CMP2 – Construction des compilateurs – S2

EPITA – Promo 2015
Documents (notes de cours, polycopiés, livres) autorisés
Machines (calculatrices, ordinateurs, tablettes, téléphones) interdites.

Juin 2013 (1h30)

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. Ce sujet contient 5 pages. Les pages 1–3 contiennent l'épreuve elle-même. Les pages 4–5 sont dédiées aux annexes.

1 Généralités

1. Rappelez quelles sont les quatre phases de la partie terminale (*back end*) d'un compilateur comme `tc`.
2. Qu'est-ce que le *frame pointer* ?
3. Citez un exemple de langage intermédiaire à base de pile.
4. Citez un exemple de langage intermédiaire à base de registres.
5. Comment se définit un bloc basique (*basic block*) ?
6. Quelle est la finalité de l'opération consistant à découper un programme en blocs basiques puis à les ré-assembler en une « trace » ?
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.

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`
2. `(1; 2; 3)`
3. `a + 5`
4. `b[i + 1]`
5. `p.y`
   où `p` est une variable de type `m`, défini comme suit : `type m = { x : int, y : int }`
6. `u.c.b`
   où `u` est une variable de type `n`, défini comme suit : `type n = { a : int, b : int, c : n }`
7. `if a then 42 else 51`

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 en écrivant la partie droite (sur votre copie, pas sur l'énoncé).

1. `sxp(eseq(s,e))` ⇒
2. `sxp(call(eseq(s, e1), const(4), e2))` ⇒
3. `move(mem(eseq(s, 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.
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 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.

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

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 noeuds « instructions » (`stm`) servent à produire des effets de bord et au contrôle de flot) :

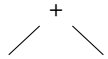
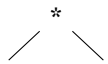
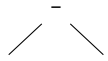
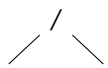
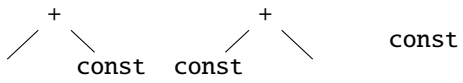
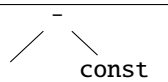
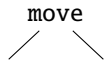
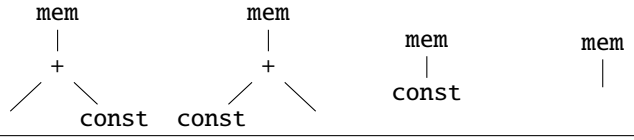
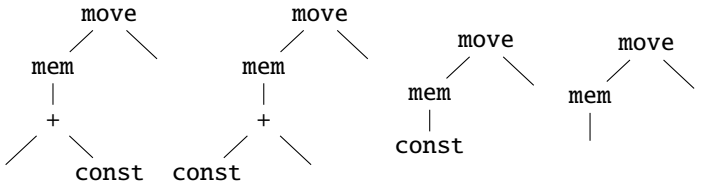
- **`move(`***e*1`,` *e*2`)`** Évalue e_1 et e_2 et place le contenu du dernier à l'emplacement désigné par le premier. En pratique, utilisé de deux façon différentes :
  - **`move(temp(`***t***`),` *e*`)`** Pour copier le résultat de l'évaluation de *e* dans la temporaire *t*.
  - **`move(mem(`**e_1**`),` 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 nœuds 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	
`MUL`	$r_i \leftarrow r_j$ `*` r_k	
`SUB`	$r_i \leftarrow r_j$ `-` r_k	
`DIV`	$r_i \leftarrow r_j$ `/` r_k	
`ADDI`	$r_i \leftarrow r_j$ `+` c	
`SUBI`	$r_i \leftarrow r_j$ `-` c	
`MOVE`	$r_i \leftarrow r_j$	
`LOAD`	$r_i \leftarrow$ `M[`r_j `+` c`]`	
`STORE`	`M[`r_i `+` c`]` $\leftarrow r_j$	
`MOVEM`	`M[`r_i`]` $\leftarrow$ `M[`r_j`]`