorable pour une  
désambiguation mémoire

```
loadI 1      => r1
sub    rj, r1 => r2
loadI 10     => r3
mult   r2, r3 => r4
sub    ri, r1 => r5
add    r4, r5 => r6
loadI @A     => r7
Add    r7, r6 => r8
load    r8    => rAij
```

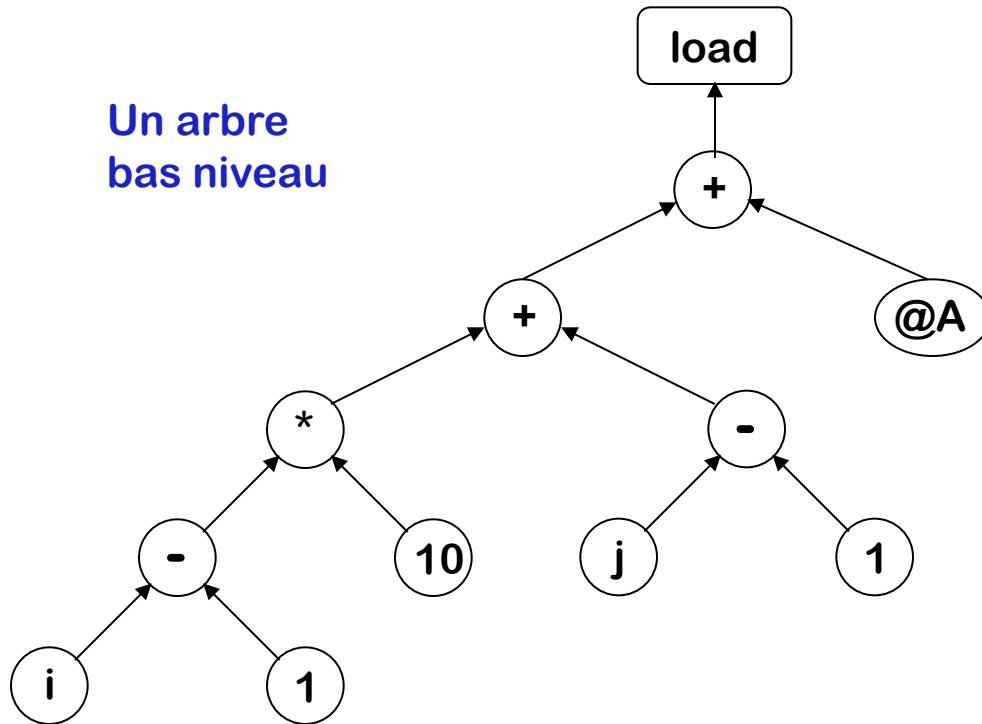
Code linéaire bas niveau :  
Favorable pour le calcul d'adresse d'un élément



# Niveau d'abstraction

- Une RI structurée est souvent considérée comme haut niveau
- Une RI linéaire est souvent considérée bas niveau.
- Ceci n'est pas nécessairement vrai!

Un arbre  
bas niveau

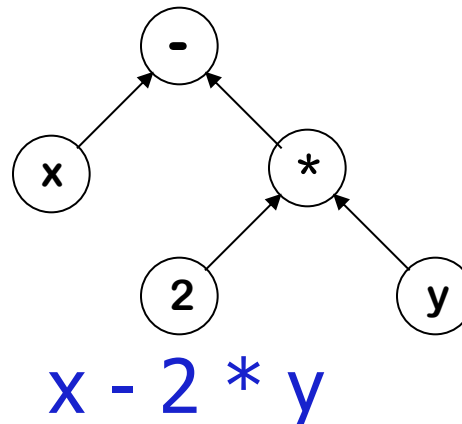


`loadArray A, i, j`

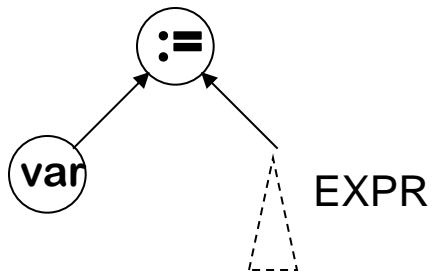
Code linéaire haut niveau

# Arbre abstrait

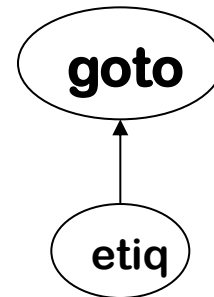
- Arbre abstrait : arbre syntaxique après avoir enlevé les nœuds des non-terminaux.
- Nœuds internes : opérateurs
- Feuilles : opérandes.



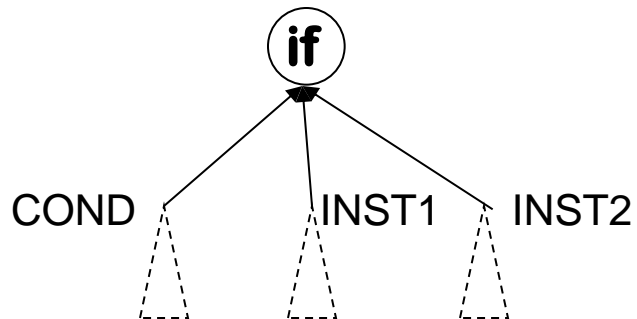
# Arbre abstrait



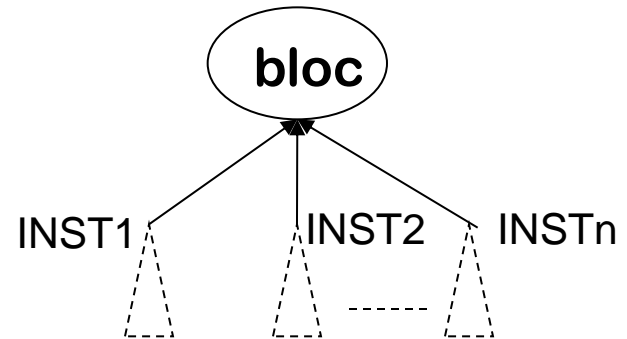
**Var := EXPR**



**goto etiq**



**if cond then INST1  
else INST2**

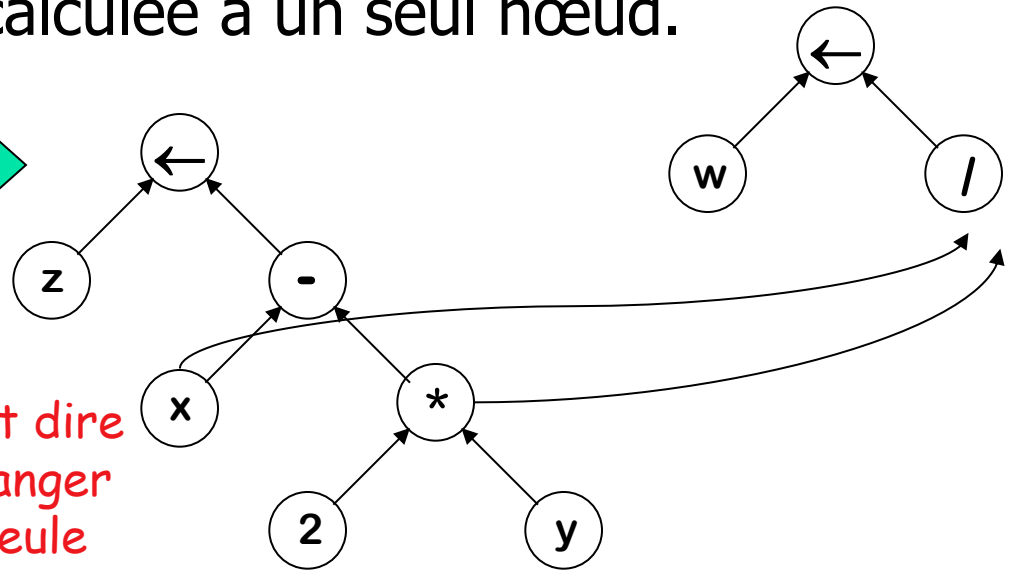
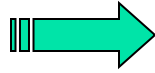


**{INST1; INST2;  
...;INSTn;}**

# Directed Acyclic Graph (DAG)

- C'est une forme optimisée de l'arbre abstrait.
- Chaque valeur calculée a un seul nœud.

