

Correction du partiel CMP2 de S2

EPITA – Promo 2015

**Documents (notes de cours, photocopiés, livres) autorisés
Machines (calculatrices, ordinateurs, tablettes, téléphones) interdites.**

Juin 2013 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain.

Best-of: Le *best-of* est tiré des copies des étudiants.

Lisez bien les questions, chaque mot est important. Écrivez court, juste et bien ; servez-vous d'un brouillon. Une argumentation informelle mais convaincante est souvent suffisante. Gérez votre temps, ne restez pas bloqué sur les questions les plus difficiles. Une lecture préalable du sujet est recommandée. Cette correction contient 12 pages. Les pages 1–10 contiennent l'épreuve et son corrigé. Les pages 11–12 sont dédiées aux annexes.

1 Généralités

1. Rappelez quelles sont les quatre phases de la partie terminale (*back end*) d'un compilateur comme tc.

Correction:

- canonisation de la représentation intermédiaire (HIR → LIR) ;
- sélection d'instructions (génération de code) ;
- analyse de la vivacité des variables ;
- allocation de registre

Best-of:

- Exécution
- Transtypage dynamique manifeste
- Traduction langage Tree vers langage HIR
- (...) Dernière étape : MISP (*sic*)
- TC9 : magic !

2. Qu'est-ce que le *frame pointer* ?

Correction: Le *frame pointer* est une variable pointeur référençant une position fixe relativement au cadre de pile de l'activation courante (fonction appelée), par exemple au début (la « frontière interne » du cadre). Il est utilisé pour accéder aux données du cadre de pile (arguments passés via la pile, variables locales, temporaires, adresse de retour, etc.). Un registre (matériel) est généralement dédié au stockage du *frame pointer* courant.

Correction: (suite) Dans beaucoup de copies, le terme « frame pointer » a été confondu avec d'autres notions (parfois liées, certes, mais différentes) :

- le cadre de pile (*stack frame*) lui-même ;
- le lien statique (*static link*) ;
- le pointeur d'instruction ou « compteur ordinal » (*instruction pointer* ou *program counter*) ;
- ou encore, le concept de *smart pointer* !

Best-of:

- Un pointeur (*C'est pas faux...*)
- Il s'agit d'un pointeur spécial.
- Frame pointer : pointer sur le registre.
- Frame pointeur : pointeur vers fonction.
- Le frame pointer indique l'adresse de fin de la fonction.
- Le frame pointer pointe sur le début du scope courant.

3. Citez un exemple de langage intermédiaire à base de pile.

Correction: On peut citer :

- le bytecode Java, exécuté par la machine virtuelle Java (JVM) ;
- le Common Intermediate Language (CIL) de la plate-forme .NET ;
- le HipHop bytecode (HHBC) exécuté par la HipHop Virtual Machine (HHVM) ;
- le langage intermédiaire UCODE, utilisé dans des compilateurs PA-RISC et MIPS.

Plusieurs copies proposent comme réponse HIR, ce qui est imprécis (dans le cadre de tc, le langage intermédiaire s'appelle TREE ; la représentation intermédiaire haut-niveau (HIR) est une traduction du programme d'entrée vers ce langage intermédiaire) et incorrect (TREE est un langage à base de registre).

Attention aussi aux abus de langage : le bytecode Java est un langage intermédiaire à base de pile, mais ce n'est pas le cas du langage Java lui-même !

Best-of:

- Le C
- Le C++
- Haskell
- Tiger
- OCaml
- lisp
- MIPS
- MISP (*lu sur plusieurs copies !*)
- Assembleur Sparck (*sic*)

```
mem
binop add
temp fp
const -4
```

4. Citez un exemple de langage intermédiaire à base de registres.

Correction: Quelques réponses possibles :

- le langage TREE, utilisé comme représentation intermédiaire (IR) dans tc ;
- Le langage RTL (Register Transfer Language) utilisé pour l'une des IR de GCC ;
- le langage LLVM (sous forme d'IR en mémoire pour compilateur, de « bitcode » (format binaire) ou encore de langage d'assemblage lisible par l'être humain) ;
- le bytecode Dalvik, utilisé par la machine virtuelle du même nom sous le système Android ;
- le jeu d'instructions de BEAM, la machine virtuelle d'Erlang.

