Partie IX : Production de code III

- Gestion des fonctions :
  - Variables locales et paramètres
  - Gestion de la pile
  - Blocs d'activation
  - Appels de fonction
  - Sauvegarde des registres
- Optimisations simples

Variables locales et paramètres

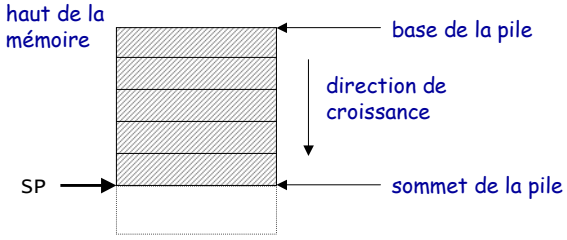
- Où stocker les variables locales et les paramètres de fonctions ?
- Première idée : à des adresses fixes en mémoire (allocation statique, à la Fortran).
- Malheureusement, cela interdit certaines formes de récursivité, car chaque fonction ne possède qu'une copie de ses variables locales et paramètres.
- Meilleur idée : allouer la mémoire pour les variables et paramètres dynamiquement, au moment de l'exécution.
- Solution utilisée par la quasi-totalité des langages actuels.

Variables locales et paramètres (2)

- Quelle stratégie d'allocation utiliser pour ces variables et paramètres ?
- Constatation : à l'exécution, les appels de fonction forment une pile :
  - la fonction en cours d'exécution se trouve toujours au sommet de la pile,
  - lorsqu'une fonction en appelle une autre, cette dernière est placée au sommet de la pile, et s'exécute *en totalité* avant de retourner à la fonction qui l'a appelée (c-à-d d'être dépilée).
- Idée pour l'allocation mémoire des variables locales et paramètres : utiliser une pile.

La pile

- Zone mémoire dans laquelle sont allouées toutes les données dont la durée de vie est incluse dans celle de la fonction qui les définit.
- Sommet de la pile repéré par le *pointeur de pile* (*stack pointer* ou SP en anglais), habituellement stocké dans un registre réservé à cet effet.
- Attention : en mémoire, la pile croît souvent vers le bas, donc son sommet se trouve à une adresse inférieure à sa base.



Gestion de la pile sur DLX

- L'architecture DLX possède deux instructions pour faciliter la gestion de la pile :
  - `PSH Ra,Rb,ic` `Rb = Rb - ic; MEM[Rb] = Ra;`
  - `POP Ra,Rb,ic` `Ra = MEM[Rb]; Rb = Rb + ic;`
- `PSH` alloue de la place sur la pile puis y copie une valeur, `POP` lit une valeur de la pile, puis libère de la place.
- Le registre `Rb` est le pointeur de pile. Par convention, on utilise `R30`, mais n'importe quel autre registre ferait l'affaire.
- Exemple d'utilisation : appel de fonction

`f(15,16)` 

```
ADDI    1,0,15
PSH     1,SP,4
ADDI    1,0,16
PSH     1,SP,4
JSR     f
```

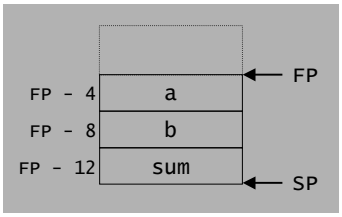

Bloc d'activation

- Chaque fonction ne manipule directement qu'une petite zone au sommet de la pile, qui lui est réservée.
- Cette zone se nomme *bloc d'activation* (*stack frame* en anglais).
- Lorsqu'une fonction commence son exécution, son bloc d'activation ne contient que les paramètres qui lui ont été passés par la fonction appelante.
- Au fur et à mesure de son exécution, la fonction peut faire grandir son bloc d'activation pour y stocker différentes données : variables locales, copies de registres à sauvegarder, paramètres pour fonctions appelées, etc.

