Compilation avancée - TD

Jordi Bertran de Balanda

TD 1

```
e ::= e + e
    | e * e
    | n
```

Problème: ambiguités

```
e ::= f + e | f
f ::= t * f | t
t ::= n
```

```
e  ::= t e'
e' ::= + t e' | E
t  ::= f t'
t' ::= * f t' | E
f  ::= n | (e)
```

On définit le premier caractère de chaque règle:

```
Premier(e) = Premier(t)
Premier(t) = Premier(f)
Premier(f) = {n, (}
Premier(e') = {+, E}
Premier(t') = {*, E}
```

n	+	*	(	)	$
e	e->te'			e->te'	
e'		e'->+te'			e'->E
t	t->ft'			t->ft'	
t'		t'->E	t'->*ft'		t'->E
f	f->n			f->(e)	

On calcule le caractère suivant pour les règles terminales:

```
Suivant(e) = Suivant(e') = {), $}
Suivant(t) = Suivant(t') = Premier(e') U {), $} = {+, ), $}
Suivant(f) = {+, *, ), $}
```

Parsing de 2*3+5

```
e -> te'
  -> fte'
  -> 2te'
  -> 2*ft'e'
  -> 2*3t'e'
  -> 2*3Ee'
  -> 2*3E+te'
  -> 2*3E+ft'e'
  -> 2*3E+5t'e'
  -> 2*3E+5EE
```

TD 2

Lambda-calcul

```
($\lambda$x.$\lambda$y.y)(($\lambda$x.(x x))($\lambda$x(x x)))
```

AST: cf. feuille

Réduction;

- de gauche à droite
- la définition la plus à l'intérieur
- sans réduire les abstractions

Exemple: Choix de l'expression à réduire: cf. feuille

(λx.λy.y)**((λx.(x x))(λx(x x)))**

(x x)[x -> λx.(x x)]

(x[x -> λx.(x x)] x[x -> λx.(x x)])

(λx.(x x))(λx.(x x))

-> Réduction infinie - expression appliquée à elle-même = wtf.

Réduction en ordre normal:

- de gauche à droite
- application la plus à l'extérieur

(λx.λy.y)((λx.(x x))(λx(x x)))

(λy y)[x -> (λx.(x x))(λx.(x x))]

/! Substitution dans une abstraction - **attention** aux variables *liées*.

λy (y[x -> (λx.(x x))(λx.(x x))])

(λy.y)

Encodage de Church

Entiers

```
0 $\equiv$ $\lambda$f.$\lambda$n.n
1 $\equiv$ $\lambda$f.$\lambda$n.f n
2 $\equiv$ $\lambda$f.$\lambda$n.f (f n)
..

succ t $\equiv$ $\lambda$t.$\lambda$f.$\lambda$n.f((t f) n)

succ 0 $\equiv$ ($\lambda$t.$\lambda$f.$\lambda$n.f((t f) n))($\lambda$f.$\lambda$n.n)
       $\equiv$ ($\lambda$f.$\lambda$n.f((($\lambda$f.$\lambda$n.n) f) n))
       $\equiv$ ($\lambda$f.$\lambda$n.f(($\lambda$n.n) n))
       $\equiv$ ($\lambda$f.$\lambda$n.(f n))

add u v $\equiv$ $\lambda$u.$\lambda$v.$\lambda$f.$\lambda$n.u f (v f n)

add 1 2 $\equiv$ ($\lambda$u.$\lambda$v.$\lambda$f.$\lambda$n.u f (v f n))($\lambda$f.$\lamb
        $\equiv$ $\lambda$f.$\lambda$n.($\lambda$f.$\lambda$f. n) f (($\lambda$f.$\lambda$n.
        $\equiv$ $\lambda$f.$\lambda$n ($\lambda$n.f n) (f (f n))
        $\equiv$ $\lambda$f.$\lambda$n f (f (f n))

mult u v $\equiv$ ??
```

Booléens

```
Condition: $\lambda$t.$\lambda$f.$\lambda$c
True:  $\lambda$a.$\lambda$b.a
False: $\lambda$a.$\lambda$b.
```

```
If  : $\lambda$t.$\lambda$f.$\lambda$c c t f
Not : $\lambda$b.condition false true b
        $\lambda$b.b ($\lambda$a.$\lambda$b b)($\lambda$a.$\lambda$b b)
```

Compilation

```
e::= n
   | e + e
```

Machine: stack et code

Opérations: ADD et PUSH n

Comp(n) $\equiv$ PUSH n Comp(a + b) $\equiv$ Comp(a), Comp(b), ADD

Comp(2+3+5) $\equiv$ Comp(2+3); Comp(5); ADD $\equiv$ Comp(2); Comp(3); ADD; Comp(5); ADD $\equiv$ PUSH 2; PUSH 3; ADD; PUSH 5; ADD

Code | Pile | Code | Pile

PUSH n; c | s | c | s.n ADD; c | s.a.b | c | s.(a+b)