Même si c'est un abus de langage (cf. réponse à la question précédente), j'ai accepté HIR ou LIR comme bonne réponse.

Best-of:		
– Le C	– askell (<i>sic</i>)	– THREE (<i>sic</i>)
– Le C++	– Lips (<i>sic</i>)	– YAIL (<i>sic</i>)
– L'Ada	– Langages fonctionnels	– temp t1

5. Comment se définit un bloc basique (*basic block*) ?

Correction: Une suite d'instructions dans laquelle « on entre toujours par le début et on sort toujours par la fin », telle que

- sa première instruction est une étiquette (`label` en `TREE`) ;
- sa dernière instruction est un saut (`jump` ou `cjump` en `TREE`) ;
- elle ne contient aucune autre occurrence d'étiquette ou de saut.

Dans certaines copies, la notion de bloc basique a été confondue avec celle de bloc d'activation (*activation block*), un autre nom pour le cadre de pile (*stack frame*). Le premier appartient au monde de la compilation, tandis que le second est du ressort de l'exécution.

D'autres fois, il y a eu confusion avec un bloc au sens d'un langage haut niveau ; ou encore avec un fragment de code (procédure compilée composée d'un prologue, d'un corps et d'un épilogue).

Best-of:
– Un bloc de code contenant une instruction.
– Une succession d'opération entre deux mots-clés.
– Espace mémoire qui contient des données non activables.
– Un block basique contient des déclarations.

6. Quelle est la finalité de l'opération consistant à découper un programme en blocs basiques puis à les ré-assembler en une « trace » ?

Correction: À faciliter plus tard la traduction des sauts conditionnels à deux branches (`cjump`) en saut conditionnels à une branche, en faisant suivre un `cjump` par l'étiquette (`label`) de la condition fausse.

Best-of:
– Pour le binding.
– Effectuer l'allocation des registres.
– Cette opération a pour finalité d'aboutir à la compilation.

7. Il existe une optimisation qui consiste à se débarrasser du `frame pointer` dans le code généré. Citez un exemple où cette optimisation n'est pas réalisable.

Correction: Le `frame pointer` devient essentiel dès lors que la taille du cadre de pile (bornée par le `stack pointer`) n'est plus statiquement connue. En effet, le `stack pointer` ne peut alors plus servir de point de référence pour accéder au contenu du cadre. On peut citer les deux situations suivantes :

- l'allocation dynamique sur la pile (via la routine `alloca` ou les tableaux dynamiques du C99 par exemple), qui déplace le `stack pointer` dynamiquement pour agrandir le bloc d'activation ;
- les fonctions à liste d'arguments variable (comme `printf`), dont la taille du cadre de pile n'est pas connue statiquement.

Best-of:
– Dans une boucle <code>for</code> .
– Dans une fonction récursive.
– Lorsqu'il y a des sauts conditionnels.
– Cette optimisation n'est pas réalisable pour la fonction <code>main</code> principale.
– Cette opération n'est pas réalisable lorsqu'il n'y a pas de <code>frame pointer</code> .

2 Traduction

Dans cet exercice, on s'intéresse à la traduction de programmes Tiger vers la représentation intermédiaire haut niveau (HIR) de tc. En vous aidant de l'annexe A, traduisez les expressions Tiger qui suivent dans le langage TREE, au choix en utilisant la syntaxe concrète du langage TREE ou en les représentant sous forme d'arbres.

On observera les consignes suivantes :

- On ne s'intéressera qu'à la traduction des expressions seules et non à celle d'une fonction qui les contiendrait : il est donc inutile de faire apparaître un prologue ou un épilogue.
- De même, on n'utilisera pas de traduction « intelligente » basée sur les catégories d'expressions (« traductions retardées ») Ex, Nx, Cx.
- On considérera que toutes les variables sont locales et allouées sur la pile. Pour y accéder, il sera donc nécessaire de construire une expression représentant un décalage (*offset*) par rapport au frame pointer ('temp fp' en TREE). Pour une variable foo, on considérera que ce décalage est égal à la constante entière nommée foo ('const foo' en TREE).
- On rappelle qu'en Tiger toutes les variables simples, champs d'enregistrements, cases de tableaux ont la taille (fixe) d'un mot mémoire de 32 bits.
- On rappelle enfin que les tableaux et enregistrements Tiger sont alloués sur le tas et manipulés par pointeurs.

