Correction du partiel CMP2

EPITA – Promotion 2012
Tous documents (notes de cours, polycopiés, livres) autorisés.
Calculatrices et ordinateurs interdits.

Juin 2010 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants, fautes de français y compris, le cas échéant.

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante sera souvent suffisante.

Une lecture préalable du sujet est recommandée.

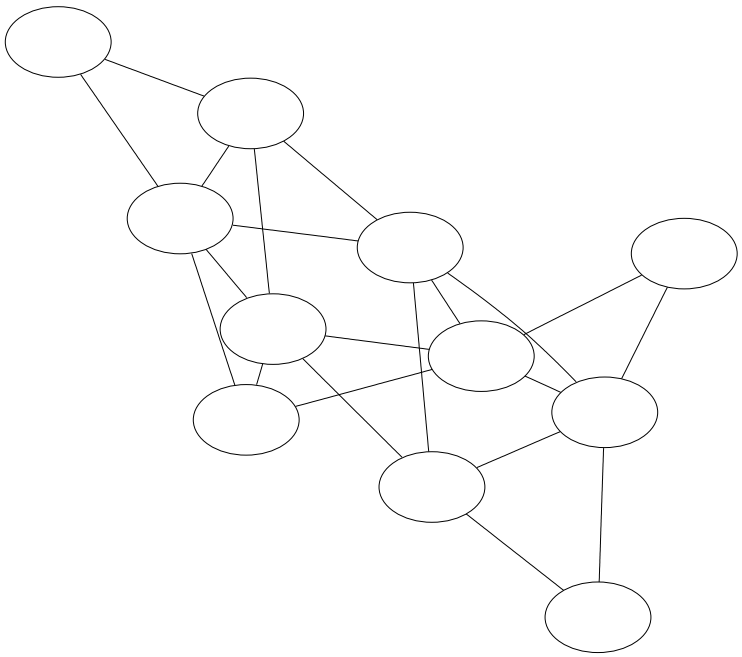
Écrivez votre nom en haut de la première page du sujet, et rendez-le avec votre copie.

1 Partie terminale : allocation de registres

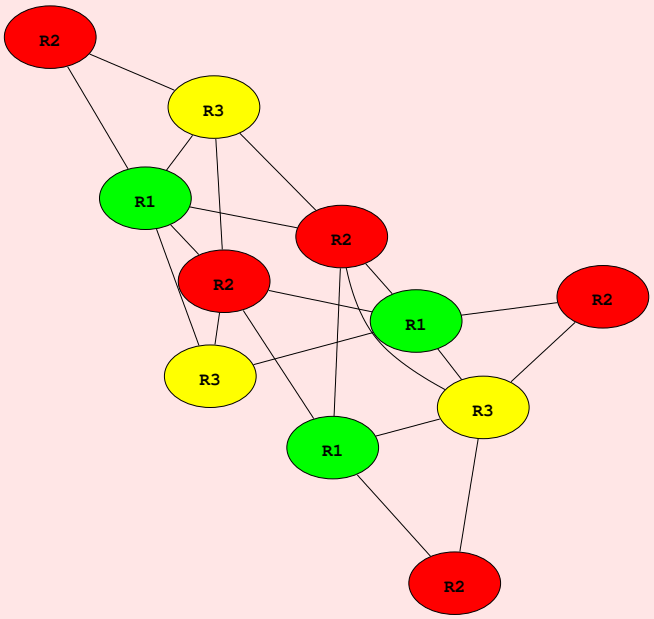
Colorer ce graphe d'exclusion mutuelle en 3 registres : R1, R2, R3. Rendre le sujet en ayant annoté chaque nœud d'un R1, R2, R3.

L'utilisation de stylos de couleurs différentes n'est pas requise, mais sera appréciée.

N'oubliez pas d'écrire votre nom en haut de la première page.



Correction:



2 Partie centrale

1. Définissez la partie centrale d'un compilateur.

   Correction: *Middle end* : tout ce qui ne dépend ni du langage source, ni du langage cible. En pratique, il est possible qu'il reste des "call-back" ou d'autres formes de paramétrisation en fonction du langage source et/ou cible (par exemple `$v0` dans `tc`), mais le code est essentiellement générique.