TD 2

Lambda calcul (mul, encodage de Church)

```
e := x
  |  $\lambda$x.e
  |  e e

n $\equiv$ $\lambda$f$\lambda$n f .. (f n)
succ $\equiv$ $\lambda$t.$\lambda$f.$\lambda$n f (t f n)
add $\equiv$ $\lambda$t.$\lambda$u.$\lambda$f.$\lambda$n(t succ)
mul $\equiv$ $\lambda$t.$\lambda$u.$\lambda$f.$\lambda$n t (add u) ($\lambda$f.$\lambda$n.n)
```

Machine de Krivine

Schéma d'évaluation

Commande	Env	Pile	Com	Env	Pile
PUSH(C'); C	E	S	C	E	S.C'[E]

Commande	Env	Pile	Com	Env	Pile
GRAB; C	E	S.C'[E']	C	E.C'[E']	S
ACCESS(n); C	E	S	C	E' (E(n) = C'[E'])	S

Schéma de compilation

- C(n) = ACCESS(n)
- C(λ.a) = GRAB; C(a)
- C(a b) = PUSH(C(b)); C(a)

Exemple

Compilation de T $\equiv$ ((λx.λy x) 2) 3

SECD

```
C((($\lambda$x.$\lambda$y x) 2) 3) = C(($\lambda$x.$\lambda$y x) 2); C(3); APPLY
                     = C($\lambda$x.$\lambda$y x); C(2); APPLY; C(3); APPLY
                     = Closure(C($\lambda$y x); RETURN); C(2); APPLY; C(3); APPLY
                     = Closure(Closure(C(x); C($\lambda$y x); RETURN); RETURN); C(2); APPLY;
                     = Closure(Closure(ACCES(1); RETURN); RETURN); INT(2); APPLY; INT(3); A
```

<table>
<tr><th>C</th><th>E</th></tr>
<tr><td>INT(2); APPLY; INT(3); APPLY</td><td>‘| S;Closure(Closure(ACCES(1); RETURN); RETURN</td></tr>
<tr><td>Closure(Closure(ACCES(1); RETURN); RETURN)</td><td>‘’;INT(2) | S;(INT(3);APPL</td></tr>
<tr><td>RETURN</td><td>‘’ | S;(INT(3);APPLY);’;(ACCES(1); RE</td></tr>
<tr><td>INT(3); APPLY</td><td>’</td></tr>
</table>

Krivine

```
C((($\lambda$x.$\lambda$y x) 2) 3) = PUSH(C(3)); C(($\lambda$x.$\lambda$y x) 2)
                   = PUSH(INT(3)); PUSH(C(2)); C($\lambda$x.$\lambda$y x)
                   = PUSH(INT(3)); PUSH(INT(2)); GRAB; C($\lambda$y x)
                   = PUSH(INT(3)); PUSH(INT(2)); GRAB; GRAB; C(x)
                   = PUSH(INT(3)); PUSH(INT(2)); GRAB; GRAB; ACCES(1)
```

C	E	S
PUSH(INT(2)); GRAB; GRAB; ACCES(1)	'| INT(3) GRAB; GRAB; ACCES(1) |'	INT(3); INT(2)
GRAB; ACCES(1)	INT(2)	INT(3)
ACCESS(1)	INT(2); INT(3)	'INT(2) |'

Récursion

- Fact = λn if n = 0 then 1 else n * Fact(n-1)
  - Problème: le symbole Fact n'existe pas. Abstraction => intrduction d'un symbole
- Fact = λFact.λn if n = 0 then 1 else Fact Fact (n-1)
  - Problème: réduire fact fact => Application de fonction
- Fact = λFact.**(λf.λn if n = 0 then 1 else n * f(n-1))** (Fact Fact)
  - En gras: terme G.

```
Fact = $\lambda$Fact.G (Fact Fact)
     = ($\lambda$Fact.G (Fact Fact))($\lambda$Fact.G (Fact Fact))
     = ($\lambda$f.($\lambda$x.f (x x)) ($\lambda$f.($\lambda$x.f (x x))) G
       ---------------------------------
                Y COMBINATOR
(Y $\lambda$f.$\lambda$n if n = 0 then 1 else n * f(n-1))
```

Mini-ML

Transcrire de l'application partielle de fonctions en Java.

Exemples de code à traduire

```
let h =
    let f = fun a b c -> a + b + c in
    let g = f 2 in
    g
in h 3 5

let c = 2 in
    let f =
        a * b * c
```

```
  ..

let rec fact n =
    if n = 0 then 1
    else n * fact (n-1)
```

Classe MLValue: valeurs retranscries pour ML

Classe MLFun:

- invoke
  - Tous les arguments sont passés -> invoke_real
  - Sinon ajout de l'arg dans l'environnement (tableau)
- invoke_real
- compteur - 0 à la création

Let rec: représenter la fonction dans l'environnement.