$z \leftarrow x - 2 * y$   
 $w \leftarrow x / (2 * y)$

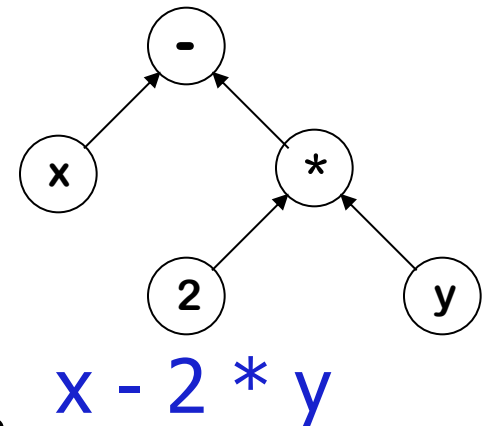


Dans un DAG, si un nœud est utilisé plusieurs fois, cela veut dire que le compilateur peut s'arranger pour qu'il ne l'évalue qu'une seule fois.

- Réutilisation des sous-expressions communes
- Encode la redondance de calcul

# Formes pré et post-fixées

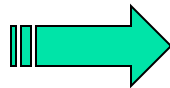
- C'est une linéarisation de l'arbre abstrait. Elles sont définies par un parcours récursif de cet arbre.
- Forme pré-fixée : visiter la racine ensuite le fils gauche suivi du fils droit
  - Ex:  $- x * 2 y$
- Forme post-fixée : visiter le fils gauche suivi du fils droit, en enfin la racine
  - Ex:  $x 2 y * -$



# Code de machine à pile

- Utilisé à l'origine pour des machines n'ayant pas de registres, maintenant utilisé pour le bytecode java (JVM).
- Exemple :

$x - 2 * y$



```
push x  
push 2  
push y  
multiply  
subtract
```

## Avantages

- Les noms utilisés sont *implicites*, et non *explicites*
- Simple à générer et à exécuter.
- Exercice : montrez comment il est facile de générer du code de machine à pile à partir d'une notation post-fixée, et vice-versa.

# Code trois adresses

Il y a plusieurs représentations d'un code à trois adresses

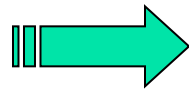
- En général, les instructions à trois adresses ont la forme :

$$x \leftarrow y \text{ op } z$$

avec un seul opérateur (op) et au plus trois opérandes (x, y, z)

Exemple:

$$z \leftarrow x - 2 * y$$



<b>t</b>	$\leftarrow$	2	*	<b>y</b>
<b>z</b>	$\leftarrow$	<b>x</b>	-	<b>t</b>

Caractéristiques :

- Ce code ressemble à celui de plusieurs machines (de moins en moins vrai).
- Introduit un ensemble de variables temporaires
- Forme compacte

# Code trois adresses : Quadruplets

Représentation naïve de code trois adresses

- Table de  $k*4$  entiers
- Structure simple
- Réordonnancer le code plus aisé
- Noms (et numéros de temp) explicites


```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

assembleur RISC



Le compilateur  
FORTRAN d'origine  
utilisait les "quads"

opérateur dest src1 src2



opérateur	dest	src1	src2
load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quadruplets





# Code trois adresses : Triplets

- L'indice de la table est utilisé pour implicitement indiquer les numéros du temporaire contenant le résultat de l'instruction.
- Economie de 25% d'espace comparé aux quads (on a éliminé une colonne)
- Plus difficile de réordonnancer le code

load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	X	
sub	5	4	2

Quadruplets

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(2)

Triplets

# Code trois adresses : Triplets indirects

- Une optimisation des triplets qui utilise deux tables
  - une table contient une liste de triplets distincts
  - une table qui contient le code consistant en un ensemble de numéros de triplets.
- Ce n'est qu'une compression des triplets
- L'économie d'espace n'est pas au rendez-vous mais il est plus facile de réordonnancer le code.

(100)	load	y	
(101)	loadI	2	
(102)	mult	(100)	(101)
(103)	load	x	
(104)	sub	(103)	(102)

100	14	166	79	100	45	101	345
-----	----	-----	----	-----	----	-----	-----

Code : numéros des triplets

Table des triplets distincts



# Triplets vs. Quadruplets

---

- Compromis entre les triplets et les quadruplet → la compacité versus la facilité de manipulation
  - Dans le passé, le temps de compilation et l'espace utilisé par le compilateur étaient des ressources critiques
  - De nos jours, la vitesse de compilation est un facteur plus important
    - Attention ! Empreinte mémoire toujours un souci pour les compilateurs optimisants
  - Les triplets et quadruplets peuvent paraître bien simples par rapport à la complexité des architectures/compilateurs actuels

# Code à deux adresses

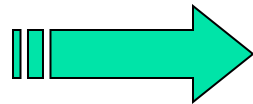
- Les instructions sont de la forme

$x \leftarrow x \text{ op } y$

Il y a un opérateur (*op*) et au plus deux opérandes (*x* et *y*). L'un d'eux est nécessairement détruit (affecté).

Exemple :

$z \leftarrow x - 2 * y$



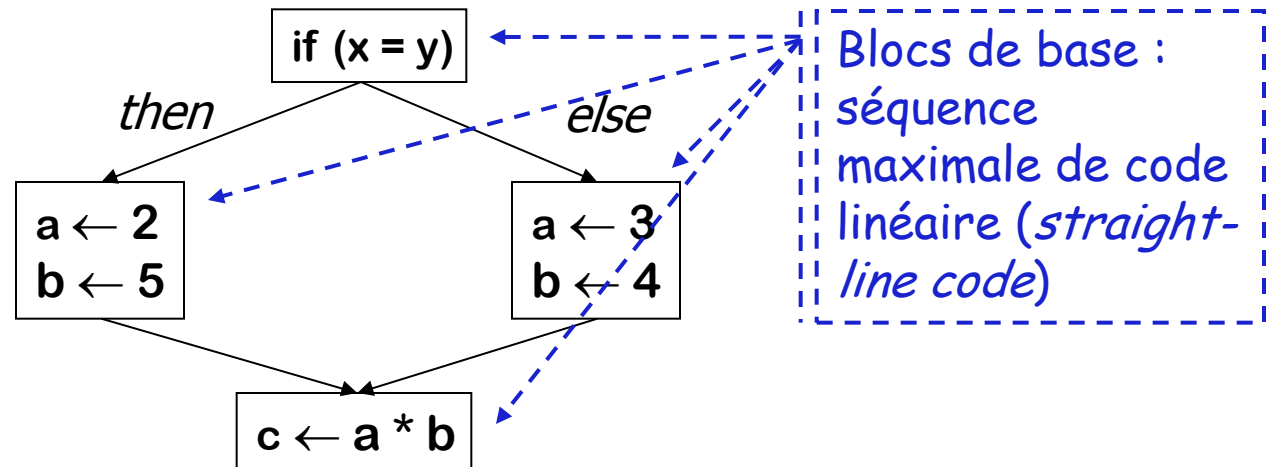
```
t1 ← 2
t2 ← load y
t2 ← t2 * t1
z ← load x
z ← z - t2
```

- Problème
- Beaucoup de machines ne se basent pas sur des opérations destructives (à effet de bord) : machines avec accumulateur
  - Les opérations destructives rendent difficile la réutilisation des temporaires (registres).

# Intermédiaire hybride : Graphe de flot de contrôle

Modélise le transfert de contrôle dans un programme/fonction

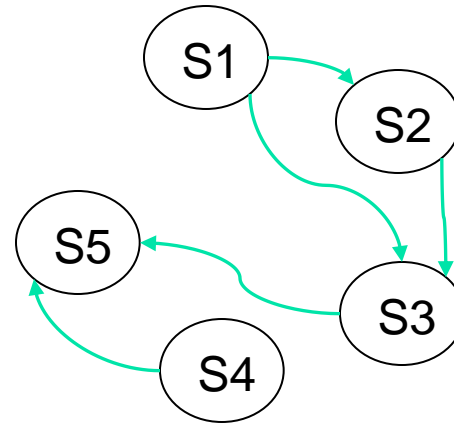
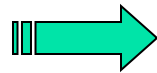
- Les nœuds représentent les blocs de base
  - Les instructions dans les blocs de bases peuvent être des quads, triplets, ou tout autre représentation intermédiaire
- Les arcs représentent le flot de contrôle (branchements, appels de fonctions, etc.)



# Graphe de flot de données

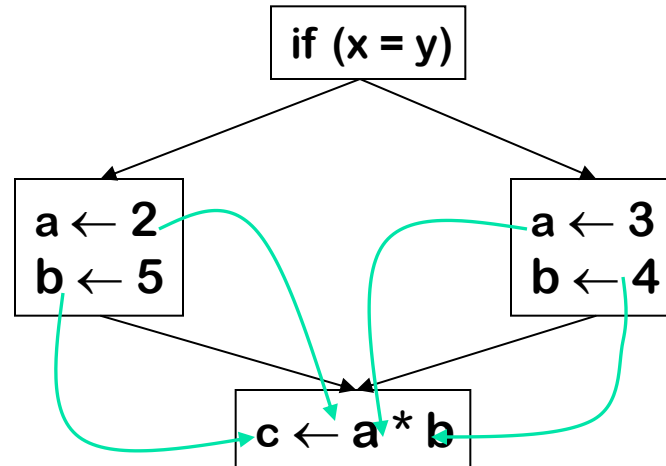
- Les nœuds sont les instructions.
- Un arc de  $a$  vers  $b$  veut dire que l'instruction  $a$  calcule un résultat lu par l'instruction  $b$

S1:	$t_1$	$\leftarrow$	$x+y$
S2:	$t_2$	$\leftarrow$	$t_1 + 2$
S3:	$t_2$	$\leftarrow$	$t_2 * t_1$
S4:	$z$	$\leftarrow$	load $x$
S5:	$z$	$\leftarrow$	$z - t_2$



# Intermédiaire hybride : Graphe de dépendances d'un programme

- C'est le graphe de flot de contrôle auquel on ajoute les arcs de dépendances de données.



Il évite de gérer en parallèle un graphe de contrôle et un graphe de dépendances de données



# Blocs de base (*basic blocks*)

---

Un **bloc de base** est une séquence d'instructions *consécutives* où le flot d'exécution (contrôle) entre au début et ne sort qu'à la fin de la séquence

Seulement la dernière instruction d'un bloc de base peut être un branchement, et seulement la première instruction peut être la destination d'un branchement.





# Algorithme de partition en blocs de base

---

1. Identifier les instructions de tête en utilisant les règles suivantes:
  - (i) La *première instruction* d'un programme est une instruction de tête
  - (ii) Toute instruction qui est une *destination d'un branchement* est une instruction de tête (dans la plupart des langages intermédiaires, ce sont des instructions avec des étiquettes)
  - (iii) Toute instruction qui suit *immédiatement* un branchement est une instruction de tête
2. Le bloc de base correspondant à une instruction de tête correspond à cette instruction, plus toutes les instructions suivantes, jusqu'à la prochaine instruction de tête exclue.



# Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

**Code source**

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
```