   Best-of:
   – La partie centrale d'un compilateur est la compilation.

2. Ci-après figure la représentation intermédiaire haut-niveau (HIR), en langage Tree, d'un programme Tiger.

```
1  label l0
2  move temp t2 temp fp
3  move temp fp temp sp
4  move
5   temp sp
6   binop sub temp sp const 4
7  move mem temp fp temp i0
8  move temp t0 temp i1
9  move
10  temp rv
11  eseq
12   seq
13    cjump le temp t0 const 1 name l1 name l2
14    label l1
15    move temp t1 const 1
16    jump name l3
17    label l2
18    move
19     temp t1
20     binop add
21      call
22       name l0
23       mem temp fp
24       binop sub temp t0 const 1
25      call end
26      call
27       name l0
28       mem temp fp
29       binop sub temp t0 const 2
30      call end
31    label l3
32   seq end
33   temp t1
34 move temp sp temp fp
35 move temp fp temp t2
36 label end
```

   Correction: Mes excuses : ce numéro n'était associé à aucune question.

3. Ce programme comporte trois parties. Indiquez quelles sont ces parties (en donnant les trois intervalles de lignes correspondant) et nommez-les.

Correction:
Lignes 2 à 8 Prologue

Lignes 9 à 33 Corps de la fonction

Lignes 34 à 35 Épilogue

Best-of:
– l. 34–36 : Travail « post-opératoire »

4. À quoi servent
   (a) les lignes 2 à 6 ?

   Correction: À sauvegarder les frame (`fp`) et stack (`sp`) pointers de la fonction précédente (appelante), et installer ceux de la fonction courante.

   (b) les lignes 7 et 8 ?

   Correction: Il s'agit du passage des arguments, copiés depuis les registres `i0` et `i1`.

   (c) les lignes 34 et 35 ?

   Correction: Ces dernières lignes forment l'épilogue, qui restaure les précédentes valeurs de `fp` et `sp`.

5. « Décompilez » le programme ci-dessus en un programme Tiger ou C équivalent.

   Correction:

```
function fib (n : int) : int =
  if n <= 1
  then 1
  else fib (n − 1) + fib (n − 2)

unsigned fib (unsigned n)
{
  return n <= 1 ? 1 : fib (n − 1) + fib (n − 2);
}
```

6. Que fait ce code ?

   Correction: Il s'agit d'une version récursive du calcul du $n^{\text{ème}}$ terme de la suite de Fibonacci :

$$fib : \left\{ \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & \left\{ \begin{array}{ll} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ fib(n-1) + fib(n-2) & \text{sinon} \end{array} \right. \end{array} \right.$$

   (Vous remarquerez que j'ai allègrement passé sous le tapis les cas où $n < 0$.)

   Best-of:
   – Il fait des additions...
   – Ce code fait des additions et des soustractions binaires.
   – Ce code fait beaucoup de travail pour pas grand chose ; une addition de deux nombres passé en paramètres.
   – Il s'agit d'un calcul de puissance : $\texttt{t0}^{\texttt{t1}}$ (ou, avec les notations du code Tiger ou C de la réponse précédente : $\texttt{x}^{\texttt{n}}$). (*Ça sent le copier-coller hâtif de la correction de l'examen de l'année précédente...*)

7. Le sous-arbre `call` des lignes 21 à 25 est un appel à la fonction désignée par l'étiquette `l0`, à laquelle sont passés deux arguments.
   (a) Quelle est cette fonction ?

   Correction: Il s'agit précisément de celle dont le code a été donné au début de cet exercice (qui commence par l'instruction `label l0`). Il s'agit donc d'un appel récursif.

   Best-of:
   – C'est la fonction '`--`' décrément unitaire.

   (b) À quoi sert le premier argument (`mem temp fp`) ?

