Partie VII : Production de code I

- Architectures de processeurs et production de code
- Machines à pile et machines à registres
- Instructions RISC
- Code pour expressions arithmétiques
- Gestion de la mémoire
- Production de code dépendant du contexte

Production de code

- Jusqu'ici, le compilateur s'est contenté de vérifier si le programme source était légal par rapport aux règles du langage de programmation.
- On appelle cette partie *l'analyse.*
- On s'intéresse maintenant à la seconde tâche du compilateur : la traduction vers un *langage cible* directement exécutable.
- On appelle cette partie *la synthèse.*
- Il existe deux sortes de langages cibles : les langages machine et les langages intermédiaires.
- Les langages machine peuvent être directement exécutés par des composants matériels (des *machines concrètes*), tandis que les langages intermédiaires sont soit interprétés (par des *machines virtuelles*), soit compilés à nouveau.

Machines à pile

- Les langages intermédiaires sont en général basés sur une *pile.*
- Exemples d'opérations d'une machine à pile (cas de la JVM) :
  - Charger une valeur sur la pile.
    P.ex. `iload 5` charge la variable locale entière d'adresse 5 sur la pile.
  - Opérer sur les valeurs du sommet de la pile, en les remplaçant par le résultat de l'opération.
    P.ex. `iadd` remplace les deux entiers au sommet de la pile par leur somme.
  - Stocker le sommet de la pile en mémoire.
    P.ex. `istore 3` stocke le sommet de la pile dans la variable locale entière d'adresse 3.

Machines à registres

- La plupart des processeurs sont basés sur des *registres.*
- Exemples d'opérations d'un processeur à registres :
  - Charger une valeur dans un registre.
    P.ex. `LDW R1, 5[R2]` charge la valeur à la position 5 par rapport au registre R2 dans le registre R1.
  - Opérer sur les valeurs dans des registres en plaçant le résultat dans un autre registre.
    P.ex. `ADD R1, R4, R5` additionne les entiers contenus dans R4 et R5 et place le résultat dans R1.
  - Stocker le contenu d'un registre en mémoire.
    P.ex. `STW R1, 3[R2]` stocke le contenu de R1 dans la position 3 par rapport au registre R2.

Influences du matériel sur les jeux d'instructions

- Loi de Moore : le nombre de transistors par puce double environ tous les 18 mois. Loi valable au moins pour les 25 dernières années.
- La vitesse des processeurs augmente à peu près au même rythme.
- Toutefois, la vitesse d'accès à la mémoire semble doubler seulement tous les 7 ans !

	temps accès mémoire	vitesse processeur	instructions/accès
1980 :	env. 400 ns	env. 1 MIPS	0.4
1990 :	env. 150ns	env. 64 MIPS	10
2001:	env. 50ns	env. 4096 MIPS	200

- Trou de plus en plus important entre la vitesse du processeur et celle de la mémoire.
- Comment prendre en compte cela ?

1980 : CISC

- Les processeurs CISC essaient de minimiser la mémoire nécessaire au stockage des instructions en augmentant leur complexité.
- Instructions de haut niveau qui font plusieurs choses à la fois (p.ex. sauvegarde de plusieurs registres en mémoire).
- Les instructions prennent souvent leurs opérandes directement de la mémoire.
- Beaucoup de modes d'adressage sophistiqués (p.ex. indirect, indirect double, indexé, post-incrémenté, etc.)
- Réalisation au moyen de micro-code : chaque instruction est interprétée par un programme en micro-code.
- Processeurs typiques : Digital VAX, Motorola MC 68000, Intel 8086, Pentium.