**Code trois adresses**



# Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

<b>Règle (i)</b>	(1) prod := 0
	(2) i := 1
	(3) t1 := 4 * i
	(4) t2 := a[t1]
	(5) t3 := 4 * i
	(6) t4 := b[t3]
	(7) t5 := t2 * t4
	(8) t6 := prod + t5
	(9) prod := t6
	(10) t7 := i + 1
	(11) i := t7
	(12) if i <= 20 goto (3)
	(13) ...

Code trois adresses



# Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

<b>Règle (i)</b>	(1) prod := 0
	(2) i := 1
<b>Règle (ii)</b>	(3) t1 := 4 * i
	(4) t2 := a[t1]
	(5) t3 := 4 * i
	(6) t4 := b[t3]
	(7) t5 := t2 * t4
	(8) t6 := prod + t5
	(9) prod := t6
	(10) t7 := i + 1
	(11) i := t7
	(12) if i <= 20 goto (3)
	(13) ...

Code trois adresses



# Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

<b>Règle (i)</b>	(1) prod := 0
	(2) i := 1
<b>Règle (ii)</b>	(3) t1 := 4 * i
	(4) t2 := a[t1]
	(5) t3 := 4 * i
	(6) t4 := b[t3]
	(7) t5 := t2 * t4
	(8) t6 := prod + t5
	(9) prod := t6
	(10) t7 := i + 1
	(11) i := t7
	(12) if i <= 20 goto (3)
<b>Règle (iii)</b>	(13) ...

Code trois adresses



# Exemple

Blocs de base

**B1** (1) `prod := 0`  
(2) `i := 1`

**B2** (3) `t1 := 4 * i`  
(4) `t2 := a[t1]`  
(5) `t3 := 4 * i`  
(6) `t4 := b[t3]`  
(7) `t5 := t2 * t4`  
(8) `t6 := prod + t5`  
(9) `prod := t6`  
(10) `t7 := i + 1`  
(11) `i := t7`  
(12) `if i <= 20 goto (3)`

**B3** (13) ...



# Control Flow Graph (CFG)

---

Un ***graphe de flot de contrôle*** (CFG) est un multi-graphe orienté tel que :

(i) les nœuds sont les blocs de base et  
(ii) les arcs représentent le flot de contrôle (ordre possible d'exécution)

- Le nœud de départ est celui qui contient la première instruction du programme
- Il peut y avoir plusieurs nœuds finaux car on peut avoir plusieurs "exits" dans le programme



# Control Flow Graph (CFG)

---

- Il y a un arc orienté du bloc de base B1 vers le bloc de base B2 dans le CFG si :
  - (1) Il y a un branchement de la dernière instruction de B1 vers l'instruction de tête de B2, ou
  - (2) Le flot d'exécution peut passer de B1 à B2 si:
    - (i) B2 suit immédiatement B1, et
    - (ii) B1 ne finit pas avec un branchement inconditionnel



# Exemple

Graphe de flot de contrôle:

**Règle (2)**

**B1** (1) `prod := 0`  
(2) `i := 1`



**B2** (3) `t1 := 4 * i`  
(4) `t2 := a[t1]`  
(5) `t3 := 4 * i`  
(6) `t4 := b[t3]`  
(7) `t5 := t2 * t4`  
(8) `t6 := prod + t5`  
(9) `prod := t6`  
(10) `t7 := i + 1`  
(11) `i := t7`  
(12) `if i <= 20 goto (3)`

**B3** (13) ...

# Exemple

Graphe de flot de contrôle:

**B1** (1) `prod := 0`  
(2) `i := 1`

**Règle (1)**

**Règle (2)**

**B2**

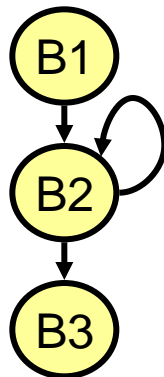
(3) `t1 := 4 * i`  
(4) `t2 := a[t1]`  
(5) `t3 := 4 * i`  
(6) `t4 := b[t3]`  
(7) `t5 := t2 * t4`  
(8) `t6 := prod + t5`  
(9) `prod := t6`  
(10) `t7 := i + 1`  
(11) `i := t7`  
(12) `if i <= 20 goto (3)`

**B3**

(13) ...