Pointeur de bloc d'activation

- Chaque fonction conserve un pointeur vers la base de son bloc d'activation, habituellement dans un registre réservé à cet effet et nommé FP (pour *frame pointer*).
- Ce pointeur permet d'accéder aux paramètres et aux variables locales, qui se trouvent à une *distance fixe* par rapport à FP.
- Exemple :

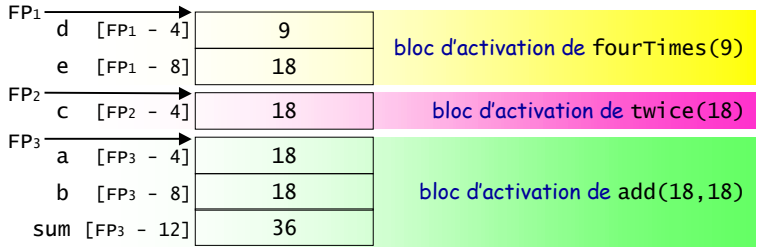
```
def add(a: Int, b: Int): Int = {
  var sum: Int = a+b;
  sum
};

...
LDW  1,FP,-4   // charge a
LDW  2,FP,-8   // charge b
ADD  1,1,2     // calcule a+b
PSH  1,SP,4    // stocke dans sum
LDW  1,FP,-12  // charge sum
...
```



Pointeur de bloc d'activation (2)

```
def add(a: Int, b: Int): Int = { var sum: Int = a+b;★sum };
def twice(c: Int): Int = add(c,c);
def fourTimes(d: Int): Int = { var e: Int = d+d; twice(e) };
fourTimes(9)
```

état de la pile à l'endroit ★



Peut-on se passer de FP ?

- Utiliser un registre pour stocker FP a plusieurs inconvénients :
  - un registre de moins disponible pour utilisation générale,
  - il faut sauvegarder/restaurer FP lors d'appels de fonction.
- Question : est-il possible de faire sans ?
- Invariant : `frame_size = FP - SP`
- En d'autres termes : `FP = SP + frame_size`
- Conclusion : si `frame_size` est connu à tout moment lors de la compilation, on peut se passer de FP !
- C'est ce qu'on fait dans le projet : la classe `Code` fournit les méthodes `incFrameSize`, `decFrameSize` et `getFrameSize` pour mémoriser la taille courante du bloc d'activation.
- A utiliser sans faute chaque fois qu'on produit des instructions qui modifient la taille du bloc d'activation (p.ex. PSH et POP).

Passage de paramètres par registres

- Les paramètres d'une fonction peuvent aussi être passés dans les registres plutôt que sur la pile.
- Quelle solution est la meilleure ?
- Avantages du passage par registres :
  - code potentiellement plus compact, car l'appelant n'a pas besoin de placer les paramètres sur la pile (important pour les processeurs sans instruction PSH),
  - code potentiellement plus rapide, car les procédures feuille (qui ne contiennent pas d'appels) n'ont pas besoin de sauvegarder les paramètres,
  - aucune correction à apporter pour accéder aux données stockées sur la pile.

Passage de paramètres par registres (2)

- Avantages du passage sur la pile :
  - nécessite moins de registres, ce qui peut être important lorsqu'il y en a peu, comme sur le Pentium,
  - lors d'appels imbriqués, les paramètres ne doivent pas être sauvegardés et restaurés de manière répétée; par exemple, si l'on passe les arguments dans les registres, lors de l'appel
    ```
    f(x, g(y, h(z)));
    ```
    le paramètre `x` est chargé en premier, puis doit être sauvegardé durant l'appel à `g`; de même `y` doit être chargé puis sauvegardé lors de l'appel à `h`; cela n'est pas nécessaire si l'on passe les arguments sur la pile.
- Résumé : le passage par registres est meilleur lorsqu'on a suffisamment de registres à disposition.

Valeur de retour

- Toute fonction doit communiquer sa valeur de retour à l'appelant.
- Pour les résultats de taille inférieure ou égale à un mot, on peut utiliser un registre.
- En misc, *toutes* les valeurs ont une taille d'un mot, on retourne donc toujours le résultat dans un registre, `R1` par convention (appelé aussi `RES`).
- Pour les résultats plus grands, il y a deux possibilités :
  - sur la pile, ce qui complique la séquence de sortie,
  - l'appelant alloue la place dans sa portion de pile et passe un pointeur vers cet espace à l'appelé, qui y stocke son résultat.

