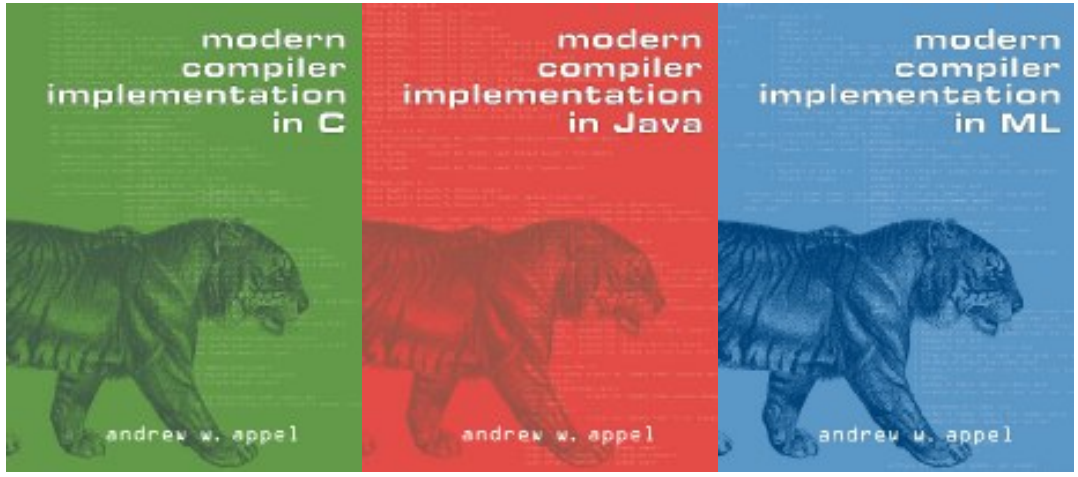
Projet 2: après l'analyse syntaxique

cours: Daniel Hirschkoff
TPs: Paul Renaud-Goud

Livres d'A. Appel



- disponibles à la bibliothèque
- chapitres 1-12
- vous pouvez y cueillir de l'inspiration
- le langage qui y est compilé, Tiger, partage certains aspects avec le sous-ensemble de Pascal qui nous préoccupe

Table des symboles (chap. 5 Appel)

- rôle: gérer questions de portée et de typage
  répondre à la question *"qui est x"?* pour x un identificateur utilisé quelque part dans le programme
- à chaque introduction d'identificateur est associée une "carte d'identité" — par exemple:
  - symboles de départ (p.ex. `writeln`)
  - variable locale, paramètre de fonction: nom, type, ...
  - nom de fonction: types des arguments et du résultat, fonction au-dessus, fonctions en-dessous, variables locales, nom du label dans le code généré, récursive, récursive terminale, fonction-feuille, ...
- c'est une structure de données qui n'existe qu'au moment de la compilation
  - typiquement, table de hachage
- implémentation
  - bien réfléchir à la structure (gérer portées imbriquées, notamment)
  - programmer proprement, car il se peut que vous fassiez évoluer la structure au fur et à mesure du développement du projet

Organisation

- je vous encourage à assister aux groupes de lecture de F. Zappa Nardelli (aujourd'hui et dans une semaine)
- TP en salle machine vendredi, 15h45-17h45?
  - objet: aide pour le DM
  - si oui, apportez vos portables (nous sommes en 125)
- binômes
  - qui compile quoi vers quoi en quoi
- rappel: DM pour le 17/2
- ensuite: premier rendu le 17 mars
  - description du sous-ensemble de Pascal sur la page du cours
  - analyses lexicale et syntaxique, analyse statique
  - rappel: exigences adaptées, uniformité de la notation pour ce qui est de la ponctualité/organisation

Représentation intermédiaire (chap. 7 livre A. Appel)

- le langage qui vient après l'AST

  7.1 dans le livre d'A. Appel offre une bonne illustration
- instructions:
  - MOVE
    - avec MOVE(TEMP t, e) et MOVE(MEM e1, e2)
  - JUMP(e,labs) labs liste les labels où l'on peut sauter (selon l'évaluation de e)
    - usage typique: JUMP(l,[l])
  - CJUMP(op,e1,e2,l1,l2) saute en li en fonction de op(e1,e2)
    - par ex. pour compiler $b1 \mid\mid b2$
  - SEQ(s1,s2) la séquence
  - CALL(f,arglist) appel de fonction avec ses arguments
- expressions:
  - constantes entières
  - labels
  - MEM(e) le mot mémoire situé à l'adresse e
  - TEMP(t) un temporaire (ou *registre virtuel*)
  - BINOP(o,e1,e2) opération binaire