1990 : RISC

- En 1990, le trou mémoire/processeur s'est creusé.
- Solutions à ce problème :
  - Éviter les accès mémoire inutiles.
    - Plus de micro-code, réalisation uniquement matérielle.
    - Grand nombre de registres, pour stocker les résultats intermédiaires et les variables des programmes.
    - Utilisation d'antémémoires (*cache memories*) pour accélérer l'accès répétitif aux données.
  - Utilisation du parallélisme au moyen d'un *pipeline.*
    - Cela fonctionne mieux avec un grand nombre d'instructions simples et régulières qu'avec un petit nombre d'instructions complexes.
- Ne pouvait être réalisé qu'avec un jeu d'instructions simple.
- D'où le concept de processeur à jeu d'instructions réduit (ou RISC, pour *Reduced Instruction Set Computer*).
- Processeurs typiques : MIPS, Sun SPARC

2000 : Formes avancées de parallélisme

- La place disponible sur une puce actuelle permet même la réalisation efficace de jeux d'instructions complexes (p.ex. Pentium)
- De manière interne, plusieurs techniques RISC sont utilisées.
- Les processeurs actuels gèrent le parallélisme à plusieurs niveaux :
  - Au niveau de l'instruction, via
    - les pipelines,
    - l'exécution « super-scalaire »,
    - les mots d'instructions très larges (*VLIW*).
  - Au niveau du processus, via
    - les architectures à fils d'exécution multiples (*multi-threaded architectures*) : changement de contexte lors d'indisponibilité de ressources (mémoire, unité de calcul, ...),
    - multi-processeurs,
    - grappes de machines (*clusters*).

Influences sur la production de code

- A l'origine, les processeurs CISC étaient conçus pour rendre le travail du compilateur simple en « fermant le trou sémantique ».
- Cela s'est avéré être un échec, étant donné qu'il était difficile d'optimiser le code avec des instructions complexes.
- Il est mieux d'avoir des instructions RISC simples, tant et aussi longtemps qu'elles sont régulières.
- Nouveau défi : détecter le parallélisme potentiel
  1. au niveau de l'instruction.
  2. au niveau des fils d'exécution (*threads*).
- Les compilateurs actuels se débrouillent assez bien dans le premier cas, mais ont encore des problèmes dans le second.

L'architecture DLX

- Nous allons produire du code pour un microprocesseur fictif.
- Il s'agit d'une version légèrement simplifiée de DLX, qui est lui-même un processeur RISC idéalisé (voir Hennessy & Patterson *Computer Architecture - A quantitative approach* Morgan Kaufmann 1990).
- 32 registres de 32 bits chacun : R0-R31
- R0 contient toujours la valeur 0.
- La mémoire est formée de mots de 32 bits, et est adressée par octets.
- Architecture de type *load/store.*
- Les types d'instructions suivants existent :
  - Instructions sur registres (opérandes et résultats : registres).
  - Instructions de chargement/stockage (*load/store*).
  - Quelques instructions spéciales pour les appels système.

Phase 1 : code pour expressions arithmétiques

Exemple : Traduction de l'expression x + y * z

On admet que x, y, z sont stockées aux adresses #x, #y et #z par rapport à un registre SP, dont on expliquera la signification plus tard.

On admet de plus que ce registre a le numéro 30.

Idée : simulation d'une pile au moyen des registres.

Code	Effet	Contenu de la « pile »
LDW 1,SP,#x	R1 := x	1: x
LDW 2,SP,#y	R2 := y	1: x, 2: y
LDW 3,SP,#z	R3 := z	1: x, 2: y, 3: z
MUL 2,2,3	R2 := R2*R3	1: x, 2: y*z
ADD 1,1,2	R1 := R1+R2	1: x+(y*z)

Schéma de génération simple pour les expressions arithmétiques

Nous gérons une variable globale RSP (*register stack pointer*) qui pointe toujours vers le registre au sommet de la pile.

Extraits de code :	code (E)
E(t) =	
Operation Add $E_1(t_1)$ $E_2(t_2)$	**code (E_1); code(E_2);** **gen(ADD RSP-1, RSP-1, RSP);** **RSP := RSP-1**
| Operation Sub $E_1(t_1)$ $E_2(t_2)$	**code (E_1); code(E_2);** **gen(SUB RSP-1, RSP-1, RSP);** **RSP := RSP-1**
| ...	
| IntLit (value)	**RSP := RSP + 1;** **gen(ADDI RSP, 0, value)**