Listes comme paramètres et résultats de fonctions

- Lorsqu'on passe des listes en paramètre, on passe en fait un pointeur vers la première cellule de la liste.
- Lorsqu'on retourne une liste comme résultat, on retourne également un pointeur.
- En misc, une fonction peut retourner une liste qu'elle a créée localement, au moyen de l'opérateur « cons » (`::`). Exemple :

```
def prependTwice(elem: Int, lst: List[Int]): List[Int] =
  elem :: elem :: lst;
```

- Il n'est donc pas possible d'allouer les listes sur la pile, car elles ne survivraient pas à la fonction qui les a créées (ou alors il faudrait les copier au retour, ce qui est coûteux).
- Cette constatation explique l'usage d'un tas et d'un ramasse-miettes en misc.

Adresse de retour

- Sur certains processeurs, la pile est aussi utilisée pour mémoriser l'adresse de retour :
  - au moment de l'appel d'une fonction, l'adresse de retour est placée sur la pile,
  - au moment du retour, elle est simplement dépilée.
- Le processeur DLX fonctionne différemment :
  - au moment d'un appel, l'instruction `JSR` place l'adresse de retour dans le registre `R31`, appelé aussi `LNK` (*link*),
  - au moment du retour, l'instruction `RET` prend en argument le registre contenant l'adresse de retour (`LNK`).
- Les fonctions qui appellent d'autres fonctions doivent donc sauvegarder sur la pile le registre `LNK` au début de leur exécution et le restaurer juste avant l'instruction `RET`.

Appel de fonctions « inconnues »

- Dans un appel de fonction misc, la fonction peut être une expression quelconque, p.ex. `(if (x>2) f else g)(5)`
- Cela rend l'utilisation de l'instruction `JSR` impossible, car elle admet que l'adresse de la fonction appelée est connue au moment de la compilation...
- Il faut donc émuler le comportement de l'instruction `JSR` :
  - placer l'adresse de retour dans le registre `LNK`,
  - sauter à l'adresse de la fonction (qui est connue seulement moment de l'exécution).
- L'adresse de retour est facile à calculer grâce à la fonction `pc()` de la classe `Code`.
- Le saut se fait au moyen de l'instruction ... `RET` !

Sauvegarde des registres

- Lorsqu'une fonction en appelle une autre, il est probable que la fonction appelée modifie le contenu de registres dont la fonction appelante aura besoin par la suite.
- Il faut donc sauvegarder le contenu des registres au moment de l'appel de fonction.
- La sauvegarde s'effectue sur la pile.
- Deux stratégies :
  - sauvegarde par la fonction appelante,
  - sauvegarde par la fonction appelée.

Sauvegarde par l'appelant ou l'appelé ?

- La sauvegarde et restauration des registres lors d'appels peut être faite par l'appelant (*caller-save* en anglais) ou par l'appelé (*callee-save* en anglais).
- Sauvegarde par l'appelant :
  - avant un appel, sauvegarder tous les registres dont le contenu sera nécessaire ensuite (SP excepté),
  - après un appel, les restaurer.
- Sauvegarde par l'appelé :
  - au début d'une fonction, sauvegarder tous les registres qui seront modifiés dans le corps (RES et SP exceptés),
  - à la fin d'une fonction, les restaurer.

Sauvegarde par l'appelant ou l'appelé ? (2)

- Si les variables locales sont stockées dans des registres, la sauvegarde par l'appelant risque de sauvegarder et restaurer beaucoup de registres à chaque appel.
- Amélioration : on réserve un certain nombre de registres pour les variables locales, registres qui sont sauvegardés par l'appelé.
- Donc, chaque fonction qui désire utiliser des registres pour y stocker ses variables locales doit sauvegarder et restaurer leur valeur.
- La plupart des compilateurs utilisent la technique de la sauvegarde par l'appelant, ou une combinaison de sauvegarde par l'appelant et l'appelé.
- Projet : sauvegarde par l'appelant, sauf pour le registre LNK qui est sauvegardé par l'appelé.

