Projet 2: après l'analyse syntaxique

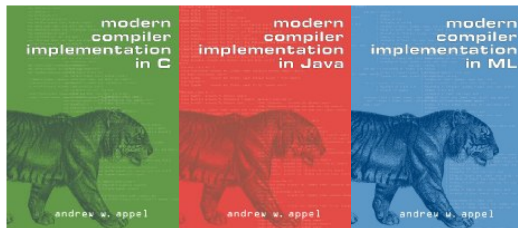
cours: Daniel Hirschkoff
TPs: Paul Renaud-Goud

Au sujet de Pascal-

```
procedure f(x:integer, var y:integer)
```

- procedure, ca veut dire fonction
- `y` est passé *par référence*
  (`f` a le pouvoir de modifier la valeur de `y`: pas le sien, celui de l'appelant)

Livres d'A. Appel



- disponibles à la bibliothèque
- vous pouvez y cueillir de l'inspiration; ne vous y perdez pas!
- le langage qui y est compilé, Tiger, partage d'importants aspects avec le sous-ensemble de Pascal qui nous préoccupe

Organisation – rendus

- restent trois rendus:
  1. 19/4 tout, sauf les fonctions (!?! IMP, quoi)
     + lambda-lifting (sortir les fonctions)
  2. 3/5 tout
  3. 23/5 tout++ (à préciser)
- en deux déclinaisons (pour ceux qui compilent Pascal)
  1. groupe 1 *tout dans la pile*

     R. Crubille R. Nolllet G. Marti R. Tetley O. Manoussakis M. Schmitt X. Dumont
  2. groupe 2 *vivacité / registres*

     A. Mottet L. Parlant B. Simon A. Durier G. Gilbert M. Savaro F. Dross M. Rosenfeld

     H. Derycke T. Pecatte
- ce qui suit ne concerne **pas** que le groupe 2

Lambda-lifting

vous savez déjà tous ce que c'est

Pile, registres

- on a besoin de la pile: pourquoi?
  - pour “faire vivre” les appels de fonctions
  - pour stocker des résultats intermédiaires de calculs

  groupe 1: on s’en tient à cela
- accéder à la pile (en mémoire) est considérablement plus coûteux qu’accéder aux registres
  - SPIM propose 32 registres
    - certains sont “réservés” (usage canonique)
    - d’autres sont à disposition
  - l’idée:

    ```
    function f (x,y : integer) : integer;
    begin
    f := x+y*x;
    end;
    ```

    devrait être compilé vers un code qui n’accède pas à la pile pour calculer `x+y*x`

Code intermédiaire

pour déterminer si on peut travailler uniquement sur les registres:

- code intermédiaire
  - on traduit de façon systématique (voire bête) le programme Pascal dans un langage aux constructions très simples
  - à mi-chemin entre IMP et SPIM
  - en particulier, on engendre généreusement des temporaires (registres virtuels)

    typiquement,

    ```
        t1 := 0;
    L1  t2 := t1+1;
        t3 := t3+t2;
        t1 := t2*2;
        if t1<64 goto L1;
        res := t3;
    ```
  - il y a donc un nouveau langage
- allocation de registres

  seconde étape: on associe à chaque temporaire

  $\left\{ \begin{array}{l} \text{soit un registre physique,} \\ \text{soit l'information ''à stocker sur la pile''} \end{array} \right.$

Vivacité (chapitre 10 A. Appel)

- soit le code

```
1       t1 := 0;
2  L1   t2 := t1+1;
3       t3 := t3+t2;
4       t1 := t2*2;
5       if t1<64 goto L1;
6       res := t3;
```

  représenté par son *graphe de flot de contrôle* au tableau
  sur lequel on peut représenter la *durée de vie* des variables au tableau

- calcul de la vivacité: point fixe pour les équations
  - in[n] = use[n] ∪ (out[n] \ def[n])   n nœud du graphe
  - out[n] = $\bigcup_{s \in succ(n)}$ in[s]
  - si on regarde n: `c := u+v`
    use[n] = $\{u, v\}$ def[n] = $\{c\}$ in[n] = $\{u, v\}$
- exemple: au tableau
  - l'information de vivacité est un saumon

L'allocation de registres (chapitre 11 A. Appel)

- on suppose qu'on dispose de K registres pour faire des calculs
  (important! ne pas fixer K en dur à plein d'endroits du compilateur!)
- l'information de vivacité permet de représenter les interférences entre registres virtuels
  1. le code intermédiaire contiendra une instruction `move`, qui se contente de recopier la valeur d'un temporaire vers un autre
  2. pour toute instruction n qui n'est pas un `move` et qui définit a: si b1,..,bk sont les variables "live-out" en n, interférences (a,bi)
  3. pour une instruction n: `a:= c`, où les b1,..,bk sont les variables "live-out", n'ajouter (a,bi) *que si bi≠c*
     - 3.1 puisque `a` et `c` contiennent la même valeur après n, pas d'interférence
     - 3.2 une affectation de `a` plus tard créera une interférence avec `c` si `c` est encore vivante à ce moment là
- pour réaliser l'allocation de registres, on cherche à construire une K-coloration du graphe, K étant le nombre de registres disponibles
  - problème NP complet, heuristiques (mais demandez à A. Darte, LIP)
  - si pas possible, certains temporaires sont stockés sur la pile (et l'enregistrement d'activation croît en conséquence) *"spilling"*

Engendrer du code

- une fois qu'on a du code intermédiaire, et que l'on sait où habite chaque temporaire, on peut cracher du SPIM
- il est plus chic d'avoir un type pour le code SPIM, plutot que de faire directement un `print` (possibilités de post-optimisations sur l'assembleur produit)

À faire

pour la semaine prochaine

- vous programmez le lambda-lifting (de Pascal à Pascal)
- si c'est déjà fait, vous entamez la suite
- vous réfléchissez de toute façon à la suite