Production des instructions

- Le compilateur produit des instructions en langage assembleur, sous forme textuelle.
- Les instructions produites sont converties au moment du chargement en format binaire, au moyen d'un assembleur qui fait partie de l'interpréteur DLX.
- Exemple de méthode utilitaire pour la production de code :

```
int emit(String op, int a, int b, int c) {
 System.out.writeln(op + " " + a + "," + b + "," + c);
}
```


Visiteurs simples de production de code

```
class Generator implements Tree.Visitor {
  /** sommet de la pile des registres */
  int RSP = 0;
  void incRSP() {
    RSP++;
    if (RSP == REG_LIMIT)
      throw new Error("expression too complex")
  }
  /** méthode du visiteur */
  public void generate (Tree tree) {
    tree.apply(this);
  }
  /** exemple de cas */
  Item caseOperation(Operation tree) {
    generate(tree.left);
    if (tree.right != null) generate(tree.right);
    switch (tree.operator) {
    case ADD: emit("ADD", RSP-1, RSP-1, RSP);
              RSP--; break;
    case SUB: ...
```

Optimisations possibles

- Examinons à nouveau l'expression x + (y * z).
- Avec le schéma simple, l'évaluation de cette expression nécessite 3 registres.
- On peut utiliser moins de registres en réorganisant l'évaluation des opérandes :

Code	Pile
LDW 1,SP,#y	1: y
LDW 2,SP,#z	1: y, 2: z
MUL 1,1,2	1: y*z
LDW 2,SP,#x	1: y*z, 2: x
ADD 1,1,2	1: x + (y*z)

- Comment peut-on produire un tel code ?
- Idée : Calculer la hauteur de l'arbre de l'expression. Pour une opération binaire A op B, produire d'abord le code pour le sous-arbre de hauteur maximale.

Dépassement de capacité des registres

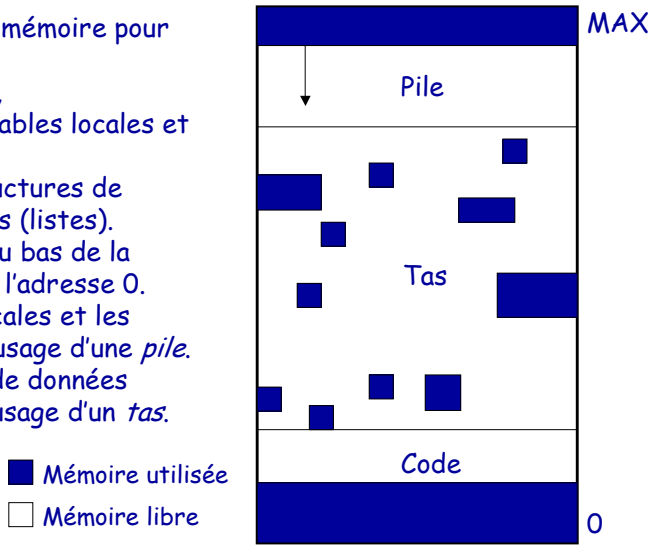
- Il est possible qu'une expression ait besoin de plus de registres qu'il n'y en a à disposition.
- Avec 31 registres, cela ne se produit que rarement, mais
  - certains processeurs ont moins de registres (p.ex. Pentium: 8),
  - certains registres sont réservés à des usages spécifiques,
  - des techniques d'optimisation plus agressives vont essayer de stocker aussi les variables et les arguments des fonctions dans des registres.
- S'il n'y a plus de registres à disposition, le compilateur doit produire du code pour sauvegarder certains registres en mémoire.
- En anglais, cela s'appelle *register spilling*.
- Dans votre projet, vous devez au moins détecter et signaler un tel dépassement de capacité, mais vous n'êtes pas obligés de le traiter.

Phase de projet

- Produire le code pour les expressions arithmétiques composées uniquement d'opérateurs et de constantes entières.
- Produire le code pour imprimer le résultat au moyen de l'instruction SYSCALL.
- Tester le code produit au moyen de l'interpréteur RISC que nous vous fournissons.

