

# Compilateurs : La partie arrière

Matthieu Amiguet

2008 – 2009



---

---

---

---

---

---

---

---

1 Interprétation

2 Génération de code

3 Machines virtuelles

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

## La partie arrière

- À la fin de l'analyse sémantique, on a une représentation *complète* et *vérifiée* de la signification du programme
  - Généralement, sous forme d'un arbre syntaxique abstrait annoté, et éventuellement cousu.
- Le but va maintenant être de la transformer en un programme exécutable par la machine
- C'est le rôle de la *partie arrière* du compilateur
- Cette partie arrière est beaucoup moins standardisée que la partie avant
- Elle est aussi plus dépendante
  - du langage/de la machine cible
  - du paradigme de programmation (impérative, fonctionnelle, logique, ...).

---

---

---

---

---

---

---

---

## Interprétation

- La façon la plus "simple" d'exécuter les actions décrites dans l'arbre abstrait est de les exécuter directement
- Un programme qui procède ainsi s'appelle un *interprète* ou *interpréteur*
- Un interprète a besoin des données d'exécution, alors qu'un compilateur non
- Il existe deux sortes d'interprètes :
  - Les interprètes récursifs
  - Les interprètes itératifs.

## Interprétation récursive

5

- On attache à chaque type de noeud de l'arbre abstrait un *sous-programme d'interprétation*
  - Appelle ses enfants dans l'ordre désiré
- On appelle ensuite le sous programme de la racine et l'interprète va parcourir l'arbre récursivement
- Pour gérer le flot de contrôle de manière générale (sauts, *break*, etc.), on peut garder à jour un *indicateur d'état*.

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

## Exemple

6

```

PROCEDURE Elaborate if statement (If node):
  SET Result TO Evaluate condition (If node .condition);
  IF Status .mode /= Normal mode: RETURN;
  IF Result .type /= Boolean:
    ERROR "Condition in if-statement is not of type Boolean";
    RETURN;
  IF Result .boolean .value = True:
    Elaborate statement (If node .then part);
  ELSE Result .boolean .value = False:
    // Check if there is an else-part at all:
    IF If node .else part /= No node:
      Elaborate statement (If node .else part);
    ELSE If node .else part = No node:
      SET Status .mode TO Normal mode;

```

---

---

---

---

---

---

---

---

---

---

## Interprète récursif : avantages et inconvénients

7

- Peut être écrit relativement vite
  - Peut servir d'aide pour déboguer le langage et sa description
- Solution *lente*!
- Dans la version la plus simple, pas d'analyse sémantique statique
  - Une erreur peut survenir à tout moment...

---

---

---

---

---

---

---

---

---

---

## Interprétation itérative

8

- Plus proche de la structure d'un processeur
- Boucle principale autour d'un énoncé d'aiguillage déterminant le code pour chaque type de noeud
- Nécessite un arbre cousu
- On tient à jour un pointeur sur le noeud courant
  - similaire au pointeur d'instruction d'un processeur
  - mais gestion explicite (pas d'incrémentation automatique).

---

---

---

---

---

---

---

---

---

---

```

WHILE Active node .type /= End of program type:
  SELECT Active node .type:
  CASE ...
  CASE If type:
    // We arrive here after the condition has been evaluated;
    // the Boolean result is on the working stack.
    SET Value TO Pop working stack ();
    IF Value .boolean .value = True:
      SET Active node TO Active node .true successor;
    ELSE Value .boolean .value = False:
      IF Active node .false successor /= No node:
        SET Active node TO Active node .false successor;
      ELSE Active node .false successor = No node:
        SET Active node TO Active node .successor;
  CASE ...

```

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

### 1 Interprétation

### 2 Génération de code

- Généralités
- Machine à pile
- Machine à registres
- Contrôle du flot d'exécution

### 3 Machines virtuelles

---

---

---

---

---

---

---

---

---

---

- Facile à écrire (si l'arbre est cousu)
- Plus rapide qu'un interprète récursif...
- ... mais plus lent qu'un programme compilé.

---

---

---

---

---

---

---

---

---

---

- L'idée générale est de remplacer progressivement l'arbre syntaxique abstrait par des portions de code pour la machine cible
- On peut parfois passer par un ou plusieurs langages intermédiaires
  - généralement des sortes d'assembleur pour un machine idéalisée
- Avant et après chaque transformation, on peut ajouter une étape d'optimisation de code.

---

---

**Sélection de code** quelle partie du code doit-elle être réécrite et par quoi la remplacer ?

**Allocation de registres** Quels résultats seront-ils stockés dans les registres de la machine ?

**Ordonnement des instructions** Le code cible doit être une suite linéaire d'instructions, alors que la représentation intermédiaire ne l'est pas toujours.

- Quand on compile pour une machine multi-processeur ou un cluster apparaît aussi la question de la distribution. . .

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

- La structure d'un interprète récursif peut se transformer facilement en générateur de code
- Il suffit d'écrire dans un fichier les actions au lieu de les effectuer
  - y compris les conditionnelles, etc.
- Convient bien à la génération de code de haut niveau
- Si le code généré passe ensuite dans un bon compilateur, l'efficacité peut être raisonnable.