   Correction: Cet argument « zéro » est le *static link* de la fonction parente de la fonction appelée. Rappelons que le static link sert à accéder aux variables non locales (échappées) à la fonction invoquée. (En l'occurrence, la fonction étudiée dans cet exercice ne fait usage d'aucune variable non locale ; le passage du static link est donc superflu ici, et une optimisation du compilateur pourrait le retirer pour cette fonction.)
   Attention aux confusions : beaucoup de copies indiquent que cet argument sert au mécanisme d'appel récursif – qui est en fait pris en charge par la gestion de la pile dans le prologue et l'épilogue ; ou encore à stocker l'adresse de retour (`ra`) – en pratique invisible dans un programme Tree, et gérée implicitement : c'est le runtime (HAVM) qui doit en assurer l'initialisation (lors d'un `call`), la sauvegarde/restauration éventuelle (dans le prologue/épilogue), et l'utilisation dans l'épilogue (`jump temp ra`).

8. Nous allons *canoniser* ce programme Tree pour le transformer en représentation intermédiaire bas-niveau (LIR). La première étape consiste en la *linéarisation* du programme.
   Commençons par de petits exercices avant de traiter le programme ci-dessus. Pour chacune des règles ci-dessous, écrivez la partie droite de la règle de réécriture (sur votre copie). On rappelle que par convention les termes commençant par '`s`' sont des instructions (*statements*), ceux commençant par '`e`' étant des expressions, et ceux commençant par '`t`' des temporaires. 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.).
   (a) **seq** (s1, **seq** (s2, s3), s4) =>

   Correction:

   seq (s1, **seq** (s2, s3), s4) => **seq** (s1, s2, s3, s4)

   (b) **mem** (**eseq** (s, e)) =>

   Correction:

   mem (**eseq** (s, e)) => **eseq** (s, **mem** (e))

   (c) **sxp** (**eseq** (s,e)) =>

   Correction:

   sxp (**eseq** (s,e)) => **seq** (s, **sxp** (e))

   (d) **sxp** (**call** (**eseq** (s, e1), e2)) =>

   Correction:

   sxp (**call** (**eseq** (s, e1), e2))
   => **sxp** (**eseq** (s, **call** (e1, e2)))
   => **seq** (s, **sxp** (**call** (e1, e2)))

(e) **move** (**temp** t, **call** (**eseq** (s, e1), e2)) =>

Correction: Si `temp t` commute avec `s` :

```
move (temp t, call (eseq (s, e1), e2))
 => move (temp t, eseq (s, call (e1, e2)))
 => seq (s, move (temp t, call (e1, e2)))
```

Si `temp t` ne commute pas avec `s` :

```
move (temp t, call (eseq (s, e1), e2))
 => move (temp t, eseq (s, call (e1, e2)))
 => seq (move (temp t0, temp t),
         s,
         move (temp t0, call (e1, e2)))
```

où `t0` est une temporaire fraîche.

9. Aidez-vous des règles que vous avez écrites à la question précédente ainsi que de celles de l'annexe A pour réécrire le programme en une version canonisée (sans `eseq` ni `seq`). Pour information, les lignes 1 à 8 ne sont pas affectées par cette réécriture, de même que les lignes 34 à 36.

Correction:

```
label l0
# Prologue.
move temp t2 temp fp
move temp fp temp sp
move
 temp sp
 binop sub temp sp const 4
move mem temp fp temp i0
move temp t0 temp i1
# Body.
cjump le temp t0 const 1 name l1 name l2
label l1
move temp t1 const 1
jump name l3
label l2
move
 temp t3
 call
  name l0
  mem temp fp
  binop sub temp t0 const 1
 call end
move
 temp t4
 call
  name l0
  mem temp fp
  binop sub temp t0 const 2
 call end
move
 temp t1
 binop add temp t3 temp t4
label l3
move temp rv temp t1
# Epilogue.
move temp sp temp fp
move temp fp temp t2
label end
```

10. Découpez le code de la question précédente en blocs basiques (commençant par une étiquette et terminant par un saut conditionnel ou inconditionnel).

