# Production de code, expressions arithmétiques

Martin Odersky, Gilles Dubochet

9 janvier 2006
version 1.0

# Plan du cours

1. Langages cibles
   - Différents types de langages cibles
   - Influence du matériel sur les langages
2. Génération de code cible
   - Génération d'expressions arithmétiques
   - Dépasser le nombre limite de registres

# Analyse et synthèse

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.

# Langages cibles

Il existe deux sortes de langages cibles :

1. les langages machine
   qui peuvent être directement exécutés par des composants matériels (des machines concrètes) ;
2. les langages intermédiaires
   qui sont soit interprétés (par des machines virtuelles), soit compilés à nouveau.

Les langages intermédiaires sont en général basés sur une pile.

## Exemple : Quelques opérations de la machine virtuelle Java

Charger une valeur sur la pile :

- `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 :

- `iadd` remplace les deux entiers au sommet de la pile par leur somme.

Stocker le sommet de la pile en mémoire :

- `istore 3` stocke le sommet de la pile dans la variable locale entière d'adresse 3.

La plupart des processeurs — et par conséquent des langages machine — sont basés sur des registres.

## Exemple : Quelques opérations d'un processeur à registres

Charger une valeur dans un registre :

- `LDW R1 R2 8` charge la valeur à la position 8 par rapport au registre `R2` dans le registre `R1`.

Opérer sur les valeurs dans des registres en plaçant le résultat dans un autre registre :

- `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 :

- `STW R1 R2 4` stocke le contenu de `R1` dans la position 4 par rapport au registre `R2`.

# Influence du matériel sur les jeux d'instructions

**Définition : «Loi» de Moore**

Le nombre de transistors par circuit intégré double tous les 18–24 mois.

La vitesse des processeurs augmente à peu près au même rythme alors que la vitesse d'accès à la mémoire semble doubler seulement tous les 7 ans !

| | accès mémoire | vitesse processeur | instructions/accès |
|---|---|---|---|
| 1980 | 400ns | 1 MIPS | 0.25 |
| 1990 | 150ns | 50 MIPS | 7.5 |
| 2000 | 50ns | 4'000 MIPS | 200 |
| 2005 | 15ns | 20'000 MIPS | 300 |

Il existe un fossé de plus en plus important entre la vitesse du processeur et celle de la mémoire.

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.

Quelques processeurs CISC typiques : Digital VAX, Motorola MC 68000, Intel 8086, Pentium.

En 1990, le trou mémoire/processeur s'est creusé. Il faut :

- 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.

Ces solutions demandant un jeu d'instructions simple, on imagine le processeur à jeu d'instructions réduit (RISC pour *Reduced Instruction Set Computer*).

Quelques processeurs RISC typiques : MIPS, Sun SPARC, IBM Power.

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 qui changent de contexte lors d'indisponibilité de ressources (mémoire, unité de calcul, ...),
  - multi-processeurs,
  - grappes de machines (clusters).

# Les défis du parallélisme

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 : une version légèrement simplifiée du processeur DLX, un processeur RISC idéalisé.

- 32 registres de 32 bits chacun : `R0`–`R31`.
- `R0` contient toujours la valeur 0.
- La mémoire est formée de mots de 32 bits adressés 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.

Référence sur le processeur DLX : *D. Patterson* et *J. Hennessy* : «Computer Architecture : a Quantitative Approach» 1990, *Morgan Kaufmann*

Considérons l'expression arithmétique suivante : $x + y * z$. On cherche à la traduire en assembleur DLX.

- 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.

## Exemple : Expressions arithmétiques en assembleur

| Code assembleur | Effet | Contenu de la pile |
|---|---|---|
| `LDW 1 SP #x` | `R1 := x` | $x$ |
| `LDW 2 SP #y` | `R2 := y` | $x$ $y$ |
| `LDW 3 SP #z` | `R3 := z` | $x$ $y$ $z$ |
| `MUL 2 2 3` | `R2 := R2*R3` | $x$ $y * z$ |
| `ADD 1 1 2` | `R1 := R1+R2` | $x + (y * z)$ |

# Schéma de génération avec pile de registre

L'idée est de gérer une variable globale `RSP` (*register stack pointer*) qui pointe toujours vers le registre au sommet de la pile.

| Expression $E$ | $Code(E)$ |
|---|---|
| `E = BinOp Add` $E_1$ $E_2$ | `code(`$E_1$`); code(`$E_2$`);`<br>`gen(ADD, RSP-1, RSP-1, RSP);`<br>`RSP = RSP-1;` |
| `BinOp Sub` $E_1$ $E_2$ | `code(`$E_1$`); code(`$E_2$`);`<br>`gen(SUB, RSP-1, RSP-1, RSP);`<br>`RSP = RSP-1;` |
| `...` | |
| `IntLit value` | `RSP = RSP+1;`<br>`gen(ADDI, RSP, 0, value);` |

# Les instructions DLX

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.

Dans le compilateur, on utilise une méthode utilitaire pour la production de code. Par exemple :

```
def emit(op: String, a: Int, b: Int, c: Int): Unit =
  AssemblerFile.println(op+"␣"+a+"␣"+b+""+c);
```

# De l'AST à l'assembleur

```
class Generator(analyzer: Analyzer) {
  val code = new Code();
  def gen(tree: Tree): Unit = {
    ... genLoad(tree, resReg) ...
  }
  def genLoad(tree: Tree, targetReg: Int): Unit = tree match {
    case Binop(op, left, right) =>
      genLoad(left, targetReg);
      val rightReg = code.getRegister();
      genLoad(right, rightReg);
      code.emit(opcode(op), targetReg, targetReg, rightReg);
      code.freeRegister(rightReg);
    ...
  }
  ...
}
```

```
class Code {
  private val RC_MIN =  1;
  private val RC_MAX = 29;
  private val freeRegisters = new Array[Boolean](32);
  def getRegister(): Int =
    Iterator.range(RC_MIN, RC_MAX+1) find
      { r => freeRegisters(r) } match {
        case Some(r) => getRegister(r)
        case None    => throw new Error("no␣more␣free␣registers")
      }
  ...
}
```

# Utilisation des registres

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.

## Exemple : Expressions arithmétiques en assembleur optimisé

| Code assembleur | Contenu de la pile |
|---|---|
| `LDW 2 SP#y` | $y$ |
| `LDW 3 SP#z` | $y$ $z$ |
| `MUL 2 2 3` | $y * z$ |
| `LDW 1 SP#x` | $y * z$ $x$ |
| `ADD 1 1 2` | $x + (y * z)$ |

Pour une opération binaire `A op B`, on produit d'abord le code pour le sous-arbre de hauteur maximale.

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 (toute la famille x86, dont les Pentium, se contente de 8 registres),
- 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 (*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.