Phase 2 : organisation de la mémoire

- En misc, on utilise la mémoire pour trois raisons :
  - stockage du code,
  - stockage des variables locales et paramètres,
  - stockage des structures de données dynamiques (listes).
- Le code est stocké au bas de la mémoire, à partir de l'adresse 0.
- Pour les variables locales et les paramètres, on fait usage d'une *pile*.
- Pour les structures de données dynamiques, on fait usage d'un *tas*.



Allocation dynamique de mémoire

- La quasi-totalité des langages de programmation actuels permettent l'allocation dynamique de mémoire.
- En C, on utilise la fonction `malloc` de la bibliothèque standard.
- En C++ et en Java, on utilise l'opérateur `new`.
- En misc, l'opérateur de construction de liste (`::`) effectue l'allocation dynamique d'une cellule de liste.

Libération de la mémoire

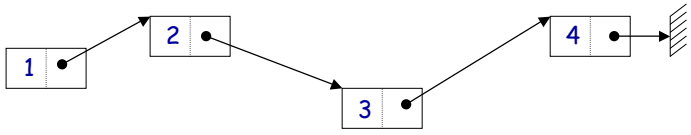
- Si on a à disposition une quantité infinie de mémoire, il n'est jamais nécessaire de libérer la mémoire allouée dynamiquement lorsqu'on n'en a plus besoin.
- En pratique, la mémoire disponible est toujours limitée. Deux solutions pour la libérer lorsqu'elle n'est plus utilisée :
  - de manière explicite (`free` en C, `delete` en C++),
  - de manière automatique, au moyen d'un *glaneur de cellules* ou *ramasse-miettes* (*garbage collector* en anglais), comme en Lisp, Java, misc, etc.
- La libération explicite de la mémoire est *très* dangereuse, et une des sources principales de bogues dans les logiciels.
- L'utilisation d'un glaneur de cellules simplifie la vie du programmeur en libérant automatiquement la mémoire qui n'est plus accessible depuis le programme.

Tas

- On appelle *tas* la zone de la mémoire dans laquelle la mémoire dynamique est allouée.
- Le tas est géré par le système de gestion de la mémoire, qui garde une liste des emplacements libres et alloués dans le tas.
- Le système de gestion de la mémoire offre des services d'allocation et de libération de mémoire dynamique. Comme dit précédemment, la libération peut être automatique.

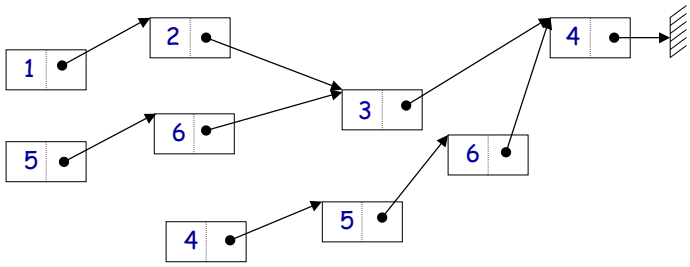
Listes en misc

- En misc, les seules structures de données allouées dynamiquement sont les listes.
- Elles sont représentées au moyen de paires (ou cellules), comme en Lisp, Scala, etc. :
  - le premier élément de la paire contient la tête de la liste,
  - le second élément de la paire contient queue de la liste (qui peut être la liste vide `[]`).
- Les paires sont toujours manipulées via des pointeurs, jamais directement. La liste vide `[]` est le pointeur nul (0).
- Par exemple, la liste `[1,2,3,4]` peut être représentée en mémoire comme :



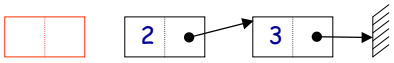
Listes en misc (2)

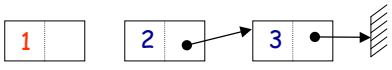
- Les listes misc sont constantes : pas moyen de modifier le contenu d'une cellule une fois qu'elle est créée.
- Cela autorise le *partage* des sous-listes. P.ex. les listes `[1,2,3,4]`, `[5,6,3,4]` et `[4,5,6,4]` pourraient être représentées ainsi :

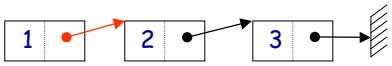


Listes en misc (3)