# Exemple

Graphe de flot de contrôle:



**B1**  
(1) **prod** := 0  
(2) **i** := 1

**Règle (1)**

**Règle (2)**

**B2**  
(3) **t1** := 4 \* **i**  
(4) **t2** := **a**[**t1**]  
(5) **t3** := 4 \* **i**  
(6) **t4** := **b**[**t3**]  
(7) **t5** := **t2** \* **t4**  
(8) **t6** := **prod** + **t5**  
(9) **prod** := **t6**  
(10) **t7** := **i** + 1  
(11) **i** := **t7**  
(12) if **i** <= 20 goto (3)

**Règle (2)**

**B3**  
(13) ...



# Les CFGs sont des multi-graphes

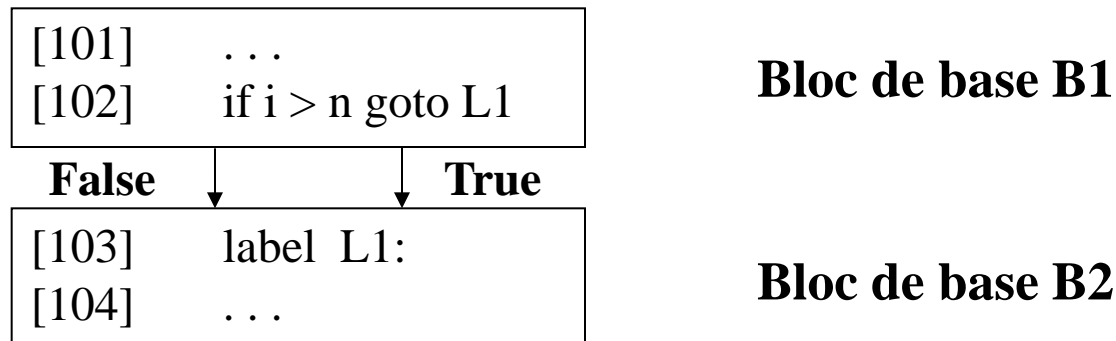
---

*Note* : il peut y avoir plusieurs arcs d'un bloc de base vers un autre dans un CFG.

Donc, en général, un CFG est un multi-graphe.

Les arcs peuvent être distingués par les étiquettes des conditions.

Un exemple trivial ci-dessous:





# Graphe d'appels de fonctions

---

- C'est un graphe complémentaire au CFG
  - Il représente une vision de contrôle globale à l'application
  - Le CFG représente le contrôle dans une fonction
- Un nœud = une fonction
- Un arc entre f1 et f2 ssi f1 appelle f2

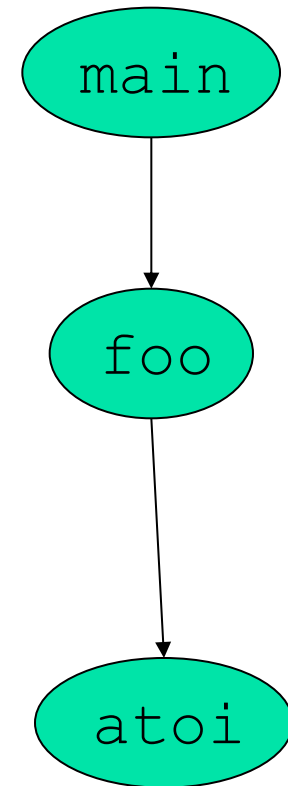


# Graphe d'appels de fonctions

---

```
int main () {  
    ...  
    y=foo(5);  
    ...  
}
```

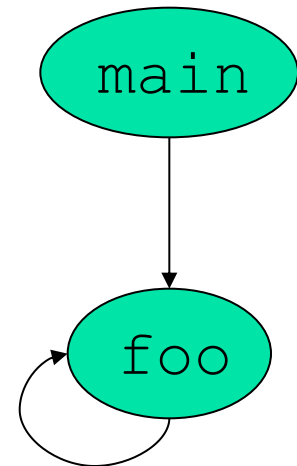
```
int foo (int y) {  
    int x=y+3;  
    if (x-5= 0)  
        x=atoi(...);  
    return x;  
}
```



# Graphe d'appels de fonctions

```
int main () {  
    ...  
    y=foo(5);  
    ...  
}
```

```
int foo (int y) {  
    int x=y+3;  
    if (x-5= 0)  
        x=foo(x-1);  
    return x;  
}
```





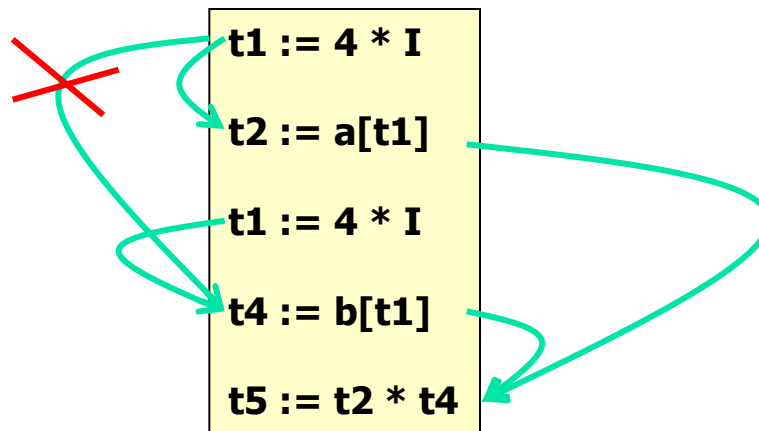
# Flot de données

---

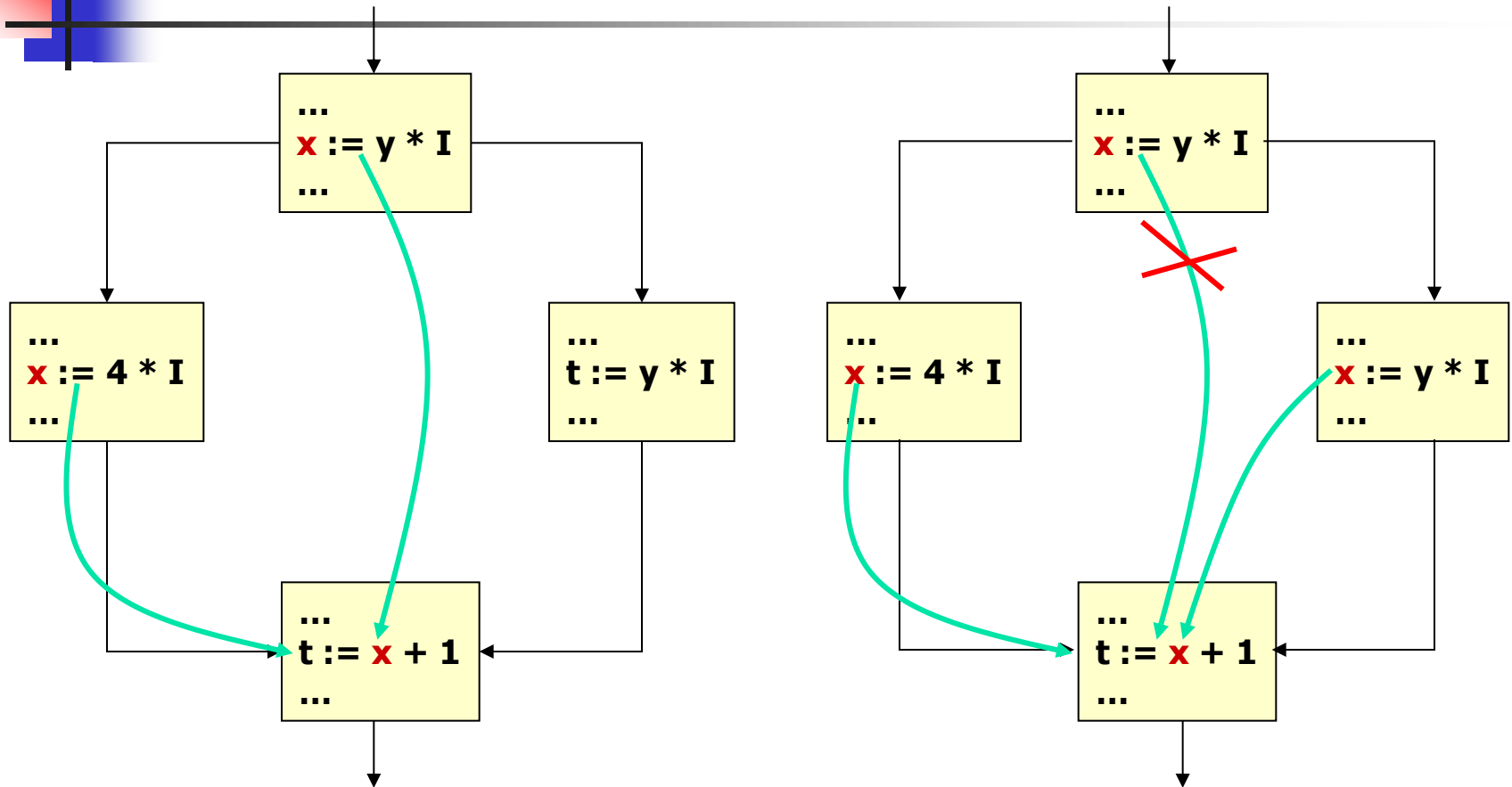
- On dit qu'il y a une dépendance de flot de données entre une instruction  $i_1$  et une instruction  $i_2$  si  $i_1$  produit un résultat lu par  $i_2$ .
  - Notion de dépendance de données vu dans le précédent cours
- Le problème de détection des dépendances de flot de données est indécidable dans le cas général :
  - Existence de pointeurs, de branchements, etc.