Traduction vers la représentation intermédiaire

- pour traduire `x := y+2`, il faut en particulier traduire les accès à `y` (en lecture) et à `x` (en écriture)
  - où est `y` (par exemple) ?
    `y` peut être
    - un paramètre de la fonction courante
    - un paramètre d'une "fonction ancêtre"
    - une variable locale
- SPIM: conventions pour les registres
  - 0,1, 26-31: pas touche (usages spéciaux)
  - 2,3, v0,v1: évaluation, résultat de fonction
  - 4-7, a0-a3: arguments des fonctions
  - 8-15, 24, 25, t0-t9: à sauvegarder par l'appelant
  - 16-23, s0-s7: à sauvegarder et restaurer par l'appelé
- (et pour `a[n+2]` (tableau) ?)

Organisation d'un enregistrement d'activation (chap. 6)

arguments entrants

enregistrement précédent

← `$fp`

enregistrement courant

arguments sortants

← `$sp`

lien statique: pointeur vers l'enregistrement de la fonction qui *domine statiquement* (=dans le code) la fonction courante (car on a des fonctions imbriquées)

Localisation d'une variable

- dans un registre ou en mémoire (dans la pile)
  pas de tas en première approche
- temporaires: registres virtuels
  - "on ne sait pas"
  - tout comme les labels
- on sait dans certains cas dès l'analyse statique qu'il faut stocker en mémoire
  - paramètres/variables utilisés par des fonctions "au-dessous"
    - décalages à partir de `$sp` ou `$fp`, en suivant éventuellement des suites de liens statiques
  - paramètres/variables passés par référence dans un appel
    - on passe l'adresse
- dans le cas général, le code effectuant l'accès à la variable n'est engendré que plus tard, après avoir décidé "où elle habite"
- du coup, la taille de l'enregistrement d'activation d'une fonction n'est connue que *tard* dans le processus de compilation

Vers l'assembleur

- chapitre 9 A. Appel: un "langage assembleur abstrait"

```
datatype instr =   OPER of {      assem:string,
                                  dst:  temp list,
                                  src:  temp list,
                                  jump:  label list option }
                 | LABEL of {     assem:  string,
                                  lab:  Temp.label}
                 | MOVE of {      assem:  string,
                                  dst:  temp,
                                  src:  temp}
```

  - `MOVE` ne fait que transférer des données
  - `OPER` fait 'le reste' (calculs et sauts)
- au passage: une structure de données pour le code, pas un `print` directement (*post-optimisations* sur l'assembleur produit)
- ainsi une expression telle que BINOP(+,tmp1,BINOP(*,tmp2,tmp3)) est développée en quelque chose comme

```
tmp100 = tmp2 * tmp3    (dst [tmp100], src [tmp2;tmp3], jump [])
tmp101 = tmp100 + tmp1    (dst [tmp101], src [tmp1,tmp100], jump [])
```

Vivacité (chapitre 10 A. Appel)

- soit le code pseudo-assembleur

```
1         a ←0
2   L1:   b ←a+1
3         c ←c+b
4         a ←b*2
5         if a<12 goto L1
6         return c
```

  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
  - $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$ n nœud du graphe
  - $\text{out}[n] = \bigcup_{s \in succ(n)} \text{in}[s]$
  - si on regarde n: `c ←u+v`
    $\text{use}[n] = \{u, v\}$ $\text{def}[n] = \{c\}$ $\text{in}[n] = \{u, v\}$
- exemple: au tableau
  - l'information de vivacité est un saumon

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

- l'information de vivacité permet de représenter les interférences entre registres virtuels
  1. 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)
  2. pour une instruction n: `a`$\leftarrow$`c`, où les b1,..,bk sont les variables "live-out", n'ajouter (a,bi) *que si bi*$\neq$*c*
     - 2.1 puisque `a` et `c` contiennent la même valeur après n, pas d'interférence
     - 2.2 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"*
- première version du compilateur: sans allocation de registres, tout dans la pile
  - enregistrements d'activation obscènes