Correction:

```
label l0
move temp t2 temp fp
move temp fp temp sp
move
  temp sp
  binop sub temp sp const 4
move mem temp fp temp i0
move temp t0 temp i1
cjump le temp t0 const 1 name l1 name l2
```

```
label l1
move temp t1 const 1
jump name l3
```

```
label l2
move
  temp t3
  call
    name l0
    mem temp fp
    binop sub temp t0 const 1
  call end
move
  temp t4
  call
    name l0
    mem temp fp
    binop sub temp t0 const 2
  call end
move
  temp t1
  binop add temp t3 temp t4
jump name l3
```

```
label l3
move temp rv temp t1
move temp sp temp fp
move temp fp temp t2
label end
```

Il ne faut pas oublier d'ajouter une instruction `jump name l3` à la fin du troisième bloc pour en faire un bloc basique valide.
Le dernier bloc, quant à lui, contient un saut implicite à l'adresse de retour (`jump temp ra`).

11. Ré-assemblez les blocs basiques de la question précédente en une trace valide (de telle sorte que les `cjumps` soient normalisés en sauts à une seule branche).

Correction: Voici une trace possible :

```
label l0
move temp t2 temp fp
move temp fp temp sp
move
 temp sp
 binop sub temp sp const 4
move mem temp fp temp i0
move temp t0 temp i1
cjump le temp t0 const 1 name l1 name l2

label l2
move
 temp t3
 call
  name l0
  mem temp fp
  binop sub temp t0 const 1
 call end
move
 temp t4
 call
  name l0
  mem temp fp
  binop sub temp t0 const 2
 call end
move
 temp t1
 binop add temp t3 temp t4
jump name l3

label l3
move temp rv temp t1
move temp sp temp fp
move temp fp temp t2
label end          # Implicit 'jump temp ra' here.

label l1
move temp t1 const 1
jump name l3
```

La trace ci-dessus commence classiquement avec le premier bloc. Celui-ci se termine par un `cjump` ; comme le bloc correspondant à sa branche « faux » (commençant par l'étiquette `l2`) est disponible, on peut le poser à la suite. Ce second bloc se termine par un saut en `l3`, dont le bloc n'est pas encore posé : on peut donc le placer à la suite. Enfin, on installe le dernier bloc restant (commençant par `label l1`) à la fin de la trace.

3 Pour terminer

1. Combien de nombres entiers différents peut-on représenter avec 256 bits ?

Correction: 2^{256}.

Best-of:
- 64 entiers
- 2^{64}
- $2^{256} - 1$
- $2^{256} + 1$
- 2^{256} entiers positifs + 2^{255} entiers négatifs
- 2^{256} entiers naturels
  $2^{256} - 1$ entiers relatifs
- Un nombre entier est sur 4 octets, donc 32 bits ; $\left(\frac{256}{32}\right)^{10} = 8^{10}$ possibilités différentes.
- $\left.\begin{array}{rcl} 256 \text{ bits} & = & 32 \text{ octets} \\ 1 \text{ entier} & = & 4 \text{ octets} \end{array}\right\}$ 32 octets = 8 entiers
- Après les calculs différents dont un qui revenait bizarrement à faire fibo [*calcul de la suite de Fibonacci*], je dirai 8192 mais mon résultat me semble petit.

2. Écrivez une fonction Tiger qui prend une chaîne de caractères en argument, renvoyant cette chaîne renversée. Par exemple, pour l'entrée « and », cette fonction renvoie « dna ». Rappel : les chaînes de caractères Tiger ne sont pas modifiables (dit autrement : non mutables). Vous trouverez un résumé de la bibliothèque standard Tiger dans l'annexe B.

Correction: Voici deux possibilités. Tout d'abord, une version itérative :

```
function reverse (s : string) : string =
  let
    var res := ""
  in
    for i := 0 to size (s) - 1 do
      res := concat (substring (s, i, 1), res);
    res
  end
```

Puis une version récursive :

```
function rec_reverse (s : string) : string =
  if size (s) <= 1 then
    s
  else
    concat (rec_reverse (substring (s, 1, size(s) - 1)),
            substring (s, 0, 1))
```

Best-of:
- Je préfère le TIGROU au TIGER... en plus l'interpréteur est léger (...si oublie PERL :))

4 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire. Les renseignements ci-dessous ne seront bien entendu pas utilisés pour noter votre copie. Ils ne sont pas anonymes, car nous souhaitons pouvoir confronter réponses et notes. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Sauf indication contraire, vous pouvez cocher plusieurs réponses par question. Répondez sur la feuille de QCM qui vous est remise. N'y passez pas plus de dix minutes.