# Cas simple : scalaires dans blocs de base



# Cas simple : scalaires dans un CFG





# Plan du cours

---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation



# Représentation intermédiaire et passes

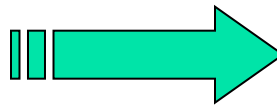
---

- Intérêt de la représentation intermédiaire
  - Permettre d'appliquer des passes sur le code
  - Passe = analyse, transformation ou optimisation
- Application d'une passe
  - Entrée : code en forme intermédiaire
  - Sortie : code transformé en même forme intermédiaire
- Analyse
  - Utilisation en lecture seule de la représentation intermédiaire
  - Mise à jour parfois de certaines informations
- Transformation
  - Modification de la structure du code permettant d'appliquer plus *facilement* une autre passe
  - Exemple : forme SSA (*Static Single Assignment*)

# Forme SSA : *Static Single Assignment*

- Représentation textuelle du graphe de flot de données
- Idée principale : chaque variable est définie/écrite une seule fois
- Utiliser une fonction abstraite appelée  $\phi$ -fonction pour restaurer le flot de données

```
Code d'origine
x ← ...
y ← ...
while (x < k)
    x ← x + 1
    y ← y + x
```



```
forme SSA
    x0 ← ...
    y0 ← ...
    if (x0 > k) goto next
loop:  x1 ←  $\phi(x_0, x_2)$ 
      y1 ←  $\phi(y_0, y_2)$ 
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < k) goto loop
next:  ...
```

Caractéristiques :

- Décrit le flot de données = sémantique
- Nécessite et permet une analyse de code plus pointue
- (parfois) des algorithmes d'analyse et d'optimisation plus rapides



# Passé d'optimisation

---

- But final de chaque optimisation
  - Minimiser ou maximiser une certaine fonction de coût
- Exemples
  - Réduction du temps d'exécution du code (*time to solution*)
  - Réduction de l'empreinte mémoire
  - Réduction de l'énergie consommée
- Modèle de coût dépendant du domaine
  - Calcul haute performance : réduction du temps d'exécution
  - Architecture embarquée (système enfoui) : réduction de l'énergie consommée
- Plusieurs modèles peuvent rentrer en jeu
  - Réduction de l'empreinte mémoire pour le HPC et l'embarqué
- Exemple
  - Réduction du nombre d'instruction
  - Sélection d'une instruction moins coûteuse
    - $x * 2 \rightarrow x + x$  ou  $x \ll 1$



# Passe d'optimisation

---

- Principe : heuristiques avant tout
- Heuristiques
  - Beaucoup d'optimisations sont basées sur des heuristiques
  - La latence mémoire par exemple n'est pas forcément connue au moment de la compilation
  - La position du code en mémoire (problème d'I-Cache)
  - Le nombre de registres potentiel pour un bout de code
- Mais comment décider ?
  - Benchmarks, benchmarks, ...
  - *If you get 1 percent better performance, commit!*
  - Restriction sur le temps de chaque passe (optimisation/transformation) : complexité en  $O(N)$  avec  $N$  le nombre d'instructions



# Compromis

---

- Compromis mémoire versus temps
  - Souvent considérer comme antagonistes puisque
  - Faire un pré-calcul et le stocker en mémoire permet de ne plus refaire le calcul
- Il faut donc décider au niveau de la compilation quelle est la priorité
- Exemple

```
for (i=0 ; i < N; i++) {  
    tab[i] = 5 * a + i ;  
}
```





# Compromis

---

- Compromis mémoire versus temps
  - Souvent considérer comme antagonistes puisque
  - Faire un pré-calcul et le stocker en mémoire permet de ne plus refaire le calcul
- Il faut donc décider au niveau de la compilation quelle est la priorité
- Exemple

```
tmp = 5 * a ;  
for (i=0 ; i < N; i++) {  
    tab[i] = tmp + i ;  
}
```



# Ordre des passes

---

- Mais dans quel ordre appliqué les passes ?
  - L'ordre a son importance
- Souvent une optimisation va modifier le code
- Mais l'optimisation qui va suivre n'aime pas forcément le nouveau format
- Beaucoup de recherche a été faite pour déterminer le meilleur ordre
  - Aucune solution n'est parfaite
  - Cela dépend de l'application, de la fonction, de la boucle
- Nous en revenons aux heuristiques, aux tests, à l'intuition
  - Notion de *pass manager* dans les compilateurs



# Ordre des passes

---

- Exemple

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 5 * a + i ;  
}
```

- Passes choisies

- Dead code elimination
- Propagation de constante

- Après dead code elimination

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 5 * a + i ;  
}
```

- Après constant propagation

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 15 + i ;  
}
```



# Ordre des passes

---

- Exemple

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 5 * a + i ;  
}
```

- Passes choisies

- Propagation de constante
- Dead code elimination

- Après constant propagation

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 15 + i ;  
}
```

- Après dead code elimination