1. 1 + 2 * 3

Correction:

```
binop add
  const 1
  binop mul
    const 2
    const 3
```

2. (1; 2; 3)

Correction:

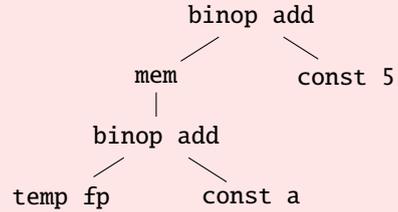
```
eseq
  seq
    sxp
      const 1
    sxp
      const 2
  seq end
  const 3
```

Best-of:

3. $a + 5$

Correction:

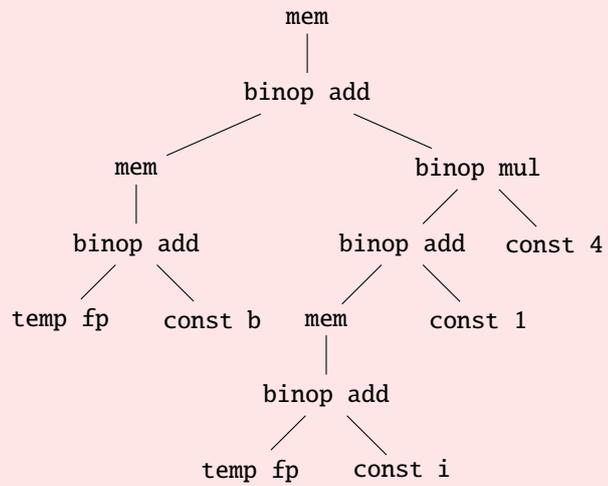
```
binop add
  mem
    binop add
      temp fp
      const a
    const 5
```



4. $b[i + 1]$

Correction:

```
mem
  binop add
    mem
      binop add
        temp fp
        const b
      binop mul
        binop add
          mem
            binop add
              temp fp
              const i
            const 1
          const 4
```

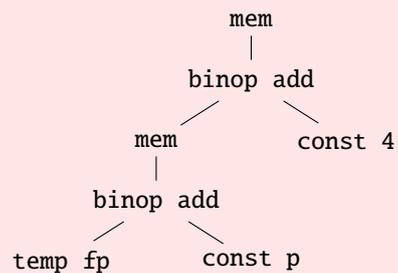


5. $p.y$

où p est une variable de type m , défini comme suit : `type m = { x : int, y : int }`

Correction:

```
mem
  binop add
    mem
      binop add
        temp fp
        const p
    const 4
```



Best-of:

```
jump (const (y))
```



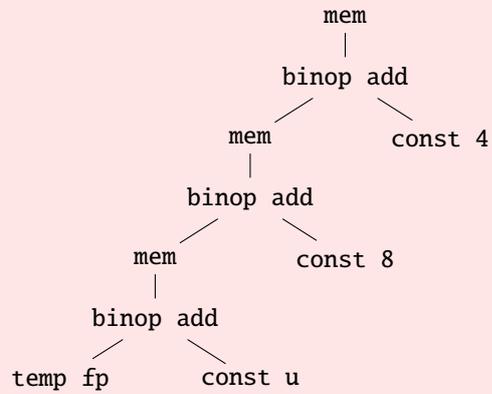
6. u.c.b

où u est une variable de type n, défini comme suit: **type** n = { a : int, b : int, c : n }

Correction:

```

mem
  binop add
    mem
      binop add
        mem
          binop add
            temp fp
            const u
          const 8
        const 4
    const 4
  