Variables libres

```
FV(x) = x
FV($\lambda$x.e) = FV(e) \ {x}
FV(a b) = FV(a) U FV(ab)
```

TD 4 - Garbage Collectors

Simulation d'algorithmes de GC

Compteur de références

TD6 - Assembleur

Ex1. Lecture ASM

Délimitation des fonctions:

```
.ent nom_fct
nom_fct:
     Code
     Code
     ..
.end nom_fct
```

- test_asm32.s
  - sum
  - max_min_tab
  - mat_mul
  - mul
  - main
- ex_asm.s
  - max2
  - max
  - max_tab
  - main

Ex2. Blocs de base

Reconnaissance des blocs de base:

- A partir du début
- Jusqu'aux delayeded slots (inclus) correspondant à un jump

Graphes: izi.

Ex3. Blocs dominants

Cf. feuille

Ex4. Arcs retour

Arc retour: arc revenant vers un noeud dominant le noeud.

Blocs inclus: tous les blocs atteints en chemin inverse à partir du bloc de fin de l'arc retour jusqu'à atteindre son début.

mat_mul

Arc retour	Blocs inclus
2 *rightarrow* 3	2, 3

Arc retour	Blocs inclus
4 *rightarrow* 5	4, 3, 2, 1, 5
13 *rightarrow* 14	13, 12, 8, 10, 9, 11, 7, 14
9 *rightarrow* 10	9, 10
11 *rightarrow* 12	11, 10, 9, 8, 12

- 2 boucles imbriquées: 2 *rightarrow* 3, 4 *rightarrow* 5
- 3 boucles imbriquées: 9 *rightarrow* 10 dans 11 *rightarrow* 12 dans 13 *rightarrow* 14

min_max_tab

Arc retour	Blocs inclus
4 *rightarrow* 5	4, 2, 3, 1, 5

Préparation du TP

Q1.

“‘c++ Line* debut = nullptr;

for (int i = 0; i < n; i++) { if (instr(i).type() == label) { if (debut != nullptr) { // Bloc sans saut res.add(new BasicBlock(debut, instr(i-1), NULL) // NULL: saut debut = instr(i); } } else if (instr(i).type() == saut) { if (debut != nullptr) { res.add(new BasicBlock(debut, instr(i+1), instr(i))); i = i + 1; } else { // debut == nullptr res.add(new BasicBlock(instr(i), instr(i+1), instr(i))); debut = nullptr; } } else { if (debut == nullptr) debut == instr(i); } } if (debut != nullptr) res.add(new BasicBlock(debut, instr(i), nullptr));

```
 TD7

# Rappels

## Dépendances

Types de dépendances:

* contrôle
```

```
* données
    * RAW
    * WAR
    * WAW
    * mémoire:
        * lecture/écriture
```

R4 <- Mem[A] RAW R7 -> Mem[A]

add R1, R2, R2 WAR addi R2, R1, 7

add R1, R2, R2 WAW addi R1, R2, 7

R3 -> Mem[C] dépendance mémoire R7 -> Mem[D] C = D? C chevauche D? “‘

Dans un bloc de base

- Au plus 1 dépendance RAW par registre source (2 maximum dans MIPS32)
- Au plus j dépendances WAR pour l'instruction i_j. Si k $\in [0, j-1]$ l'indice de la dernière l'instruction qui écrit dans le registre source de i_j , au plus $j - k$ dépendances.
- Au plus 1 dépendance WAW (avec l'instruction précédente écrivant dans le registre

Ex1. DAG, chemin critique et ordonnancement

Q1.

- Registre défini:
  - On remonte en notant les lectures, on s'arrête à la première redéfinition.
- Registre utilisé:
  - On remonte en s'arrêtant à la première écriture.

```
main:
    lw $4, 0($6)
    lw $2, 0($4)
    add $5, $14, $2
    ori $10, $6, 0
    sw $5, 0($10)
    lw $2, -12($10)
    addi $5, $2, 4
```

```
    bne $5, $2, $L5
    add $0, $0, $0     BLOC 1

    lw $4, 0($6)
    lw $2, 0($7)
    add $5, $4, $2
    sw $5, 0($6)
    addiu $12, $8, 2
    addiu $7, $12, 1
    bne $7, $0, $L5
    add $0, $0, $0     BLOC 2

    sub $6, $0, $5
    sll $6, $3, 4
    addiu $5, $6, -2
    sw $15, 12($7)
    sw $10, -4($6)     BLOC 3

$L5:
    sub $8, $10, $15
    sll $10, $10, 4
    sw $8, 8($7)
    add $10, $8, $10
    sw $10, 12($7)
    jr $31
    add $0, $0, $0     BLOC 4
```

Q2.

Chemin critique: 1 2 3 6 7 8.

Q4.

inst_j sort au cycle t = max {cycle de sortie de inst_{j-1} + 1, max(cycle de p + délai de la dépendance p vers inst_j)}