```
for (i=0 ; i < N; i++)  
{  
    tab[i] = 15 + i ;  
}
```



# Compilateurs GCC

---



# Plan du cours

---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation



# Chaîne de compilation GNU

---

- GCC : GNU Compiler Collection
  - Historiquement GNU C Compiler
- Ensemble d'outils et de bibliothèque pour la compilation
  - Plusieurs langages, plusieurs architectures
  - Générateur de compilateurs !
- Disponible sous licence GPL
  - <http://gcc.gnu.org>
- Support principal des TDs/TPs !



# Historique de GCC

---

- 0.9 : 22 Mars 1987
  - Première version beta
- GCC 1.0 : 23 Mai 1987
- GCC 3.0 : 18 Juin 2001
  - Ajout du support du langage JAVA
- GCC 4.0 : 20 Avril 2005
  - Ajout de la branche *tree-ssa*
  - Ajout de l'algorithme de pipeline logiciel *Swing Modulo Scheduling (SMS)*
  - Représentation intermédiaire GIMPLE
- GCC 4.2.0 : 13 Mai 2007
  - Support de OpenMP pour C, C++ et Fortran
- GCC 4.5.0 : 14 Avril 2010
  - Optimisations au *link* (LTO)
- GCC 4.6.0 : 25 Mars 2011
  - Réduction de l'empreinte mémoire / meilleure exploitation du cache
  - Ajout de nouveaux langages : CAF et GO
- GCC 4.7.0 : 22 Mars 2012
  - OpenMP 3.1
  - Standard C++11
- GCC 4.8.0 : 22 Mars 2013
  - Programmation en partie en C++ 2003
  - Support intégral du standard C++11





# Historique de GCC

---

- GCC 4.9.0 : 22 Avril 2014
  - OpenMP 4.0
  - Amélioration des diagnostics (incluant de la couleur)
  - Support expérimental pour C++14
  - Go 1.2.1
  - Support AVX-512
- GCC 5.1 : 22 Avril 2015
  - Amélioration du support C++ 14
  - OpenMP 4.0 offloading
  - Implémentation préliminaire pour OpenACC 2.0
  - Support spécifiques pour les architectures Intel Xeon Phi
  - Go 1.4.2
- GCC 5.2 : 16 Juillet 2015
  - Support du mot clé « vector »
  - Support amélioré pour les instructions AMD
  - Support du processeur IBM z13
- GCC 5.3 : 4 Décembre 2015
  - Support du processeur Intel Skylake avec AVX-512
  - Support des processeurs IBM z pour le langage GO
- GCC 6.1 : 27 Avril 2016
  - OpenMP 4.5
  - Amélioration du support de OpenACC 2.0
  - Support expérimental pour C++ 17
- Version courante : 6.2 sortie le 22 Août 2016
  - Support SPARC



# Survol des fonctionnalités

---

- Langages supportés
  - C, C++
  - Objective-C, Objective-C++
  - JAVA,
  - Fortran
  - ADA
- Processeurs supportés
  - ARM, IA-32 (x86), x86-64, IA-64, MIPS, SPARC, ...
- Système de *plugins* pour ajouter/modifier des passes de compilation
- Combien de lignes de code pour GCC ?



# Evolution de la taille de GCC

Count		GCC 4.3.0	GCC 4.4.2	GCC 4.5.0
Lines	Main source	2,029,115	2,187,216	2,320,963
	Libraries	1,546,826	1,633,558	1,671,501
	Subdirectories	3,527	3,794	4,055
Files	Number of files	57,660	62,301	77,782
	C source files	15,477	18,225	20,024
	Header files	9,646	9,213	9,389
	C++ files	3,708	4,232	4,801
	Machine description	186	206	229

(Line counts estimated by David A. Wheeler's sloccount program)



# Taille de GCC 4.6.2

Language	Files	Code	Comment	Comment %	Blank	Total
c	18624	2106311	445288	17.5%	419325	2970924
cpp	22206	989098	230376	18.9%	215739	1435213
java	6342	681938	645505	48.6%	169046	1496489
ada	4616	680251	316021	31.7%	234551	1230823
autoconf	91	405517	509	0.1%	62919	468945
html	457	168378	5669	3.3%	38146	212193
make	98	121136	3658	2.9%	15555	140349
fortranfixed	2989	100688	1950	1.9%	13894	116532
shell	148	48032	10451	17.9%	6586	65069
assembler	208	46750	10227	17.9%	7854	64831
xml	75	36178	282	0.8%	3827	40287
objective_c	869	28049	5023	15.2%	8124	41196
fortranfree	831	13996	3204	18.6%	1728	18928
tex	2	11060	5776	34.3%	1433	18269
scheme	6	11023	1010	8.4%	1205	13238
automake	67	9442	1039	9.9%	1457	11938
perl	28	4445	1316	22.8%	837	6598
ocaml	6	2814	576	17.0%	378	3768
xslt	20	2805	436	13.5%	563	3804
awk	11	1740	396	18.5%	257	2393
python	10	1725	322	15.7%	383	2430