Cette épreuve

1. Sans compter le temps mis pour remplir ce questionnaire, combien de temps ce partiel vous a-t-il demandé (si vous avez terminé dans les temps), ou combien vous aurait-il demandé (si vous aviez eu un peu plus de temps pour terminer) ?
   A Moins de 30 minutes.
   B Entre 30 et 60 minutes.
   C Entre 60 et 90 minutes.
   D Entre 90 et 120 minutes (estimation).
   E Plus de 120 minutes (estimation).
2. Ce partiel vous a paru
   A Trop difficile.
   B Assez difficile.
   C D'une difficulté normale.
   D Assez facile.
   E Trop facile.

Le cours

3. Quelle a été votre implication dans les cours CMP1 et CMP2 ?
   A Rien.
   B Bachotage récent.
   C Relu les notes entre chaque cours.
   D Fait les annales.
   E Lu d'autres sources.
4. Ce cours
   A Est incompréhensible et j'ai rapidement abandonné.
   B Est difficile à suivre mais j'essaie.
   C Est facile à suivre une fois qu'on a compris le truc.
   D Est trop élémentaire.
5. Ce cours
   A Ne m'a donné aucune satisfaction.
   B N'a aucun intérêt dans ma formation.
   C Est une agréable curiosité.
   D Est nécessaire mais pas intéressant.
   E Je le recommande.

6. La charge générale du cours en sus de la présence en amphi (relecture de notes, compréhension, recherches supplémentaires, etc.) est
   A Telle que je n'ai pas pu suivre du tout.
   B Lourde (plusieurs heures par semaine).
   C Supportable (environ une heure de travail par semaine).
   D Légère (quelques minutes par semaine).

Les formateurs

7. L'enseignant
   A N'est pas pédagogue.
   B Parle à des étudiants qui sont au dessus de mon niveau.
   C Me parle.
   D Se répète vraiment trop.
   E Se contente de trop simple et devrait pousser le niveau vers le haut.
8. Les assistants
   A Ne sont pas pédagogues.
   B Parlent à des étudiants qui sont au dessus de mon niveau.
   C M'ont aidé à avancer dans le projet.
   D Ont résolu certains de mes gros problèmes, mais ne m'ont pas expliqué comment ils avaient fait.
   E Pourraient viser plus haut et enseigner des notions supplémentaires.

Le projet Tiger

9. Vous avez contribué au développement du compilateur de votre groupe (une seule réponse attendue) :
   A Presque jamais.
   B Moins que les autres.
   C Équitablement avec vos pairs.
   D Plus que les autres.
   E Pratiquement seul.
10. La charge générale du projet Tiger est
    A Telle que je n'ai pas pu suivre du tout.
    B Lourde (plusieurs jours de travail par semaine).
    C Supportable (plusieurs heures de travail par semaine).
    D Légère (une ou deux heures par semaine).
    E J'ai été dispensé du projet.
11. Y a-t-il de la triche dans le projet Tiger ? (Une seule réponse attendue.)
    A Pas à votre connaissance.
    B Vous connaissez un ou deux groupes concernés.
    C Quelques groupes.
    D Dans la plupart des groupes.
    E Dans tous les groupes.

Questions 12-18 Le projet Tiger vous a-t-il bien formé aux sujets suivants ? Répondre selon la grille qui suit. (Une seule réponse attendue par question.)

A Pas du tout
B Trop peu
C Correctement
D Bien
E Très bien

12. Formation au C++.
13. Formation à la modélisation orientée objet et aux *design patterns*.
14. Formation à l'anglais technique.
15. Formation à la compréhension du fonctionnement des ordinateurs.
16. Formation à la compréhension du fonctionnement des langages de programmation.
17. Formation au travail collaboratif.
18. Formation aux outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 19-34 Comment furent les étapes du projet ? Répondre selon la grille suivante. (Une seule réponse attendue par question ; ne pas répondre pour les étapes que vous n'avez pas faites.)

A Trop facile.
B Facile.
C Nickel.
D Difficile.
E Trop difficile.