```

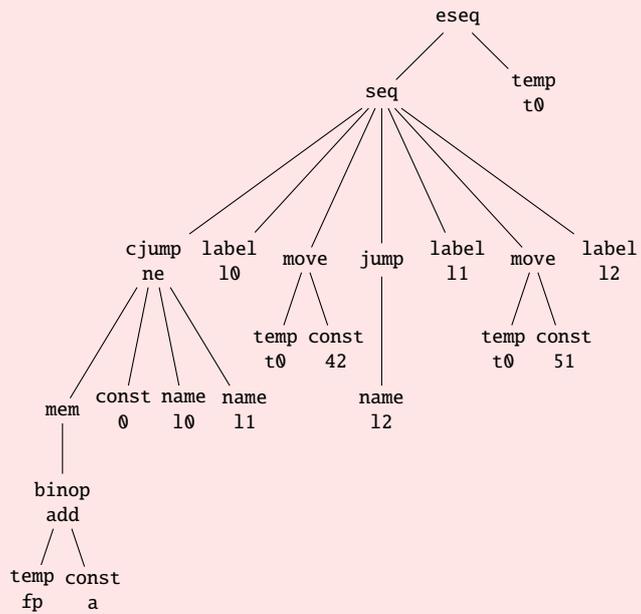


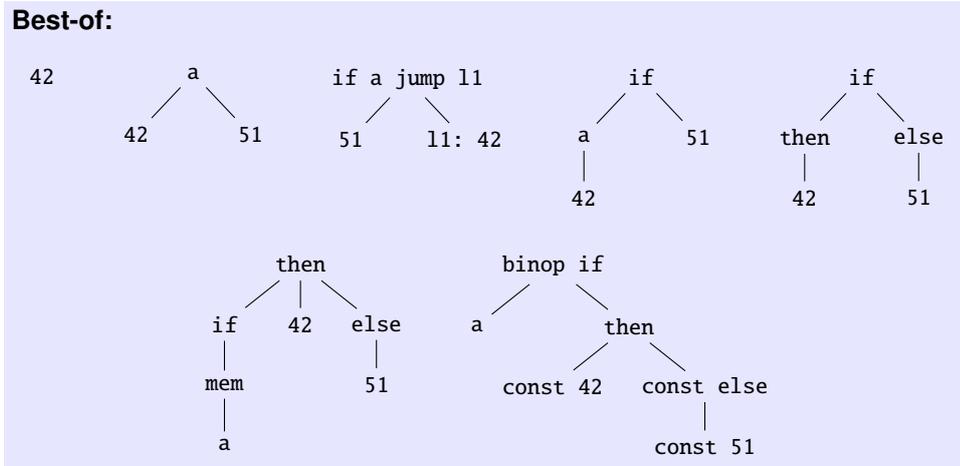
7. if a then 42 else 51

Correction:

```

eseq
  seq
    cjump ne
      mem
        binop add
          temp fp
          const a
        const 0
      name 10
      name 11
      label 10
      move
        temp t0
        const 42
      jump
        name 12
      label 11
      move
        temp t0
        const 51
      label 12
    seq end
  temp t0
  
```





3 Linéarisation

La première étape de la transformation de la représentation intermédiaire haut-niveau (HIR) vers sa version bas-niveau (LIR) est la *linéarisation*. Un des objectifs de cette opération est la suppression des nœuds eseq par application de règles de réécriture d'arbres. On s'intéresse ici à l'application de telles règles à plusieurs motifs HIR. Complétez les transformations suivantes en écrivant la partie droite (sur votre copie, pas sur l'énoncé).

1. $\text{sxp}(\text{eseq}(s, e)) \Rightarrow$

Correction:

$\text{sxp}(\text{eseq}(s, e)) \Rightarrow \text{seq}(s, \text{sxp}(e))$

2. $\text{sxp}(\text{call}(\text{eseq}(s, e1), \text{const}(4), e2)) \Rightarrow$

Correction:

$\text{sxp}(\text{call}(\text{eseq}(s, e1), \text{const}(4), e2))$
 $\Rightarrow \text{sxp}(\text{eseq}(s, \text{call}(e1, \text{const}(4), e2)))$
 $\Rightarrow \text{seq}(s, \text{sxp}(\text{call}(e1, \text{const}(4), e2)))$

3. $\text{move}(\text{mem}(\text{eseq}(s, e1)), e2) \Rightarrow$

Correction:

$\text{move}(\text{mem}(\text{eseq}(s, e1)), e2)$
 $\Rightarrow \text{move}(\text{eseq}(s, \text{mem}(e1)), e2)$
 $\Rightarrow \text{seq}(s, \text{move}(\text{mem}(e1), e2))$