css	24	1589	143	8.3%	332	2064
pascal	4	1044	141	11.9%	218	1403
csharp	9	879	506	36.5%	230	1615
dcl	2	402	84	17.3%	13	499
tcl	1	392	113	22.4%	72	577
javascript	4	341	87	20.3%	35	463
haskell	49	153	0	0.0%	17	170
bat	3	7	0	0.0%	0	7
matlab	1	5	0	0.0%	0	5
Total	57801	5476188	1690108	23.6%	1204724	8371020

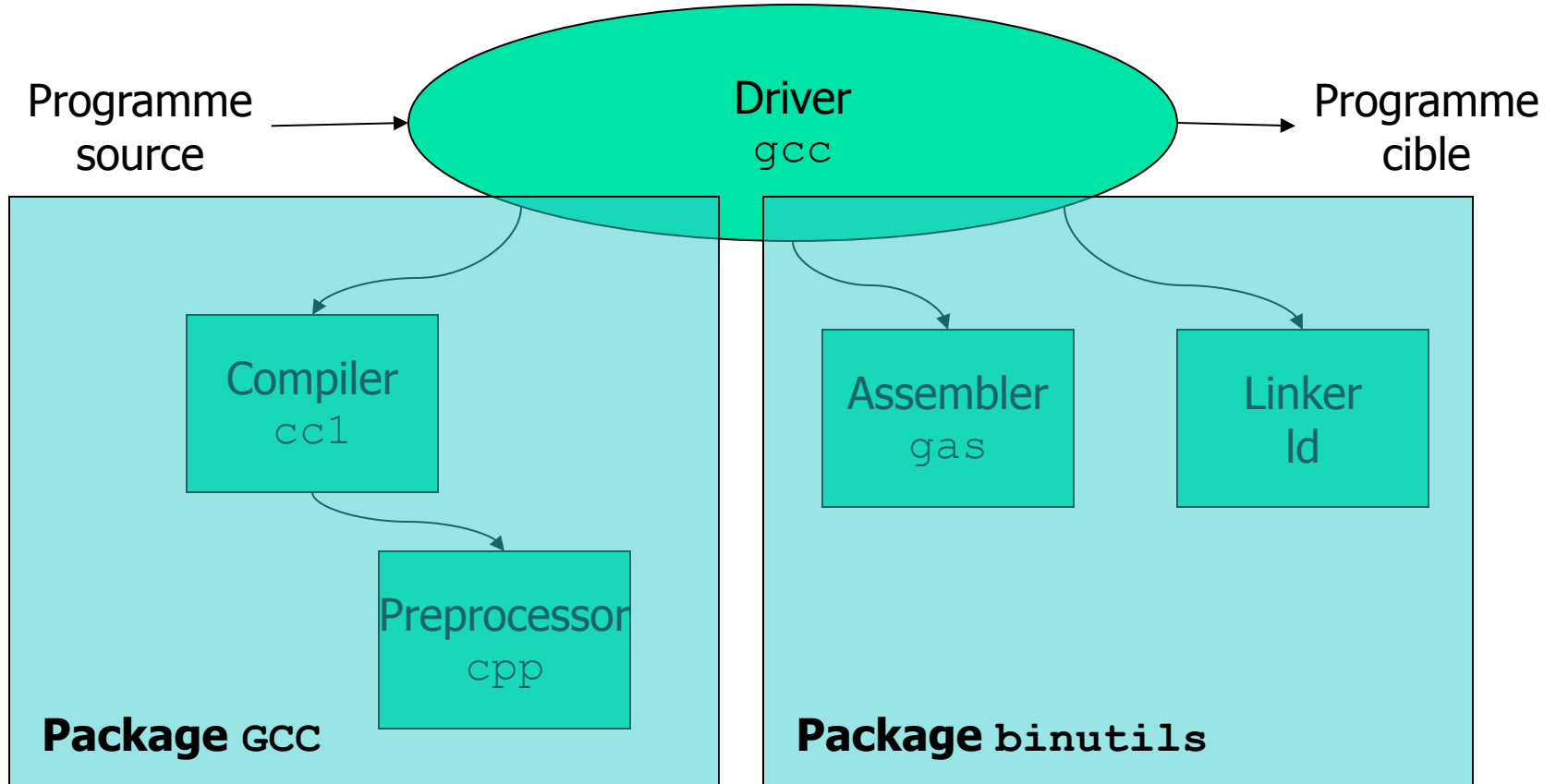


# Plan du cours

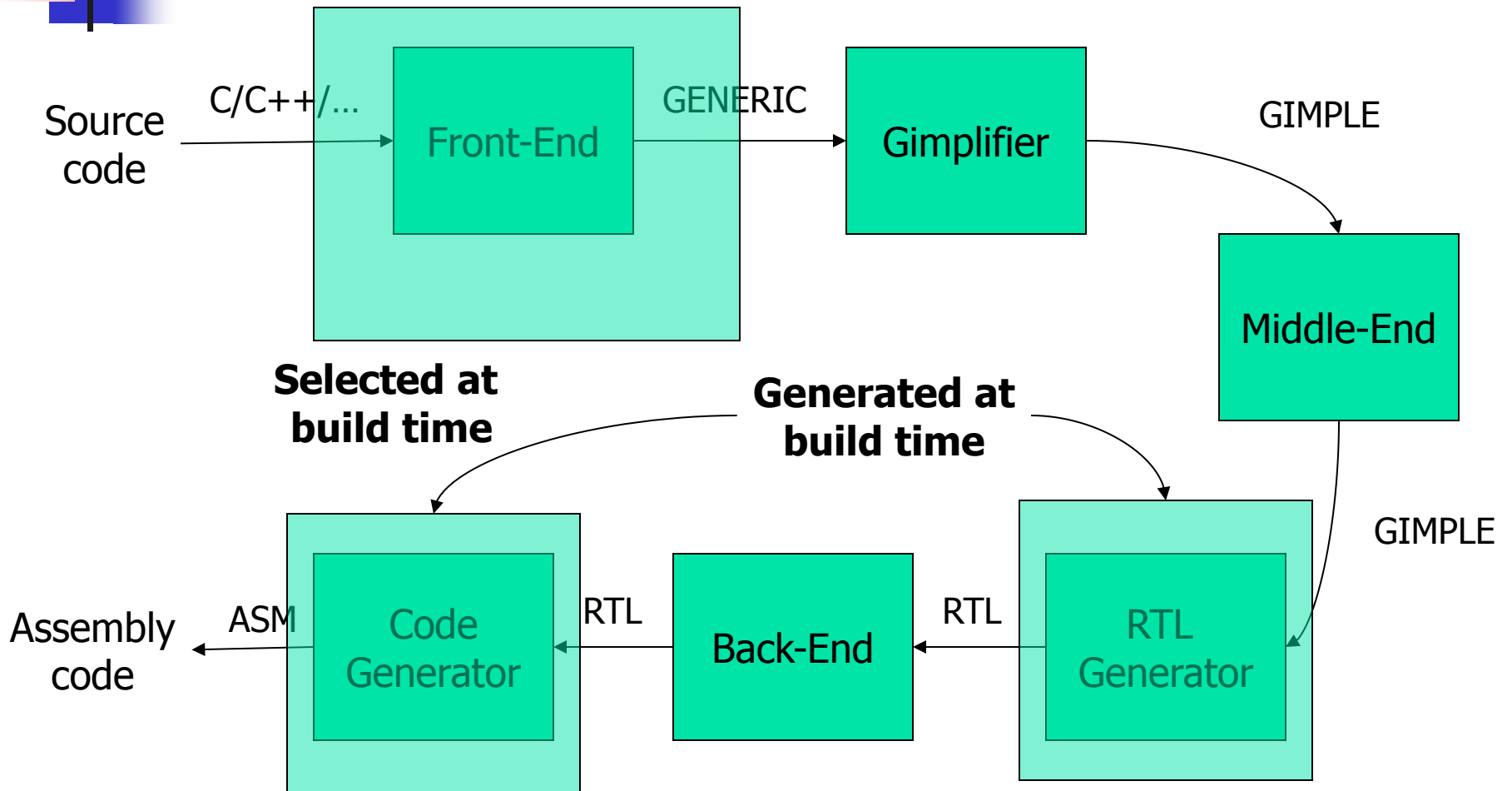
---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation

# Architecture de GCC



# Architecture de GCC





# Transformations dans GCC

---

- GCC possède un total de 203 passes de transformations
- Le nombre total de passes effectuées lors d'une compilation est 239
  - Certaines transformations sont appelées plusieurs fois
- Pour l'enchaînement des transformations sur les représentations intermédiaires, GCC utilise un *pass manager*
  - Situé dans les fichiers `${SOURCE}/gcc/passes.c` et `${SOURCE}/gcc/passes.def`





# Transformations GIMPLE

---

Pass Group	Number of passes
Lowering	12
Interprocedural optimizations	49
Intraprocedural optimizations	42
Loop optimizations	27
Remaining intraprocedural optimizations	23
Generating RTL	01
Total	154



# Transformations RTL

---

Pass Group	Number of passes
Intraprocedural Optimizations	21
Loop optimizations	7
Machine Dependent Optimizations	54
Assembly Emission and Finishing	03
Total	85



# Préprocesseur

---

- CPP : Gestion des directives de précompilation
- Syntaxe des directives
  - #keyword
- Exemple de directives
  - #ifdef
  - #include
  - #warning
  - #error
- Explosion de la taille du code après *preprocessing*
- Attention #pragma n'est pas traité par le préprocesseur



# Front-end

---

- Lecture du fichier source en entrée
  - C, C++, Fortran, Java, C#, ...
- Vérification de la validité du code
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
  - Cf. CPA cours 1
- Chaque front-end est dans un répertoire différent :
  - C, ObjectiveC → `${SOURCE}/gcc/c/`, `${SOURCE}/gcc/c-family/`
  - C++ → `${SOURCE}/gcc/cp/`, `${SOURCE}/gcc/c-family/`
  - Fortran → `${SOURCE}/gcc/fortran/`
- En sortie, le code est représenté en GENERIC
  - Sauf pour C/C++ qui génère directement du GIMPLE



# GENERIC

---

- Représentation intermédiaire sous forme d'arbre
- Indépendant du langage source
- Processus de création d'une représentation GENERIC
  - Génération de l'arbre de syntaxe abstraite par le *parser*
  - Le parser peut garder cette représentation
  - Suppression des constructions spécifiques au langage
  - Emission de l'arbre GENERIC à la fin de la phase de *parsing*
- Tous les noeuds sont définis dans  
`$(SOURCE)/gcc/tree.def`
  - Notion de *tree codes*



# Middle-end

---

- Optimisation haut niveau
  - Indépendante de l'architecture
- Granularités
  - Optimisation par fonction
  - Optimisation par boucle
  - Optimisation inter-procédurale
- Ordre des transformations géré par le *pass manager* de GCC
- Travail sur une représentation intermédiaire nommée GIMPLE
  - En conjonction avec d'autres RIs (par exemple CFG)
    - Détails dans le prochain cours



# GIMPLE

---

- Représentation intermédiaire de haut niveau
  - Introduite dans GCC 4.4
  - Basée sur une représentation avec un arbre
  - Nœud avec une sémantique
- Sous-ensemble simplifié de GENERIC
  - Représentation 3-adresses
  - Aplatissement du flot de contrôle
  - Simplifications et nettoyage (la grammaire est restreinte)
  - Transformation de GENERIC vers GIMPLE
    - `gimplify_function_tree()` dans le fichier `gimplify.c`
- Deux niveaux de GIMPLE
  - *High GIMPLE*
  - *Low GIMPLE*



# GIMPLE – Exemple en C

---

- Exemple simple
  - Langage C
  - Une seule fonction main
- Compilation avec sortie des fichiers intermédiaires :
  - `gcc -fdump-tree-all test.c`
  - Génération de la représentation GIMPLE entre les transformations

