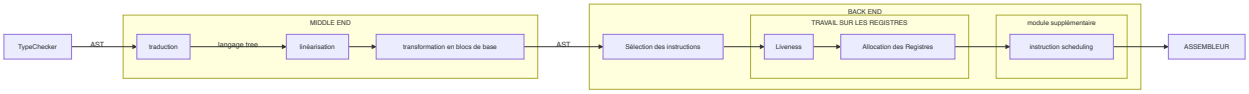
Revisions CMP2

```
Paviel Schertzer
Ing1 2022 EPITA

Reprise du cours de Etienne Renault : Revision de CMP2 (02/07/2020)
```



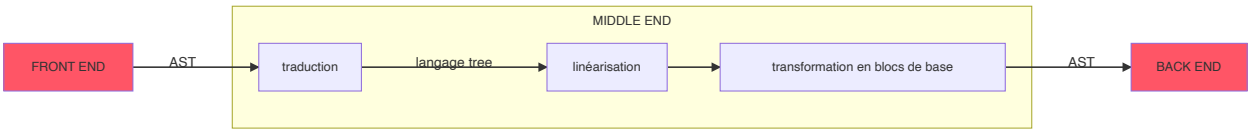
Middle-end a pour but d'obtenir un code générique, avec toutes les optimisations communes à tous les langages.

Back-end on commence à diverger en fonction des micro-processeurs que l'on vise.

Middle-End

À la fin du type checker, on sait que l'AST qui est renvoyé est un AST saint, on a pas de problèmes de types, et que toutes les fonctions sont déclarées tout comme les variables.

On arrive donc au middle-end avec des traitement génériques. Si mon compilateur prend du java, C++, C, et qu'il traduit le C vers Assembleur, et d'autres traducteurs, pour éviter d'avoir une multiplicité de résultats, il va tout aggréger en mettant cet AST sous une forme beaucoup plus linéaire. Quand on vise l'assembleur, ce dernier est très linéaire. On doit être capable de linéariser l'AST, traduire vers de blocs de base.



1. Traduire cet AST vers un `AST de tree` comme appelé dans le cours, plus adapté à une traduction générique.
2. Linéariser le code
3. Traduire vers de blocs de base

Langage `Tree`

Le `langage tree` est un langage très pauvre. Il est composé de

- `jump`
- `labels`
- `opérations binaires`
- `appels de fonctions`

- `combinaisons d'expressions`
- `combinaisons de statements`

Linéarisation

Une fois que l'on a fait une traduction, on obtient un AST Tree. Il faut alors faire une étape de linéarisation.

> Quand on a $1 + 2 \times 3$ on a une grosse arborescence qui se construit, il faut la casser. Quand on a `f  + 1 + 2`, il va faloir casser cet appel à `f`. L'idée est donc de casser cette arborescence.
>
> On fait monter progressivement les expressions et instructions imbriquées progressivmeent jusqu'à la racine. Le code est alors naturelement linéarisé. On a encore de l'arborescence à quelques endroits, mais c'est déjà un objectif de faire remonter ces instructions et expressions jusqu'au top level.

On remonte donc au top-level. Le problème de cette linéarisation et que l'on doit tout de même conserver la sémantique du programme. C'est à dire que si l'on a beaucoup d'imbrications, et un `t=0` à un endroit, et qu'on le remonte trop haut, il risque d'écraser une définition précédente. Remonter plus haut risque d'écraser avec d'anciennes valeurs de variables.

La **linéarisation se base donc sur l'idée de la commutativité**. On va prnendre un sous-arbre, regarder si il commute avec le sous-arbre d'à côté, et à partir de là, si oui, on va pouvoir être capable de relinéariser les choses.

Transformation en blocs de base

Une fois que l'on a linéarisé notre code, la dernière étape est la transformation en blocs de base.

Cette transformation en blocs de base est nécessaire pour deux raison :

- Quand on regarde les micro-processeurs actuels, ils ont pas de `if` `then` `else`. Ils n'ont que `si jump`, `ma condition`, `un label`. Cela veut donc dire que si ma condition est réalisée, je vais sauter au label `LX`, sinon je vais aller à la ligne juste après.

```
IF JUMP
      CONDITION LX,
LY
```

Un bloc de base aura la structure suivante :

```
IF JUMP       CONDITION LX,   ELSE LY
```

Ce que l'on souhaite faire, c'est le prendre le `else`, donc le `LY` et le coller juste en dessous. On va donc prendre notre code, et le tronçonner afin d'obtenir des blocs de base.

Parfois on a des blocs qui vont commencer sans label et finir sans jump, dans ce cas on va les rajouter de manière à obtenir des blocs de base. On va pouvoir les bouger à volonté à l'intérieur de notre programme.

