Quelques mots sur la compilation

- analyses lexicale et syntaxique: le *front end* d'un compilateur
  - cf. cours Projet 1
  - résultat: un *arbre de syntaxe abstraite*

```
type expr =
    Const of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | IfEq of expr*expr*expr*expr
                        (* if e1 = e2 then e3 else e4 *)
```

- *back end*: génération de code

  à partir d'un arbre (pas trop l'arbre de syntaxe abstraite, mais le résultat de potentiellement beaucoup de transformations et d'analyses) on engendre du code *linéaire*

La pile

- la structuration de programmes en fonctions qui s'appellent entre elles rend naturelle l'utilisation d'une **pile**
- sur la pile: *enregistrements d'activation*
  - ce sont des enregistrements *(records)*
  - leur présence sur la pile correspond à la période durant laquelle l'appel de la fonction est actif
- pour commencer, des expressions arithmétiques
  pas de fonction pour le moment

```
type expr =
    Const of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | IfEq of expr*expr*expr*expr
                        (* if e1 = e2 then e3 else e4 *)
```

Evaluer les expressions arithmétiques sur la pile

- pour évaluer `Add(Const 5, Const 7)`, on engendre

```
push 7
push 5
add
```

  - l'appel à `add`
    - dépile les arguments
    - fait le calcul
    - met le résultat sur le sommet de la pile
  - l'expression `Add(Const 5, Const 7)` est traduite en une suite d'instructions destinées à s'exécuter sur une *machine abstraite*
    - la pile + un jeu simple d'instructions
    - machine abstraite: on s'abstrait de l'architecture sous-jacente *(portabilité, p.ex. Java)*
- le sommet de la pile est accédé très fréquemment

  trois accès lors de l'exécution de `add`

Premier raffinement de la machine abstraite

- la pile n'est utilisée que pour stocker les arguments
- les calculs se font sur l'*accumulateur*, qui est un *registre* spécial
  - registre: accès rapide (il y en a généralement plus d'un!)
- cela donne

```
acc <- 7
push acc
acc <- 5
acc <- acc + top     (* opération atomique *)
pop
```

  - *leitmotiv*: on rend la pile dans l'état de propreté dans laquelle on l'a trouvée

Exemple un peu plus grand: 3 + (7 + 5)

```
acc <- 3
push acc
acc <- 7
push acc
acc <- 5
acc <- acc + top
pop
acc <- acc + top
pop
```

- but du jeu
  - jeu d'instructions simples
  - traitement uniforme (ici: évaluer une addition)
  - avec des instructions correspondant à des opérations élémentaires en machine
  - le résultat est en `acc` à la fin

Raffinement encore: 'programmer' la pile soi-même

```
                                        commentaires
acc <- 7          ORI R1 R0 7     R1 := 7
push acc          PSH R1 R30 4      (R30 = registre de pile)
acc <- 5          ORI R1 R0 5     R1 := 5
acc <- acc + top  POP R29 R30 4   R29 := sommet de la pile
                  ADD R1 R1 R29   R1 := R1+R29
```

- `ORI Rr R0 k` *met le registre `Rr` à `k`* (fait un "or" avec 0)
- `ADD r1 r2 r3` *ajoute les contenus de `r2` et `r3` et stocke cela en `r1`*
- `R30` registre contenant l'adresse de la fin de la pile
- `PSH r1 r2 k` recopie le contenu de `r1` à l'adresse pointée par `r2`, et décale `r2` de `k` (octets)
- `POP r1 r2 k` recopie la valeur pointée par `r2` en `r1` et décale `r2` de `k`
- le résultat est en `R1` à la fin, `R30` contient le *pointeur de pile*, `R0` contient toujours la valeur 0

Le backend: de l'arbre au fichier objet

- `Const i` → `print_string("ORI R1 R0 i\n");`
- `Add (e1,e2)` →

```
engendre e1;
print_string "PSH R1 R30 4\n";
engendre e2;
print_string "POP R29 R30 4\n";
print_string "ADD R1 R1 R29\n";
```

compilation vs interprétation:

cf. en TP

```
let v1 = eval e1 in
let v2 = eval e2 in
v1+v2
```

Branchements (in)conditionnels

`if e1=e2 then e3 else e4`

```
              ''engendre(e1)''
              PSH R1 R30 4
              ''engendre(e2)''
              POP R29 R30 4
              CMP R1 R1 R29        R1 := R1-R29
              BEQ R1 branche_vrai  saute si R1=0
branche_faux: ''engendre(e4)''
              BSR fin_if           saute inconditionnellement
branche_vrai: ''engendre(e3)''
fin_if:       ...
```

Les fonctions

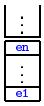
- l'enregistrement d'activation d'un appel de fonction contient
  - les paramètres d'appel de la fonction nécessaires à l'exécution de l'appel
    - en réalité, c'est plus compliqué (on a intérêt à passer les paramètres dans des registres)
  - + de quoi entrer et sortir dans la fonction
- deux morceaux de code à engendrer
  - préparer la pile avant l'appel et lancer l'appel
    `f(3,12,4)`
  - corps de la fonction: faire le calcul, et sortir de la fonction
    `int f(x,y,z){...}`
- ce que l'on va raconter est *juste un exemple*
  - beaucoup de facteurs à prendre en compte pour définir l'organisation de la pile et les conventions d'appel des fonctions *(langage source, architecture cible, ...)*

Appel de la fonction: `f(e1,...,en)`

R30: sommet de la pile

```
''engendre(en)''
PSH 1 30 4
⋮
''engendre(e1)''
PSH 1 30 4
BSR code_f
```

on stocke sur la pile les arguments de la fonction puis on y va



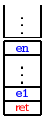
- BSR: saute à l'adresse donnée, et stocke l'adresse qui suit le BSR dans un registre spécial, R31
- lorsqu'on saute,
  - R30 pointe vers l'extrémité de la pile, qui a grandi
  - R31 contient l'adresse de l'instruction a exécuter en retournant de l'appel

Au sein de la fonction `f(x1,...,xn)=corps`

R30: sommet de la pile
R31: adresse de retour

```
code_f:  PSH 31 30 4
         ''engendrer(corps)''
         POP R31 R30 4
         ADDI R30 R30 ''4*n''
         JSR R31
```

- on stocke ce que contient R31 dans la pile l'enregistrement fait donc 4*n+4 octets
- on calcule le corps de la fonction
- on remet l'adresse de retour dans R31
- on fait décroître la pile
- on sort (JSR: sauter à l'adresse contenue dans un registre)



dans le corps de la fonction, le ième argument est atteint par R30+4*i+4

- on ne voit pas apparaître R1: il est dans `corps`
- otpimisation: si `corps` ne fait pas d'appel de fonction, ne pas toucher à R31

Quelques mots encore

- un "compilateur": un front end, un back end
  DÉMO
- il s'agit ici d'un exemple
  - autres jeux d'instructions
  - autres conventions d'appel pour les fonctions
    (si possible, tous les paramètres dans les registres)
  - ⋮
- cet assembleur est encore "haut niveau"
  - les sauts sont en réalité calculés en indiquant explicitement le décalage (au lieu d'utiliser un label)
- ce n'est pas fini: optimisations
  - *inlining*: `let f x = x+1, for i=1 to 2 do BLA`
  - *"jump around"*
  - ...