Correction/Commentaires du Partiel de Compilation

Le sujet a été écrit par Akim Demaille. Les questions 2 (généricité) et 4 (mise en ligne) sont fortement inspirées par les chapitres correspondant dans le livre d'A. Appel, « Modern Compiler Implementation ». La correction a été rédigée par Pierre-Yves Strub, qui a également corrigé les copies. C'est à la demande d'Akim Demaille qu'il a inclus ci-dessous des commentaires sur les réponses les plus amusantes et/ou effrayantes.
`<brains on>`

1 Bison

Une `exp` est de type `exp_t`.

1. Qu'exprime la grammaire suivante ?
   ```
   exps: exp | exp ';' exps
   ```

   Correction: Une suite de `exp` séparés par des "`;`". Cette dernière est au minimum composée d'une `exp`. Il n'y a pas de limite quant au nombre maximum de `exp`[a].
   Bien voir que le "`;`" est ici un séparateur, pas un terminateur. (Différence du "`;`" entre `Pascal` et `C`)

   [a] J'ai vu plusieurs fois le `|` considéré comme un caractère, non comme un séparateur de règles. *Snirf.*

2. Que pourrait-on lui reprocher et comment la réécriez-vous ?

   Correction: On ne demandait pas ici de commenter le langage décrit par la grammaire, mais bien la grammaire en elle-même ! Toutes les réponses du genre *On ne peut pas avoir une suite vide* ou *le `;` est un séparateur, non un terminateur* étaient totalement à côté de la plaque[a].
   Le problème ici est que cette grammaire est récursive à droite. L'analyse `LR` s'en trouve fortement perturbée. Sans m'étendre, une récursion à gauche éviterait de voir la pile des *tokens* grossir en faisant des *reduces* au fur et à mesure plutôt que des *shifts* et une floppée de *reduces* à la fin. Pour plus d'informations, voir le cours...
   On préférera donc la grammaire suivante :
   ```
   exps: exp | exps ';' exp
   ```
   Dans la suite, je note `SEMI` le *token* "`;`".

   [a] Je ne préfère pas m'étendre sur le *il serait plus clair de la réécrire sur deux lignes.*

3. Complétez votre proposition en incluant les actions et la reprise sur erreur.

Correction: Aucune réelle difficulté. Il suffit de connaitre un minimum de `Bison`. Juste se souvenir de l'existence du *token* `error` et éviter de le placer n'importe où. (E.g., faire une règle principale se réduisant sur `exps` ou `error` me semble un peu inutile : vous faites comment pour reprendre une fois que l'erreur vous a totalement fait remonter ?)

```
exps    : exp              { $$ = new list < exp_t* >;
                             $$->push_back ($1); }
        | exps  SEMI exp   { $$ = $1->push_back ($3); }
        | error SEMI exp   { $$ = new list < exp_t* >;
                             $$->push_back ($3); }
        ;
```

4. Ecrivez une grammaire pour "une liste d'`exp` (0 ou plusieurs) séparés par des `;`", avec les actions et la reprise sur erreur.

Correction: Il suffit de créer une nouvelle règle acceptant une suite vide ou tout bonnement une suite telle que décrite à la question précédente. Vous avez ce code dans le parseur de `Tiger`[a].
Je n'ai que trop vu de copies rajoutant une règle vide à la règle `exps`. Il faut bien voir que dans ce cas, vous autorisez des suites telles que

```
                          ; exp ; exp ; exp.
```

Voici la solution :

```
exps    : /* empty */      { $$ = new list < exp_t* > }
        | exps.1
        ;

exps.1  : exp              { $$ = new list < exp_t* >;
                             $$->push_back ($1) }
        | exps.1 SEMI exp  { $$->push_back ($3) }
        | error  SEMI exp  { $$ = new list < exp_t* >;
                             $$->push_back ($3) }
        ;
```

[a] J'ai d'ailleurs vu beaucoup de copies répondre n'importe quoi à la question précédente et parfaitement à cette dernière. Il n'est pas interdit de réfléchir et de ne pas faire de la copie brutale de parties de livres dans un partiel.

