Redescendre de l'arbre

- analyses lexicale et syntaxique: le *front end* d'un compilateur
- *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

```
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

```
acc <- 7              li $a0, 7
push acc              sw $a0, 0($sp)
                      addiu $sp, $sp, -4
acc <- 5              li $a0, 5
acc <- acc + top      lw $t1, 4($sp)
                      add $a0, $a0, $t1
pop                   addiu $sp, $sp, 4
```

- `$sp` registre contenant l'adresse de la fin de la pile
- `li reg, imm` *met le registre `reg` à `imm`*
- `sw reg1, decalage(reg2)` *stocke le contenu de `reg1` a l'adresse (contenu de `reg2`)+`decalage`*
- `addiu reg1, reg2, imm` *ajoute `imm` au contenu de `reg2` et stocke dans `reg1`*
- `add reg1, reg2, reg3` *ajoute les contenus de `reg2` et `reg3` et stocke cela en `reg1`*
- `lw reg1, decalage(reg2)` *charge ce qui est stocké en l'adresse en `reg2` + `decalage` dans `reg1`*
- le résultat est en `$a0` à la fin

Le backend: de l'arbre au fichier objet

- `Const i` $\longrightarrow$ `print_string("li $a0 i\n");`
- `Add (e1,e2)` $\longrightarrow$

```
engendre(e1);
print_string("sw $a0, 0($sp)\n");
print_string("addiu $sp, $sp, -4\n");
engendre(e2);
print_string("lw $t1, 4($sp)\n");
print_string("add $a0, $a0, $t1\n");
print_string("addiu $sp, $sp, 4\n");
```

Branchements (in)conditionnels

```
if e1=e2 then e3 else e4
```

```
                    ‘‘engendre(e1)’’
                    sw $a0, 0($sp)
                    addiu $sp, $sp, -4
                    ‘‘engendre(e2)’’
                    lw $t1, 4($sp)
                    addiu $sp $sp 4
                    beq $a0, $t1, branche_vrai
  branche_faux:     ‘‘engendre(e4)’’
                    b fin_if
  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
  - $+$ de quoi entrer et sortir dans la fonction
- deux morceaux de code à engendrer
  - préparer la pile avant l'appel et lancer l'appel
  - corps de la fonction: faire le calcul, et sortir de la fonction

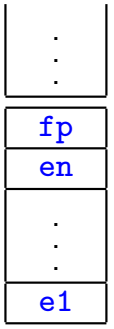
Appel de la fonction: `f(e1,...,en)`

```
sw $fp 0($sp)
addiu $sp $sp, -4
‘‘engendre(en)’’
sw $a0 0($sp)
addiu $sp $sp, -4
:
:
‘‘engendre(e1)’’
sw $a0 0($sp)
addiu $sp $sp, -4
jal code_f
```

on stocke sur la pile

- la valeur de `$fp`
- les arguments de la fonction

puis on y va



- `$fp`: frame pointer, pointeur dans la pile, vers le **début** de l'enregistrement d'activation
- `jal`: saute à l'adresse donnée, et `stocke` l'adresse qui suit le `jal` dans un registre spécial, `$ra`
- lorsqu'on saute,
  - `$sp` pointe vers le haut de la pile, qui a grandi
  - `$fp` pointe encore vers l'ancien `$fp`, qui a été stocké dans la pile
  - `$ra` contient l'adresse de l'instruction a exécuter en retournant de l'appel

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

```
code_f:   move $fp $sp
          sw $ra 0($sp)
          addiu $sp $sp, -4
          ``engendrer(corps)''
          lw $ra 4($sp)
          addiu $sp $sp ``4*n+8''
          lw $fp 0($sp)
          jr $ra
```

- on met `$fp` à `$sp`
- on stocke ce que contient `$ra` dans la pile
  l'enregistrement fait donc `4*n+8` octets
- on calcule le corps de la fonction
- on remet l'adresse de retour dans `$ra`
- on fait décroître la pile
- on remet `$fp` où il était avant l'appel de la fonction
- on sort (`jr`: sauter à l'adresse contenue dans un registre)

- on ne voit pas apparaître `$a0`: il est dans `corps`
- pourquoi avoir stocké `$fp`?
  - on veut pouvoir accéder aux paramètres de la fonction (`x1,...,xn`)
  - `$sp` bouge tout le temps, on calcule un décalage à partir de `$fp` *(qui pointe juste au-dessus de l'adresse de retour)*
  - c'est pour ça qu'on a empilé les arguments "à l'envers"
- otpimisation: si `corps` ne fait pas d'appel de fonction, ne pas toucher à `$ra`

Quelques mots encore

- voilà notre premier “compilateur” : un front end, un back end
  - DÉMO
  - on aurait pu tout faire en une passe
    - au lieu de `| expr PLUS expr { Add($1,$3) }`, mettre `| expr PLUS expr { $1 + $2 }`
    - seulement tant que le langage est nul (`let f x = ...`)
- 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)
  - ⋮
- ce n’est pas fini : optimisations
  - *inlining* : `let f x = x+1, for i=1 to 2 do BLA`
  - *“jump around”*
  - ...