4 Génération de code

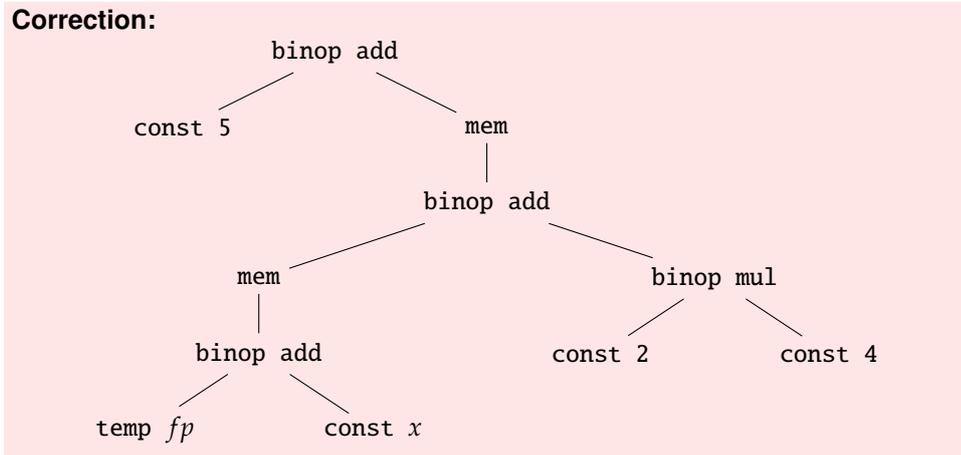
On considère le programme suivant, écrit en langage TREE (cf. [annexe A](#)) :

```
binop(add, const(5),
      mem(binop(add, mem(binop(add, temp(fp), const(x))),
                binop(mul, const(2), const(4)))))
```

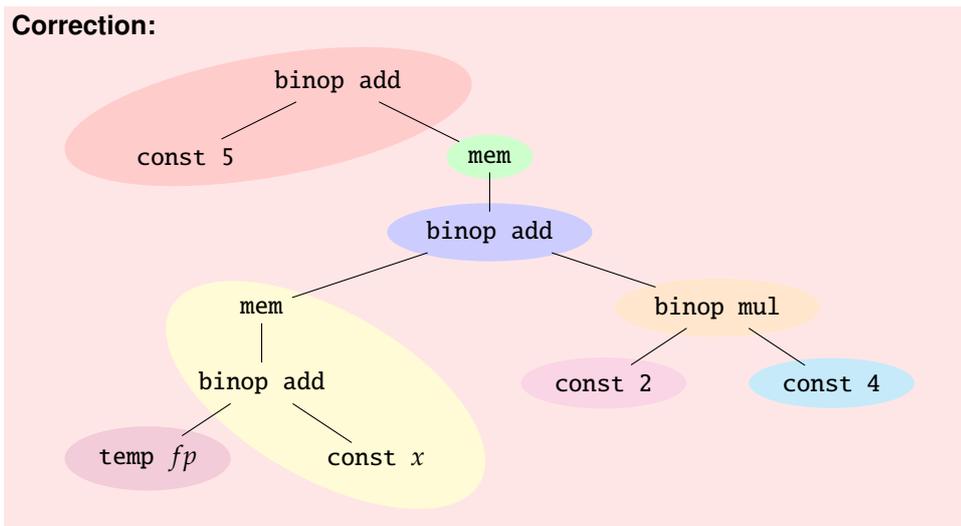
On souhaite traduire ce programme dans un jeu d'instructions machine simplifié décrit dans l'[annexe B](#). On utilisera pour cela l'algorithme du *Maximal Munch*, qui consiste à couvrir l'arbre

précédent en partant de sa racine et en utilisant à chaque fois la « tuile » (motif d'arbre) le plus volumineux (c'est-à-dire, couvrant le plus grand nombre de nœuds).

1. Représentez ce programme sous la forme d'un arbre (n'hésitez pas à en espacer les nœuds, car vous aurez à décorer cet arbre dans les questions qui suivent).