2 Typage

Soient les fonctions suivantes :

```
function int_id (a: int) : int = a
function string_id (a: string) : string = a
```

1. Qu'est-ce qui est gênant dans le pouvoir d'expression de `Tiger`.

Correction: La question était certes un peu floue. Mais ce n'est pas une raison pour répondre n'importe quoi :

- **Non**, le fait que `Tiger` ait une *saveur* fonctionnelle n'est pas dérangeant.
- **Non**, l'absence du mot clef `return`, et des `{}` n'est pas handicapante pour l'expressivité.
- **Non**, le fait de ne pas pouvoir faire de *casts* entre `string` et `int`[a] n'est pas une barrière empêchant la création de *vrais* programmes.
- **Non**, le fait que `Tiger` soit fortement typé n'est pas morose. Et d'ailleurs,
- **Non**, `Tiger` n'est pas faiblement typé.
- **Non**, il n'est pas embêtant que les paramètres des deux fonctions aient le même nom[b]. Et, **Non**, il ne faut pas produire d'erreurs quand un même nom est utilisé pour paramètres, fonctions et types[c].

Ici, on a deux fonctions ayant la même sémantique (Renvoie de leur seul argument). On voudrait n'avoir à l'écrire qu'une seule fois... pas pour tous les types.

[a] Au fait, j'aimerais savoir le sens que vous donnez à cela.
[b] Cela vous dérange en `C` ?
[c] Je vous renvoie à votre cours concernant les espaces de noms.

2. Il y a deux possibilités évidentes pour pallier ce défaut, l'une que l'on appellera *implicite* et l'autre *explicite*. Chacune de ces possibilités donnent deux extensions de `Tiger`. Réécrivez ce code pour chacune de ces deux saveurs de `Tiger`. Il ne vous est *pas* demandé du `Tiger` traditionnel, mais bien d'étendre `Tiger` de deux façons différentes.

Correction:
Solution explicite : on utilise la généricité statique. E.g. on voudrait pouvoir écrire :

```
let
  function < T > id (x: T) : T = x
in
  /* ... */
    id < int > (51)
  /* ... */
    id < string > (''foobar'')
  /* ... */
end
```

Solution implicite : on utilise l'inférence de type (A la `OCaml`). E.g. :

```
let
  function id (x) = x
in
  /* ... */
    id (51)
  /* ... */
    id (''foobar'')
  /* ... */
end
```

J'ai vu des copies proposant la surcharge de fonction. Cela ne résoud en rien le problème : nous avons toujours à réécrire la fonction pour tous les types... même si après nous aurions le même nom pour `id_int` et `id_string`[a].

[a] Et pour le plaisir des yeux, non, faire une *grammaire prenant en compte l'indentation* n'est pas une bonne solution... la couleur aurait été un meilleur choix. De plus, en `C`, `void` ou `void*` ne permet en rien de programmer générique (Surtout `void` !)... donc encore moins en `Tiger`.

3 Un gros vide

Pour passer à la page (0 point).

4 Mise en ligne (Inlining)

La *mise en ligne* consiste à replacer un appel de fonction par le corps de la fonction en elle-même. Par exemple la mise en ligne de

```
function add (a: int, b: int) : int = a + b
```

dans `add (20, 22)` donne bien entendu (Et encore...) `20 + 22`.

Sans chercher à aller jusqu'au bout, étudions les difficultés pour l'implémentation de la mise en ligne dans *notre* implémentation de `tc`.

1. En considérant e.g.

   ```
   function is_smaller (a: string, b: string) : int = a < b,
   ```

   à quelle étape de la compilation suggérez-vous d'effectuer la mise en ligne ?

   Correction: Reprenons les phases principales :

   avant/pendant/après l'analyse lexicale et le scanneur : la mise en ligne n'est pas l'insertion brutale de code (Au sens lignes de code), un compilateur n'est pas un préprocesseur... et non, malheureusement non, `Tiger` ne possède pas une phase de *preprocessing*. De plus, comment espériez-vous qu'un outils ne s'intéressant pas au sens de ce que vous écrivez puisse avoir la moindre heuristique quant à effectuer la mise en ligne de vos fonctions. Il a été proposer l'introduction d'un nouveau mot clef : `inline`. Malheureusement, dans ce cas, c'est à `#define` que vous pensiez[a]. On se ramène au problème sus-cité. De plus, imaginez la gestion des messages d'erreur.

   après la construction de l'arbre de syntaxe abstraite : certes, vous possédez des informations en plus... mais toujours pas assez. Prenons notre exemple : vous mettez en ligne (Au sens travail sur l'arbre de syntaxe abstraite) dans l'arbre la fonction `is_smaller` lors de l'appel à

   ```
   is_smaller (``foobar'', 51).
   ```

   Elle va être belle l'analyse sémantique. Les appels de fonctions ont disparu pour son code. Pour maintenir des messages d'erreurs correctes, je n'explique pas le travail (Un peu le coup du `#line` du `C`). On veut indiquer au programmeur que l'appel de fonction est incorrect... pas qu'il ne peut pas comparer un `int` avec un `string`.