---

---

---

---

---

---

---

---

- On voudrait que la génération de code effectue la meilleure traduction possible du code intermédiaire selon certains critères
- Dans le cas général, la génération de code optimal est un problème NP-complet
- On peut donc essayer de réduire la taille du problème
  - Ne considérer que des petites parties du code intermédiaire à la fois
  - Simplifier la structure de la machine cible
  - Limiter l'espace de recherche à l'aide de conventions.

---

---

---

---

---

---

---

---

```
#include "parser.h" /* For types AST_node and Expression */
#include "thread.h" /* For Thread_AST() and Thread_start */
#include "stack.h" /* For Push() and Pop() */
#include "backend.h" /* For self check */

static AST_node *Active_node_pointer; /* PRIVATE */

static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
            case 'd':
                Push(expr->value);
                break;
            case 'r': {
                int e_left = Pop(); int e_right = Pop();
                switch (expr->oper) {
                    case '*': Push(e_left * e_right); break;
                    case '+': Push(e_left + e_right); break;
                }
            }
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("%d\n", Pop()); /* print the result */
}

/* PUBLIC */

void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
}
```

---

---

---

---

---

---

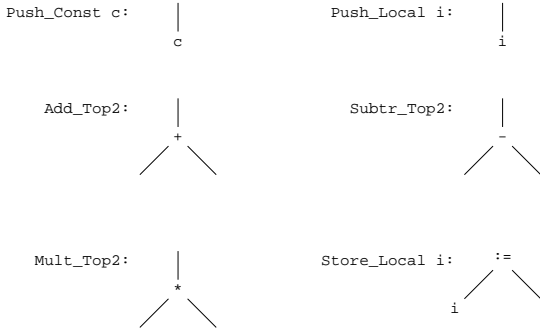
---

---



## Exemple – table de remplacement

21



---

---

---

---

---

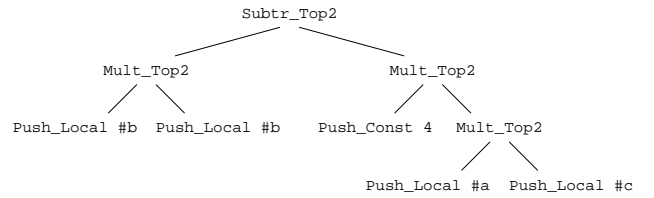
---

---

---

## Exemple – l'arbre après remplacement

22



---

---

---

---

---

---

---

---

## Exemple – le code généré

23

```
Push_Local #b  
Push_Local #b  
Mult_Top2  
Push_Const 4  
Push_Local #a  
Push_Local #c  
Mult_Top2  
Mult_Top2  
Subtr_Top2
```

## Machine à registres

24

- Dispose d'une mémoire et d'un ensemble de registres
- Deux ensembles d'instructions
  - Copie de valeurs entre mémoire et registres
  - Opérations entre valeurs des registres
- Exemple de machine à registres *pure* : IBM-360/370
- Bonne efficacité
- Principale difficulté : nombre limité de registres.

- On peut garder l'idée du parcours de l'arbre en profondeur
- On doit par contre tenir compte de l'allocation des registres
- Une manière simple de le faire est de dire que pour chaque noeud on veut le résultat dans un registre cible en utilisant un certain nombre de registres auxiliaires
- Le résultat de la racine doit être dans R1.

---

---

---

---

---

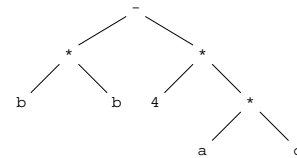
---

---

---

---

---



---

---

---

---

---

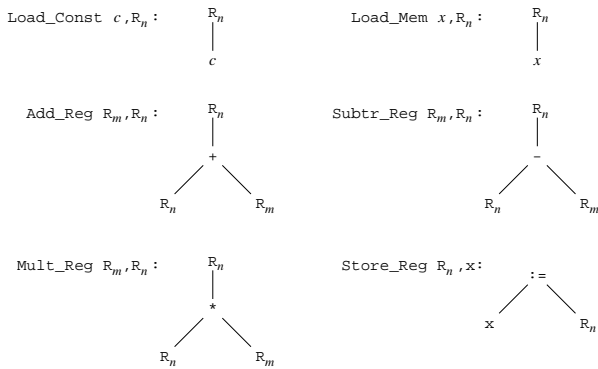
---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

```
Load_Mem b, R1
Load_Mem b, R2
Mult_Reg R2, R1
Load_Const 4, R2
Load_Mem a, R3
Load_Mem c, R4
Mult_Reg R4, R3
Mult_Reg R3, R2
Subtr_Reg R2, R1
```

- Pour un arbre compliqué, le nombre de registres de la machine sera insuffisant
- Il existe diverses techniques pour optimiser l'utilisation des registres
- Si ça ne suffit pas, il faut sauvegarder les valeurs des registres en mémoire. . .
- . . . effectuer une partie des calculs. . .
- . . . et rétablir les registres
- Il n'existe pas de technique universelle pour le faire efficacement
- L'amélioration des algorithmes d'attribution des registres fait encore l'objet d'une recherche active.