```
int main() {  
    int x = 10 ;  
    if (x) {  
        int y = 5 ;  
        x = x*y+15 ;  
    }  
}
```





# GIMPLE – Exemple en C

Fichier test.c:

```
int main() {  
    int x = 10 ;  
    if (x) {  
        int y = 5 ;  
        x = x*y+15 ;  
    }  
}
```

- Déclaration de temporaires
  - D.2720
- Simplification pour le code 3 adresses
  - D.2720 = x\*y
- Flot de contrôle avec goto

Fichier test.c.004t.gimple:

```
main() {  
    int D.2720;  
    int x;  
    x = 10 ;  
    if (x!=0) goto <D.2718>;  
    else goto <D.2719>;  
    <D.2718>:  
    {  
        int y;  
        y=5;  
        D.2720 = x*y;  
        x = D.2720+15  
    }  
    <D.2719>:  
}
```



# GIMPLE – Exemple en C

- Génération du code GIMPLE

- `gcc -fdump-tree-all-raw test.c`

Fichier test.c.004t.gimple:

```
main() {
  int D.2720;
  int x;
  x = 10 ;
  if (x!=0) goto <D.2718>;
  else goto <D.2719>;
<D.2718>:
{
    int y;
    y=5;
    D.2720 = x*y;
    x = D.2720+15
}
<D.2719>:
}
```

Fichier test.c.004t.gimple:

```
main()
gimple_bind <
  int D.2720;
  int x;
  gimple_assign<integer_cst,x,10,NULL
>
  gimple_cond <ne_expr, x, 0,
<D.2718>, <D.2719> >
  gimple_label <<D.2718>>
  gimple_bind <
    int y;
    gimple_assign<integer_cst, y,
5, NULL>
    gimple_assign<mult_expr,
D.2720, x, y>
    gimple_assign<plus_expr,x,
D.2720,15>
  >
  gimple_label<<D.2719>>
>
```



# GIMPLE – Exemple en C

---

**Fichier test.c.004t.gimple:**

```
main() {
  int D.2720;
  int x;
  x = 10 ;
  if (x!=0) goto <D.2718>;
  else goto <D.2719>;
<D.2718>:
{
    int y;
    y=5;
    D.2720 = x*y;
    x = D.2720+15
}
<D.2719>:
}
```

**Fichier test.c.011t.cfg**

```
main() {
  int y;
  int x;
  int D.2720;

<bb2>:
  x=10;
  if (x!=0) goto <bb 3>;
  else goto <bb 4>;

<bb 3>:
  y=5;
  D.2720 = x*y;
  x=D.2720+15;

<bb 4>:
  return ;
}
```



# GIMPLE - *tree code*

---

- Tous les *tree code* de GCC (152) sont listés dans  
\$(SOURCE)/gcc/tree.def
- Binary Operator
  - MAX EXPR
- Comparison
  - EQ EXPR, LT EXPR
- Constants
  - INTEGER CST, STRING CST
- Declaration
  - FUNCTION DECL, LABEL DECL, VAR DECL
- Expression
  - PLUS EXPR, ADDR EXPR
- Reference
  - COMPONENT REF, ARRAY RANGE REF
- Statement
  - GIMPLE MODIFY STMT, RETURN EXPR, COND EXPR, INIT EXPR
- Type
  - BOOLEAN TYPE, INTEGER TYPE
- Unary
  - ABS EXPR, NEGATE EXPR



# GIMPLE - Transformations

---

- Un compilateur comporte un grand ensemble de transformations de haut niveau
  - Notion de middle-end
- On peut citer quelques exemples :
  - Déroulage de boucle
  - Vectorisation
  - Factorisation de code
  - ...
- Les compilateurs introduisent des options pour définir des ensembles de transformations
  - -O2, -O3, ...
- Dans quel ordre utiliser ces transformations ?



# Pass Manager

---

- GCC utilise un *pass manager* pour enchaîner les différentes transformations
- Dépendant du niveau d'optimisation
  - Ainsi que des options de compilation
- Depuis GCC 4.5
  - Souplesse du *pass manager*
  - Possibilité de créer des plugins pour ajouter une transformation
  - Détails dans le prochain cours



# Pass Manager

---

- Construction d'un arbre de transformations dans la fonction `init_optimization_passes()` dans le fichier `passes.c`
- Exemple : *lowering passes*

```
NEXT_PASS(pass_warn_unused_results)
NEXT_PASS(pass_diagnose_omp_blocks)
NEXT_PASS(pass_mudflap_1);
NEXT_PASS(pass_lower_omp);
NEXT_PASS(pass_lower_cf);
```



# Back-end

---

- Rôles principaux
  - Optimisations dépendantes de l'architecture
  - Génération finale du code assembleur
- Travaille sur une représentation intermédiaire nommée RTL
  - *Register Transfer Language*
- Utilise une représentation de la machine
  - Notion de *machine description*





# RTL

---

- Briques de base : object RTL
  - Expressions
  - Integers
  - Wide integers
  - Strings
  - Vectors
- Chaque expression a un code
  - La liste des codes est défini dans le fichier `rtl.def`
  - Macro pour connaître le code d'une expression : `GET_CODE (x)`



# RTL

---

- Exemple d'affectation

- `DEF RTL_EXPR (SET, "set", "ee",  
RTX_EXTRA)`

- Deux opérandes

- 1. Destination (registre, mémoire, ...)
  - 2. Valeur

- Macro

- 1. Nom interne (majuscules par convention)
  - 2. Nom ASCII (minuscules par convention)
  - 3. Format d'affichage (documenté dans rtl.c)
    - 1. 'e' définit un pointer vers une expression

# RTL

Instruction  
précédente/  
courante/  
suivante

RTL code

Exemple : expression b  
est contenu dans le registre reg.SI 60

```
(insn 7 6 8 test.c:2 (set  
  (reg:SI 59)  
  (plus:SI (reg:SI 60)  
    (const_int 3 [0x3]))))  
-1 (nil))
```

Type de  
destination

Sous-expression  
(addition)



# Assembleur

---

- Le cœur du compilateur génère un fichier assembleur ASCII à la fin de la chaîne de compilation (sortie du back-end)
- Outil assembleur : traduction ASCII vers binaire
  - Simple traduction
- GCC utilise l'assembleur du système d'exploitation : `GAS` (package `binutils`)



# Linker

---

- Collecte des fichiers objets pour la création de l'exécutable final
- Mise à jour des symboles pour les appels dynamiques
- Finalisation de quelques optimisations (par exemple *Thread Local Storage* ou TLS)



# Plan du cours

---

- Structure générale d'un compilateur
  - Vision d'un compilateur
  - Représentation intermédiaire
  - Notion de passes d'optimisation et de transformation
- Présentation de GCC
  - Introduction
  - Structure générale
  - **Installation**



# Installation de GCC

---

- Site web (documentation, téléchargement, ...)
  - GCC : [http ://gcc.gnu.org/](http://gcc.gnu.org/)
  - Version 6.1 actuellement

- Dépendances (bibliothèques)

- GMP
  - MPFR
  - MPC

- Configuration

- Création d'un sous-répertoire travail

```
./configure --prefix=chemin-vers-travail --enable-languages=c,c++ --  
enable-plugin
```

- Compilation

- `make && make install`



# Installation de GCC

---

- Après l'étape `make install`
  - GCC est installé dans le répertoire donné avec l'option `-prefix` lors de la configuration
- Utilisation
  - Modification du PATH

```
export PATH=chemin-vers-travail/bin:$PATH
```
  - `gcc -v` devrait vous donner la version 6.1 et la ligne de configuration que vous avez mis
- Modification du compilateur
  - On modifie ce qu'on veut et ensuite

```
make && make install
```





# Documentation de GCC

---

- Documentation principale
  - Le code de GCC
- Important : il faut pouvoir lire le code de GCC pour comprendre comment cela fonctionne
  - Ne pas hésiter à parcourir les fichiers sources du cœur du compilateur
- Souvent la solution existe dans une autre partie de GCC
- Autre documentation de référence
  - The GCC internals
  - [http ://gcc.gnu.org/onlinedocs/gccint/Plugins.html#Plugins](http://gcc.gnu.org/onlinedocs/gccint/Plugins.html#Plugins)
- Exemple du PDF...



# Conclusion

---

- Structure générale d'un compilateur
  - 3 parties : Front End, Middle End et Back End
- Front end : dépend du langage en entrée et génère un code en une représentation intermédiaire
- Middle-end : application de passes (analyses/transformations/optimisations) et génération de multiples autres représentations intermédiaires
- Back-end : génération de code (langage cible)
- Ordre des passes :
  - Problème connu
  - Dépend du but du compilateur
  - Dépend de l'application