Résumé : gestion des fonctions en misc

- Au début d'une fonction (prologue) :
  - sauvegarder LNK sur la pile (optionnel pour les fonctions feuilles).
- A la fin d'une fonction (épilogue) :
  - restaurer LNK,
  - libérer l'espace utilisé par les arguments (bloc d'activation).
- Avant un appel de fonction :
  - sauvegarder les registres dont le contenu sera nécessaire après l'appel (sauf LNK).
- Après un appel de fonction :
  - restaurer les registres précédemment sauvegardés.

Exemple : factorielle

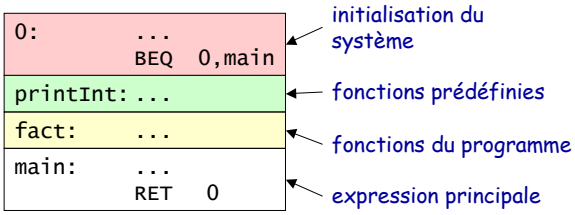
Examinons le code produit pour la fonction factorielle en misc

```
def fact(x: Int): Int =
  if (x == 0)
    1
  else {
    var y: Int = fact(x - 1);
    x*y
  };
```

```
fact: PSH  LNK,SP,4  sauvegarde LNK
      LDW  2,SP,4    charge x
      BNE  2,else
      ORIU 1,0,1     si x nul
      BEQ  0,end     retourne 1
else: LDW  1,SP,4    sinon, charge x
      ORIU 2,0,1     charge 1
      SUB  1,1,2     calcule x-1
      PSH  1,SP,4    empile
      JSR  fact      appel récursif
      PSH  1,SP,4    stocke dans y
      LDW  1,SP,8    charge x
      LDW  2,SP,0    charge y
      MUL  1,1,2     multiplie
      ADDI SP,SP,4   libère y
end:  POP  LNK,SP,8  restaure LNK
      RET  LNK       retourne
```

Exemple de programme complet

- Un programme misc complet est composé de définitions de fonctions suivies d'une expression principale. Parmi les définitions de fonctions figurent implicitement les fonctions prédéfinies `readInt`, `readChar`, `printInt` et `printChar`.
- Le code pour les fonctions est produit au fur et à mesure que les définitions sont examinées. Il faut donc insérer un saut depuis le début du programme vers l'expression principale.
- De plus, au début d'un programme il faut initialiser le glaneur de cellules et le pointeur de pile.
- Structure générale :



Optimisations simples

- Voici quelques exemples d'optimisations simples qui pourraient facilement être ajoutées au compilateur misc.
- Le point commun de toutes ces optimisation est qu'elles peuvent s'effectuer directement sur l'arbre de syntaxe abstraite.
- Certaines optimisations plus avancées requièrent en général une autre représentation du code intermédiaire.

Réduction d'expressions constantes

- Soit la déclaration suivante :

```
var x: Int = 4 * 17
```

- On attend d'un bon compilateur qu'il produise le code suivant :

```
ADDI  1,0,68
PSH   1,SP,4
```

- Le calcul de l'expression constante doit donc se faire à la compilation.
- On peut facilement ajouter cela dans une phase préalable de transformation de l'arbre, voir le faire directement à la construction de l'arbre ou durant l'analyse de types.

Réduction de force des opérateurs

- Quelques opérateurs « coûteux » peuvent être remplacés par d'autres opérateurs plus simples si une des opérandes est une constante particulière. P.ex.

  $x * 2^n$ devient $x << n$

- Si la spécification du langage définit le bon modèle d'arrondi (arrondi vers le bas pour les nombres négatifs et positifs), la division entière et le reste peuvent également se transformer :

  $x / 2^n$ devient $x >> n$

  $x \% 2^n$ devient $x \& (2^n-1)$

  Cette transformation est toujours correcte pour des x positifs, mais pour des x négatifs il faut que la division arrondisse vers le bas, et pas vers zéro. C-à-d que -3/10 doit valoir -1, pas 0.

  Malheureusement C, Java et la plupart des processeurs arrondissent vers 0.