   après l'analyse sémantique (avant le passage à l'`IR`) : nous venons de vérifier tout le code (Aussi bien au niveau lexical qu'au niveau sémantique. Nous n'avons plus rien à dire au programmeur). Nous avons notre arbre. Rien ne nous empêche de faire la mise en ligne.

   après le passage à l'`IR`, assembleur et code machine : trop tard. Vous venez de perdre trop d'informations. Le travail va être vraiment trop dur (Rien qu'au niveau allocation des registres). Et plus on avance vers le code machine, plus cela devient impossible.

   En conclusion, nous plaçons notre phase de mise en ligne entre *l'analyse sémantique* et *le passage à l'`IR`*.

   [a] Oui, le *type-checking* ne se passe avant l'analyse lexicale.

2. La mise ne ligne naïve de

```
function double (a: int) : int = a + a
```

peut conduire à des résultats désagréable. Comme quoi ?

Correction: Beaucoup de copies parlent d'un problème de priorité (Ceci est en partie dû au fait que certains ont confondu *mise en ligne* et *preprocessing*). Mais nous travaillons sur un arbre : grossièrement, la mise en ligne va consister à changer le noeud effectuant l'appel de fonction par le code de la fonction[a]. Nous changeons en rien ici l'ordre d'évaluation[b].
Formellement, au sein de la portée de

```
function f (a_1, ..., a_n) = B
```

l'appel

```
f (i_1, ..., i_n)
```

devient

```
B [a_1 -> i_1, ..., a_n -> i_n]
```

I.e. que je remplace l'appel à `f` par son code en prenant soin de remplacer les références aux paramètres formels de `f` par les arguments passés à `f`.
Donnons-nous une fonction `side_effect`. Que se passe-t-il quand j'évalue

```
double (side_effect ())
```

?

Sans mise en ligne, `side_effect` est évalué (appel de fonction) et son résultat est passé à `double` (copie). Avec mise en ligne, l'appel à `double` devient :

```
side_effect () + side_effect ()
```

Outre le fait que cela puisse ralentir l'exécution du programme (Deux appels à `side_effect` au lieu d'un), il faut remarquer que `Tiger` n'est pas un langage pur. Une fonction peut faire des effets de bords et peut donc renvoyer des résultats différents à chaque appel[c]. Une telle mise en ligne peut dont amener à produire du code incorrect.

[a] Je viens ici de passer à la trappe le probleme de passage des arguments.
[b] Pour ceux qui ne voient pas, faire un dessin des arbres avant et après la mise en ligne.
[c] J'ai souvent vu des `i++` ou des fonctions modifiant leurs arguments : en `Tiger`, les passages se font par valeur.

3. Comment faut-il faire alors ?

Correction: Quand les arguments de la fonction à mettre en ligne ne sont pas de simples variables, il faut utiliser des variables temporaires. Formellement, au sein de la portée de

```
function f (a_1, ..., a_n),
```

l'appel

```
f (E1, ..., En)
```

devient

```
let var i_1 := E1 ... var i_n := En
in
  B[a_1 -> i_1, ..., a_n -> i_n]
end
```

où `i_1...i_n` sont des noms de variables inutilisés précédemment.

4. En vous basant sur les question précédentes, quelle différence faites-vous entre fonctions mises en ligne et macros en `C` ?

Correction: Les macros du C sont juste des opérations sur le code vu en tant que texte. Elles interviennent lors de la phase de *preprocessing*, avant toute analyse du code (Lexicale et sémantique). Ainsi, elles peuvent produire des effets de bords indésirables, des modifications de priorités. On ne peut pas avoir de macros récursives sans obtenir une erreur : une macro est obligatoirement étendue. Les paramètres d'une macro sont juste des chaînes de caractères : aucune notion de type, de passage par valeur/référence... Une macro ne représente pas forcément l'expansion d'une fonction : on peut s'en servir pour produire des structures, voire des bouts de structures, des bouts d'expressions. Prenons par exemple le cas de l'exercice 2. On peut se servir d'une macro afin de générer des fonctions paramétrées par un type en C[a].

```
#define ID_GEN(Type, Name)                                  \
        static Type gen__##Name (Type x);                   \
                                                            \
        static Type                                         \
        gen__##Name (Type x) { return x; }

ID_GEN (int, int);
ID_GEN (char *, string);
```