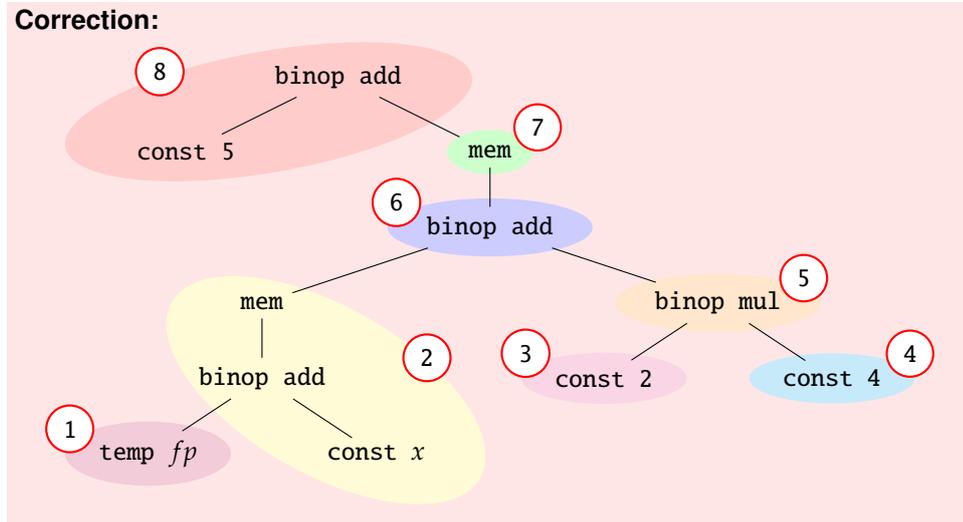


2. Procédez au « pavage » de l'arbre vous avez dessiné à la question précédente en faisant apparaître les tuiles utilisées (en entourant les nœuds concernés).



3. Numérotez les tuiles dans l'ordre dans lequel elle sont complètement évaluées par l'algorithme. On rappelle qu'une tuile ne peut être complètement évaluée que lorsque toutes ses tuiles filles ont été complètement évaluées.

Correction:



4. Écrivez la séquence d'instructions machine générées à l'issue de ce pavage. On fera figurer, pour chaque tuile traduite, le numéro qui lui a été attribué à la question précédente devant l'instruction correspondante. Le résultat attendu présentera donc un style similaire à celui de l'exemple suivant :

```

...
42  MUL   r8 ← r2 * r3
43  ADD   r9 ← r7 + r8
...
    
```

On tiendra compte des informations suivantes :

- La génération de code se fait *en sens inverse*, puisque qu'une instruction correspondant à une tuile ne peut être émise qu'après que les instructions correspondant à ses filles aient été générées.
- Certaines tuiles produisent des valeurs destinées à être stockées dans des registres (cf. **annexe B**). On nommera ces registres r_1, r_2, r_3 , etc. Par souci de simplicité, on considérera qu'on dispose d'un nombre illimité de registres.
- On rappelle que le registre r_0 contient toujours la valeur 0 (cf. **annexe B**).
- Une temporaire 'temp z ' sera traduite dans le jeu d'instructions machine comme un registre r_z . Un tuile recouvrant un nœud temp ne devra donc pas produire d'instruction : elle sera directement utilisée comme registre.
- La temporaire 'temp fp' désignant le frame pointer sera traduite par le registre spécial fp .

Correction:

```

2  LOAD   r1 ← M[fp + x]
3  ADDI   r2 ← r0 + 2
4  ADDI   r3 ← r0 + 4
5  MUL    r4 ← r2 * r3
6  ADD    r5 ← r1 + r4
7  LOAD   r6 ← M[r5 + 0]
8  ADDI   r7 ← r6 + 5
    
```

5. *Bonus.* « Décompilez » le programme TREE donné au début de cet exercice vers le langage Tiger.

Correction:

```
5 + x[2]
```

où x est une variable sur la pile, située à un décalage (*offset*) x du frame pointer (*fp*).

Annexes

A Définition du langage TREE

Le langage TREE est composé d'expressions et d'instructions (*statements*).

Expressions Les nœuds « expressions » (*exp*) représentent le calcul d'une valeur (potentiellement accompagné d'effets de bord) :

const(*i*) La constante entière *i*.

name(*n*) La constant symbolique *n*, autrement dit une référence à une étiquette (*label*).

temp(*t*) Une temporaire *t*.

binop(*o*, *e*₁, *e*₂) L'application de l'opérateur binaire *o* (qui peut être *add*, *sub*, *mul*, *div*, *and*, *or*, *xor*, *lshift*, *rshift*, *arshift*) aux opérandes *e*₁ et *e*₂, évalués dans cet ordre.

mem(*e*) Un accès au contenu d'un mot mémoire démarrant à l'adresse *e*.

call(*f*, *l*) L'appel de la routine désignée par l'expression *f* avec la liste d'argument effectifs *l* (*f* est évaluée avant les éléments de *l*, évalués de gauche à droite).

eseq(*s*, *e*) L'instruction *s* évaluée pour ses effets de bord, suivi par l'évaluation de l'expression *e* pour comme résultat.