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

- Tant que l'exécution du code est linéaire, les techniques ci-dessus suffisent
- Mais on a aussi à gérer les branchements, boucles, etc.
- Le premier but est évidemment de produire du code correct. . .
- . . . mais souvent l'efficacité entre aussi en jeu
  - Minimiser le nombre d'instructions exécutées
  - Garder le *pipeline* plein
  - . . .

---

---

---

---

---

---

---

---

```
si condition alors
  code_alors
sinon
  code_sinon
finsi
```

---

---

---

---

---

---

---

---

```
code_de_condition (condition, 0, etiquette_faux)
code pour code_alors
goto etiquette_fin
etiquette_faux:
code pour code_sinon
etiquette_fin:
```

---

---

---

---

---

---

---

---

```
cas expression
  I1 : code1
  I2 : code2
  ...
  In : coden
  sinon code_sinon
fin_cas
```

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

```
tant_que condition faire
  code_boucle
fin_tant_que
```

- Traduction naïve

```
etiquette_test:
  code_de_condition (condition, 0, etiquette_fin)
  code_pour code_boucle
  goto etiquette_test
etiquette_fin:
```

---

---

---

---

---

---

---

---

---

---

```
tmp := expression
if tmp = I1 then goto etiquette_I1
...
if tmp = In then goto etiquette_In
code_pour code_sinon
goto etiquette_fin
etiquette_I1:
  code_pour code1
  goto etiquette_fin
...
etiquette_In:
  code_pour In
  etiquette_fin:
```

---

---

---

---

---

---

---

---

---

---

```
goto etiquette_test
etiquette_suite:
  code_pour code_boucle
etiquette_test:
  code_de_condition (condition, etiquette_suite, 0)
```

- Plus efficace si plusieurs itérations
  - un seul branchement conditionnel par itération (contre un branchement conditionnel et un branchement inconditionnel dans l'autre solution).

```
for (expr1; expr2; expr3) {  
    corps  
}
```

Pratiquement équivalent à

```
expr1;  
while (expr2) {  
    corps;  
    expr3;  
}
```

(sauf en cas de `continue` dans le corps de la boucle)

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

- 1 Interprétation
- 2 Génération de code
- 3 Machines virtuelles

- Les interprètes sont (relativement) faciles à écrire et facilement portables... mais lents
- Les compilateurs sont plus difficiles à écrire et plus difficilement recyclables... mais beaucoup plus rapides
- Une solution intermédiaire popularisée par Java et assez à la mode ces dernières années est de...
  - Compiler pour une machine "idéale" un code machine "virtuel"
  - Interpréter (ou compiler !) ce code "virtuel" en code "réel" au moment de l'exécution

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

- Une *machine virtuelle* (dans le contexte des compilateurs) est au fait un interprète d'un langage pré-compilé
  - Pas tout à fait la même chose qu'une machine virtuelle "système"...
- Le langage source de cet interprète est par contre ultra-optimisé pour la rapidité de compilation...
- Avantages
  - Portabilité améliorée
  - Plus rapide qu'un interprète
- Désavantages
  - Reste (souvent) plus lent qu'un programme compilé
  - Temps de démarrage

- À pile (JVM, CLR, PVM) ou à registres (Parrot, . . .)
- Avec ou sans gestion automatique de la mémoire
- Avec ou sans fonctionnalités de sécurité
- Interprétation ou compilation "JIT" (*Just In Time*)

---

---

---

---

---

---

---

---

---

---

- Certains prétendent même que la solution "machine virtuelle JIT" peut être *plus rapide que du code pré-compilé*
  - Optimisations liées au contexte d'exécution (jeu d'instruction, type de processeur, . . .)
  - Profiling et optimisation à la volée
  - . . .
- De manière générale, un programme compilé ultra-optimisé pour un contexte d'exécution donné est imbattable. . . mais en contexte courant, c'est moins clair !

---

---

---

---

---

---

---

---

---

---

- Dans sa version élémentaire, la machine virtuelle est un simple interpréteur, avec ce que cela implique au niveau de la vitesse d'exécution. . .
- Mais certaines machines *compilent* à la volée le code à exécuter en code machine ("réel")
  - On parle de compilation *just in time*, ou *JIT*
- Permet une augmentation considérable des performances. . .
- Mais peut augmenter le temps de démarrage

---

---

---

---

---

---

---

---

---

---

- Python compile à la volée vers du bytecode (.pyc), avec un système de cache du bytecode pour gagner du temps au démarrage
  - Le module `psyco` fournit une sorte de compilation JIT pour les architectures x86
- C# / .Net demande une phase de compilation explicite vers le CIL, puis fait de la compilation JIT avec un système de cache pour gagner du temps de démarrage
- La JVM *HotSpot* de Sun part en interprétation du bytecode et compile JIT les passages fréquemment exécutés
  - le programme devrait donc s'exécuter *de plus en plus vite*