- La construction de listes se fait au moyen de l'opérateur `::` (prononcé « cons », en référence à l'opérateur Lisp original).
- Cet opérateur alloue et initialise une nouvelle cellule.
- Par exemple, si on a une variable nommée `x` contenant la liste `[2,3]`, l'expression `1::x` effectue les opérations suivantes :

1. allocation d'une cellule 

2. initialisation de la tête 

3. initialisation de la queue 

Phase 3 : Production de code dépendant du contexte

- Question : quel code produire pour le test `x==0` dans l'expression suivante ? `if (x == 0) ...`

```
        LDW     1,SP,#x         // chargement de x
        BNE     1,ELSE          // saut si différent de 0
        ...
```

- Et dans cette expression ? `var b: Int = (x == 0);`

```
        LDW     1,SP,#x         // chargement de x
        BNE     1,FLS           // saut si différent de 0
        ADDI    1,0,1           // chargement de TRUE (1)
        BEQ     0,AFTER         // saut à la suite
FLS:    ADDI    1,0,0           // chargement de FALSE (0)
AFTER:  ...
```

- Pour une même expression (`x==0`) on produit donc du code différent en fonction de son contexte d'utilisation.
- Une partie de la production de code doit donc être différée.

Items (1)

- Un item est une structure de données représentant du code partiellement produit.
- Les items offrent des méthodes qui complètent la production de code de différentes manières.
- Exemple de réalisation :

```
abstract class Item {
  // Complète le code en chargeant la valeur représentée par cet item
  // dans un registre. Retourne un item représentant ce registre.
  abstract RegisterItem load(Code code);
}
```

Items (2)

- Les sous-classes concrètes de cette classe modélisent les différentes possibilités de production de code.
- En voici une simple :

```
/** Classe pour les valeurs qui sont déjà dans un registre. */
class RegisterItem extends Item {
  int register;

  RegisterItem(int register) {
    this.register = register;
  }

  RegisterItem load(Code code) {
    return this;
  }
}
```


Items (3)

- En voici une autre :

```
/** Classe pour les valeurs constantes. */
class ImmediateItem extends Item {
  int value;

  ImmediateItem(int value) {
    this.value = value;
  }

  RegisterItem load(Code code) {
    RegisterItem ritem = code.getRegister();
    code.emit(ADDI, ritem.register, 0, value);
    return ritem;
  }
}
```

Items (4)

- Et en voici une troisième :

```
/** Classe pour les valeurs qui sont sur la pile. */
class StackItem extends Item {
   int offset;
   StackItem(int offset) {
     this.offset = offset;
   }
   RegisterItem load(Code code) {
     RegisterItem ritem = code.getRegister();
     code.emit(LDW, ritem.register, SP, stackSize - offset);
     return ritem;
   }
}
```


Utilisation des items dans le compilateur

- Toutes les méthodes de visiteur qui gèrent des expressions retournent désormais un item comme résultat.
- L'item retourné par le visiteur d'une expression décrit comment compléter la production de code pour cette expression.
- C-à-d que la classe de production de code devrait débuter ainsi :

```
class Generator implements Tree.Visitor {
  Item item; // sera assigné dans chaque cas
  /** méthode principale du visiteur */
  public Item generate(Tree tree) {
    Item old = item;     // Sauvegarde la val. actuelle
    item = null;         // Initialise à null
    tree.apply(this);    // Produit le code pour l'arbre
    Item res = item;     // Sauvegarde le résultat
    item = old;          // Restaure la valeur courante.
    return res;
  }
  ...
}
```

Exemples de cas

```
public void caseIntLit(IntLit tree) {
    item = new ImmediateItem(tree.value);
}

public void caseOperation(Operation tree) {
    switch (tree.op) {
       case ADD: {
         RegisterItem left = generate(tree.left).load(code);
         RegisterItem right = generate(tree.right).load(code);
         code.freeRegister(right);
         code.freeRegister(left);
         RegisterItem ritem = code.getRegister();
         code.emit(ADD, ritem.register, left.register, right.register);
         item = ritem;
         break;
       }
        ...
    }
}
```


Exemple d'exécution