Instructions Les nœuds « instructions » (*stm*) servent à produire des effets de bord et au contrôle de flot) :

move(*e*₁, *e*₂) Évalue *e*₁ et *e*₂ et place le contenu du dernier à l'emplacement désigné par le premier. En pratique, utilisé de deux façon différentes :

move(**temp**(*t*), *e*) Pour copier le résultat de l'évaluation de *e* dans la temporaire *t*.

move(**mem**(*e*₁), *e*₂) Pour stocker le résultat de l'évaluation de *e*₂ dans un mot mémoire démarrant à l'adresse *e*₁.

sxp(*e*) Évalue *e* et jette le résultat.

jump(*e*) Transfère le contrôle (effectue un saut) à l'adresse désignée par *e*.

cjump(*o*, *e*₁, *e*₂, *t*, *f*) Évalue *e*₁ et *e*₂ (dans cet ordre) pour produire les valeurs *a* et *b*, qui sont ensuite comparées avec l'opérateur *o*. Si ce résultat est vrai, effectue un saut à l'étiquette *t*, sinon effectue un saut à l'étiquette *f*. L'opérateur *o* peut valoir *eq* (égalité), *ne* (non égalité), *lt*, *gt*, *le*, *ge* (inégalités sur les entier signés) ou encore *ult*, *ugt*, *ule*, *uge* (inégalités sur les entier non signés).

seq(*s*) Exécute la liste d'instructions *s*, de la gauche vers la droite.

label(*n*) Définit le nom *n* comme adresse du code machine courant, autrement dit une étiquette.

Nous utilisons les notations du projet Tiger de l'EPITA (c'est-à-dire *sxp* au lieu de *exp* chez Andrew Appel, *seq* avec un nombre d'éléments arbitraires, etc.).

B Définition du jeu d'instructions machine simplifié

La tableau ci-dessous décrit les instructions d'un langage machine simplifié et les motifs d'arbres IR correspondant dans le langage TREE. À noter :

- Sur cette machine, le registre r_0 contient toujours la valeur 0.
- La notation $M[x]$ désigne le mot mémoire à l'adresse x .
- Les instructions situées au dessus de la double ligne du tableau produisent un résultat dans un registre. La toute première entrée n'est pas vraiment une instruction, mais exprime l'idée qu'un noeud `temp` est implémenté par un registre et donc qu'il peut « produire un résultat dans un registre » sans exécuter aucune instruction.
- Les instructions situées sous la double ligne ne produisent pas de résultat dans un registre et sont exécutées uniquement pour les effets de bord qu'elles produisent en mémoire.
- Pour des raisons de concision, on a représenté les noeuds TREE de type binop prenant un opérateur arithmétique (add, sub, mul, div) sous la forme +, *, -, /.

Nom	Effet	Motifs d'arbres
—	r_i	temp
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ / \quad \backslash \end{array}$
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ / \quad \backslash \end{array}$
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ / \quad \backslash \end{array}$
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ / \quad \backslash \end{array}$
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{c} + \\ / \quad \backslash \\ \text{const} \quad \text{const} \end{array}$
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ / \quad \backslash \\ \text{const} \end{array}$
MOVE	$r_i \leftarrow r_j$	$\begin{array}{c} \text{move} \\ / \quad \backslash \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{mem} \quad \text{mem} \\ \quad \\ + \quad + \\ / \quad \backslash \quad / \quad \backslash \\ \text{const} \quad \text{const} \quad \text{mem} \quad \text{mem} \\ \quad \\ \text{const} \end{array}$
STORE	$M[r_i + c] \leftarrow r_j$	$\begin{array}{c} \text{move} \quad \text{move} \\ / \quad \backslash \quad / \quad \backslash \\ \text{mem} \quad \text{mem} \quad \text{mem} \quad \text{mem} \\ \quad \quad \quad \\ + \quad + \quad \text{move} \quad \text{move} \\ / \quad \backslash \quad / \quad \backslash \\ \text{const} \quad \text{const} \quad \text{mem} \quad \text{mem} \\ \quad \\ \text{const} \end{array}$
MOVEM	$M[r_i] \leftarrow M[r_j]$	$\begin{array}{c} \text{move} \\ / \quad \backslash \\ \text{mem} \quad \text{mem} \\ \quad \end{array}$