Back-End

Son but est de finir la linéarisation, car il y a des dépendances par rapport à l'assembleur : typiquement pour un appel de fonction, certains micro-processeurs ont des registres dédiés pour passer les arguments, d'autres en ont moins. On va donc devoir choisir les registres après avoir choisi l'architecture de notre micro-processeur.

Instruction selection

On choisit un micro-processeur et on regarde ses registre. Cela nous permet de faire l'**instruction selection**, donc la sélection des instructions. Autrement dit, je vais essayer de prendre les instructions en assembleur qui existent, et d'aller les appliquer sur l'AST qui est en sortie du *middle-end*, pour essayer de trouver quelle instruction marcherait bien.

En faisant ça, on a un problème : normalement quand on produit du code assembleur, on saut que la variable `a` est dans le `registre 2`, et que la bariable `b` est dans le registre `3` et ainsi de suite. Le problème que l'on a là, c'est que l'on ne sait pas. On a va donc faire la sélection des instructions *en supposant que l'on a un nombre de registres illimités*.

Du coup, le processus de sélection des instructions va se réduire uniquement à *"J'ai un AST, et j'ai un ensemble d'instructions, et j'essaye de couvrir cet AST par mon jeu d'instructions. "*

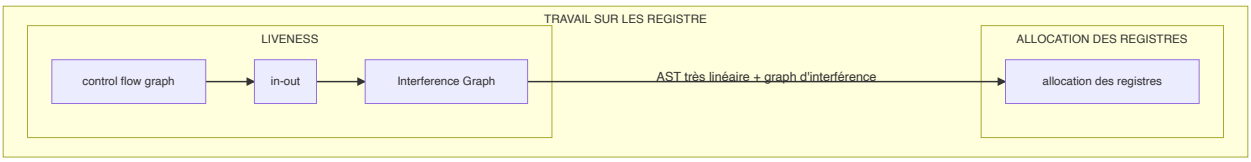
Autrement dit, *"je prend mon jeu d'instructions, je le transfome en tuiles, et je dois essayer de paver mon arbre avec ces différnetes tuiles"*. Pour ce faire, on a **deux stratégies** :

- bottom-up : approche dynamique
- top-down : approche gloutonne

Cela nous permet de trouver un pavage pour notre arbre. De la même manière, il y a plein de pavages possibles, par exemple $1+2$ on peut soit mettre `1` dans un registre et `2` dans un autre pour finalement faire la somme des deux registre, ou bien utiliser les `immédiats`, qui me permettent de faire directement le calcul.

Qu'importe la solution retenue, on se retrouve avec **un code qui n'est plus générique**.

Travail sur les registres

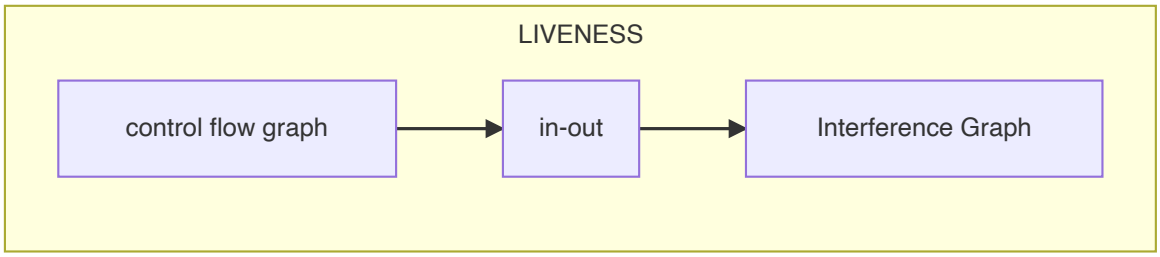


C'est à dire, de dire la variable `a` se retrouve dans tels registre, et la `b` dans tels registre. C'est plus compliqué que prévu, car on doit regarder quand notre variable naît, et meurt. C'est la **liveness**.

Au lieu de tout faire d'un seul coup, on le fait en deux :

- Liveness (analyse de vivacité)
- L'allocation des registres

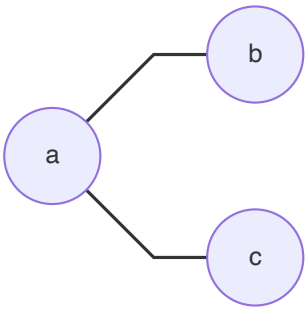
Liveness (analyse de vivacité)



On va construire un **CFG : control flow graph** , qui va représenter l'intégralité des chemins d'execution possibles.

À partir de ce control flow graph, on va calculer nos `Live - in` et `Live - out` (on va avoir des équations qui vont nous permettre de savoir la durée de vie de nos variables en se basant sur le control flow graph).