La mise en ligne est donc totalement différente. Elle concerne l'expansion du code (Une fois analysé et vue de manière structurée -un arbre e.g.-) d'une fonction. De plus, elle est effectuée par un compilateur. L'expansion est alors sûre (Pas d'effets de bords, de multiples évaluations ou modification de priorité). On n'est pas obligé de demander explicitement la mise en ligne d'une fonction pour qu'elle soit faite : on peut très bien penser à des heuristiques décidant quand faire ou ne pas faire une telle opération. Inversement, si le compilateur n'arrive pas à mettre une fonction en ligne, il peut décider de laisser le code tel quel (appel de fonction), et de finir la compilation : la mise en ligne ne change pas la sémantique d'un programme.

[a] La seule vraie généricité statique du C.

5. Comment faire dans le cas de

```
let fact (n : int) : int =
  if n = 1 then
    0
  else
    n * fact (n - 1)
in
  fact (42)
end
```

Qu'en déduisez-vous ?

Correction: Nous sommes ici face à une fonction récursive. Si nous mettons en ligne `fact`, doit-on aussi le faire au sein de `fact` et si oui, quand nous arrêter ? Dans le cas général, il n'est pas possible de savoir combien il faudra d'appels récursifs à `fact` seront. En conséquences, si nous mettons en ligne `fact` dans `fact` sans connaitre la valeur de l'argument, il nous faudrait une borne arbitraire au nombre de mises en ligne successives. 5 ?.. 10 ?.. pourquoi pas 100.
Choisissons une grande valeur. Si ensuite l'argument a toujours une faible valeur, nous nous retrouverons avec du code jamais exécuter. Et une forte valeur implique une explosion de la taille du code.
Alors choisissons une faible valeur. Oui mais si cette fonction est en générale utilisée avec de fortes valeurs[a], a quoi vont nous servir les quelques mises en lignes sur les 3472936 appels suivants.
Le plus simple semble réellement de ne pas mettre en ligne une fonction récursive en son sein.
Plus généralement (récursion plus complexe), une fonction récursive est séparée en deux parties :
un prélude appelé de l'extérieur.

une partie récursive appelé de l'intérieur (par elle même et le prélude).
Formellement, une fonction définie par :

```
function f (a_1, ..., a_n) = B
```

devient

```
function f (a'1, ..., a'n) =
  let function f'(a_1, ..., a_n) =
    B [f -> f']
  in
    f' (a'1, ..., a'n)
  end
```

`NB:` pourquoi ne pas imaginer un compilateur vraiment intelligent qui face a une recursion terminale avec un paramètre constant essaie de calculer le résultat à la compilation ?

[a] Ce qui n'est pas le cas de `fact`

6. Considérons

```
1. let var delta := 1
2.     function advance (x: int) : int = x + delta
3.     function start_from (delta: int) : int = advance (0) + delta
4. in
5.     start_from (delta)
6. end
```

   (a) A quel problème s'expose-t-on ?

Correction: Le paramètre formel `delta` de `start_from` (ligne 4) bloque la portée de la variable `delta` (ligne 1) au sein de `start_from`. Si bien que le `delta` de la ligne 5 fait référence au paramètre formel, pas à la variable.
Supposons maintenant que nous métions en ligne le code de `advance` dans `start_from`. Nous obtenons :

```
...
4. function start_from (delta: int) : int =
     (x + delta) + delta
...
```

Désormais, le premier `delta` référencie le paramètre formel de `start_from`, pas la variable.

(b) Sauriez-vous réécrire cet exemple pour blinder contre ça ?

Correction: Nous pouvons renommer le paramètre formel de `start_from`. On obient alors :

```
let var delta := 1
    function advance (x : int) : int = x + delta
    function start_from (start_from_delta : int) : int =
      advance (0) + start_from_delta
in
    start_from (delta)
end
```

(c) Quelle solution proposez-vous pour que l'implémentation de la mise en ligne ne soit pas galère à cause de ça ?

Correction: La meilleure solution (aussi bien au niveau du raisonnement que de la rapidité en moyen cas) afin d'éviter ce genre de conflit est de faire une passe du compilateur renommant toutes les variables du programme afin que jamais deux variables ne portent le même nom.

(d) Comment implémenteriez-vous ceci dans `tc` ? Evidemment je ne veux pas de code ici, mais décrivez correctement votre suggestion.

Correction: Un visiteur, qui passerait après le type checker, mais avant le calcul des échappements.

`<brains off>`