19. Rush `.tig` : mini-projet en Tiger (Bistromatig).
20. `TC-0`, Scanner & Parser.
21. `TC-1`, Scanner & Parser, Tâches, Autotools.
22. `TC-2`, Construction de l'AST et pretty-printer.
23. `TC-3`, Liaison des noms et renommage.
24. `TC-4`, Typage et calcul des échappements.
25. `TC-5`, Traduction vers représentation intermédiaire.
26. `TC-6`, Simplification de la représentation intermédiaire.
27. `TC-7`, Sélection des instructions.
28. `TC-8`, Analyse du flot de contrôle.
29. `TC-9`, Allocation de registres.
30. Option `TC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
31. Option `TC-I`, Mise en ligne du corps des fonctions.
32. Option `TC-B`, Vérification dynamique des bornes de tableaux.
33. Option `TC-A`, Surcharge des fonctions.
34. Option `TC-O`, Désucrage des constructions objets (transformation Tiger → Panther). .

A Quelques règles de réécriture pour la canonisation

Dans les règles suivantes, les termes commençant par 's' désignent des instructions (*statements*), ceux commençant par 'e' étant des expressions, et ceux commençant par 't' des temporaires.

```
eseq (s1, eseq (s2, e))      => eseq (seq (s1, s2), e)

binop (op, eseq (s, e1), e2) => eseq (s, binop (op, e1, e2))

jump (eseq(s, e1))           => seq (s, jump (e1))

cjump (op, eseq(s, e1), e2, l1, l2)
 => seq (s, cjump (op, e1, e2, l1, l2))

seq (ss1, seq (ss2), ss3)    => seq (ss1, ss2, ss3)
```

Si e1 et s commutent :

```
binop (op, e1, eseq (s, e2))
 => eseq(s, binop (op, e1, e2))

cjump (op, e1, eseq (s, e2), l1, l2)
 => seq (s, cjump (op, e1, e2, l1, l2))
```

Si e1 et s ne commutent pas :

```
binop (op, e1, eseq (s, e2))
 => eseq (seq (move (temp t, e1), s),
          binop (op, temp t, e2))

cjump (op, e1, eseq (s, e2), l1, l2)
 => seq (seq (move (temp t, e1), s),
         cjump (op, temp t, e2, l1, l2))
```

(où t est une temporaire fraîche).

B Bibliothèque standard du langage Tiger

Le listing ci-dessous contient les déclarations (annotées) du prélude actuel du compilateur Tiger (`prelude.tih`).

```
/* Print STRING on the standard output. */
primitive print (string: string)
/* Note: this is an EPITA extension. Same as print(), but the output
   is written to the standard error. */
primitive print_err (string: string)
/* Note: this is an EPITA extension. Output INT in its decimal
   canonical form (equivalent to "%d" for printf()). */
primitive print_int (int: int)
/* Flush the output buffer. */
primitive flush ()
/* Read a character on input. Return an empty string on an end of
   file. */
primitive getchar () : string

/* Return the ASCII code of the first character in STRING and -1 if
   the given string is empty. */
primitive ord (string: string) : int
/* Return the one character long string containing the character which
   code is CODE. If CODE does not belong to the range [0..255], raise
   a runtime error: "chr: character out of range." */
primitive chr (code: int) : string
/* Return the size in characters of the STRING. */
primitive size (string: string) : int

/* Note: this is an EPITA extension. Return 1 if the strings A and B
   are equal, 0 otherwise. Often faster than strcmp() to test string
   equality. */
primitive streq (s1: string, s2: string) : int
/* Note: this is an EPITA extension. Compare the strings A and B:
   return -1 if A < B, 0 if equal, and 1 otherwise. */
primitive strcmp (s1: string, s2: string) : int
/* Return a string composed of the characters of STRING starting at
   the FIRST character (0 being the origin), and composed of LENGTH
   characters (i.e., up to and including the character
   FIRST + LENGTH). */
primitive substring (string: string, start: int, length: int) : string
/* Concatenate FIRST and SECOND. */
primitive concat (first : string, second: string) : string

/* Return 1 if BOOLEAN = 1, else return 0. */
primitive not (boolean : int) : int

/* Exit the program with exit code STATUS. */
primitive exit (status: int)
```