On sait maintenant quand `a` naît, `b` naît, `a` meurt, et `b` meurt. On peut en déduire le **IG : graph d'interférene**, qui va nous permettre de dire *"la variable a et b vivent simultanément"*, et donc que `a` et `b` ne peuvent pas habiter le même registre.

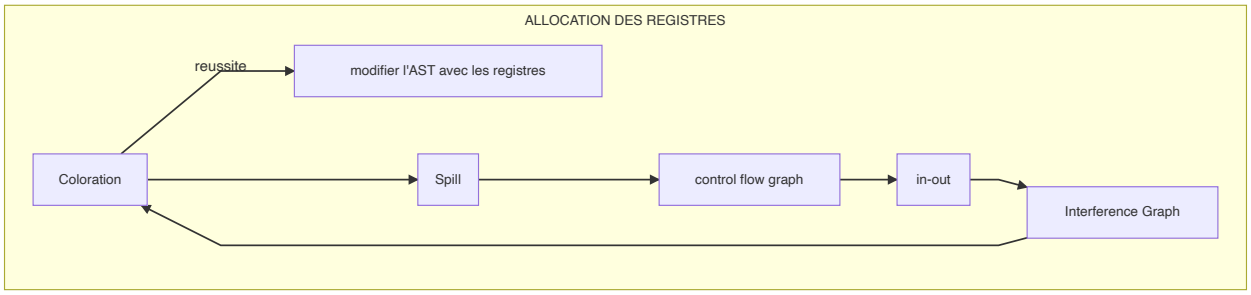
> en gros on a un graph comme en THEG, et un traît qui relie `a` et `b`
>
> 
>
>
> Avec une variable c par exemple, où `a` est en conflit avec mais pas `b`, on a :
>
> 
>

Trouver un registre pour mes variables se résume à seulement être capable de trouver un registre pour `b` différent de `a` et pour `c` différent de `a`.

Allocation des registres

On se retrouve avec un AST très linéaire (il peut être vu comme une simple liste d'instructions), et un graph d'interférences. Le but est de travailler sur ce graph d'interférence pour trouver un registre à chacune des variables, et une fois que l'on aura trouvé un registre à chacune de ces variables, on va simplement aller propager ces modifications sur l'AST qu'on a. On y remplace alors les variables par les registres. C'est l'étape d'**allocation des registres**.

Bon quand on a assez de registres, comme dans l'exemple du `a, b, c`, trois registres sont suffisants. On commence à avoir des problèmes quand on décrémente le nombre de registres que l'on a à notre disposition. Par exemple avec deux registre, on met `a` dans le registre 1, et `b et c` dans le registre 2, car `b` ne vit pas en même temps que `c`. Avec un seul registre on ne peut pas traîter le problème. On va donc faire un **SPILL**. On ba donc prendre des variables (par exemple `b`) et la mettre sur la pile. On va donc enlever progressivement les variables les unes à la suite des autres de notre problème, on va les mettre sur la pile, et cest ça qui va nous permettre d'avoir à la fin un nombre de registres suffisants pour résoudre notre problème.



Donc l'allocation des registres se passe en plusieurs étapes :

1. J'essaye de faire une coloartion.
    - **SI RÉUSSITE :** c'est gagné j'ai une allocation de mes registres, il me reste à modifier mon AST, pour produire l'assembleur qui correspond
2. Je prend une variable et fait un **SPILL**.
    - **Quand je fais un SPILL je suis en train de ré-écrire mon AST**. Je dois donc :
        - **re-calculer un graph de Float-Control**
        - reproduire du `in` et du `out`.
        - reproduire un graph d'interférence.
        - on repart depuis la coloration : on reprend les étapes jusqu'à saturation.

Progressivement, en enlevant les différentes variables, on va finir par arriver dans un cas où j'ai assez de registres pour traîter mon problème, et c'est gagné, je peux enfin produire mon code assembleur.

Module pouvant être ajouté

Le **MODULE POUVANT ETRE AJOUTÉ** est un module qui permet de minimiser le nombre de cycles dans le programme. On peut ainsi avoir un compilateur optimisant. Cela permet de faire de la ré-organisation des instructions.

On veut minimiser le nombre de cycles dans l'execution de notre programme.

- stratégie naïve : tester tous les cas possibles, et je prned celle qui minimise mon nombre de cycles.
- approches euristiques, sui nous permettent de travailler délà que au niveau de un bloc, car un bloc c'est petit, et au niveau de ce bloc là, on a des approches euristiques qui nous permettent de sélectionner les meilleures instructions, qui vont produire le cycles de gèle (quelque chose qui se passe dans le micro-processeur, au moment où l'on attend que de la donnée soit disponible)

Diverses questions :

Passage entre**HIR** et **LIR** :

LIR est un *Langage tree*. **HIR** est un *Langage tree* très succré, tellement qu'il est très proche de l'AST en entrée du *Middle-End*.