Élimination de sous-expressions communes

- Soit le morceau de programme

```
x = (a + b) / c ; y = (a + b) / d;
```

- Le compilateur pourrait décider d'évaluer `(a + b)` une seule fois, en transformant le programme ainsi :

```
var temp: Int = a + b; x = temp / c; y = temp / d
```

- On appelle cette optimisation *élimination de sous-expressions communes* (*common sub-expressions elimination* ou *CSE* en anglais).
- Pourquoi ne pas demander au programmeur d'écrire lui-même la seconde version ?
  - le premier programme peut être jugé plus clair,
  - parfois les sous-expressions communes apparaissent dans du code que le programmeur n'a pas écrit, par exemple lors d'accès aux tableaux; ainsi, en Java, `a[i] = b[i]` contient implicitement deux sous-expressions de la forme `R = i * 4` pour le calcul de l'adresse des éléments.

- Cela dit, l'élimination de sous-expressions communes ne constitue pas toujours une amélioration.
  - Le stockage et le chargement des variables temporaires introduites ont un coût.
  - Ce coût est toutefois négligeable si les variables temporaires ne sont jamais stockées en mémoire.
  - Si l'élimination de sous-expressions communes introduit trop de variables temporaires stockées dans des registres, il est possible qu'il ne reste que peu de registres pour stocker d'autres données ; on dit alors que la pression sur les registres augmente.
  - Solution à ce dernier problème : on n'effectue l'élimination de sous-expressions communes que si on a assez de registres à disposition.

Détection de sous-expressions communes

- Soit l'extrait de programme Java suivant :

```
x = (a + b) / c;
a = a + 1;
y = (a + b) / d;
```

- Ici, (a + b) n'est pas une sous-expression commune car a est modifiée dans le second énoncé. Ce problème se pose dans tous les langages qui admettent les modifications de variables.
- Comment détecter alors les sous-expressions communes ?
- Solution : la numérotation des versions (*version numbering*).
  - lors de la production de code, on donne à chaque variable locale un numéro de version, que l'on stocke p.ex. dans le symbole,
  - chaque affectation à la variable incrémente sa version,
  - on recherche ensuite les sous-expressions communes dans lesquelles les numéros de version des variables utilisées concordent.

- Par exemple, notre extrait de programme après numérotation des versions ressemble à :

```
x1 = (a1 + b1) / c1;
a2 = a1 + 1;
y1 = (a2 + b1) / d1;
```

- La fausse sous-expression commune (a+b) a désormais disparu.
- Pour trouver les expressions qui ont déjà été évaluées, on a en général recours à une table de hachage associant des variables temporaires à des arbres :

```
HashTable<Tree, Symbol> evaluatedExpressions;
```

Optimisations avancées

- Les optimisations précédentes ne sont que le début. Les compilateurs réels en effectuent bien d'autres, et on en découvre constamment de nouvelles.
- Les principales sources d'optimisations sont :
  1. Éviter d'être trop « stupide » :
     - supprimer le code mort,
     - supprimer les sauts vers d'autres sauts,
     - supprimer les chargements de variable suivis d'un stockage de la même variable.
  2. Améliorer l'utilisation de la mémoire :
     - stocker les variables dans les registres,
     - améliorer la « localité » des accès aux variables, de sorte à ce qu'elles soient trouvées autant que possible dans l'antémémoire et pas dans la mémoire principale (ou, pour les gros programmes, dans la mémoire principale et pas sur disque).

  3. Augmenter le parallélisme :
     - réordonner les instructions pour que plus d'instructions puissent être exécutées en parallèle et pour éviter les « bulles » dans le pipeline,
     - récrire les programmes pour diminuer le nombre de sauts, p.ex. en déroulant les boucles,
     - récrire les programmes pour utiliser les fils d'exécution multiples (*threads*).
  4. Éviter les calculs répétés :
     - déplacer les instructions d'un endroit où elles sont exécutées souvent vers un endroit où elles sont exécutées moins souvent. Par exemple, déplacer les instructions en dehors d'une boucle si leur valeur est invariante par rapport à la boucle.