Correction du Partiel CMP2

EPITA – Promo 2010
Tous documents (notes de cours, polycopiés, livres) autorisés.
Tous dispositifs de calcul automatisé interdits[*]

Juin 2008 (1h30)

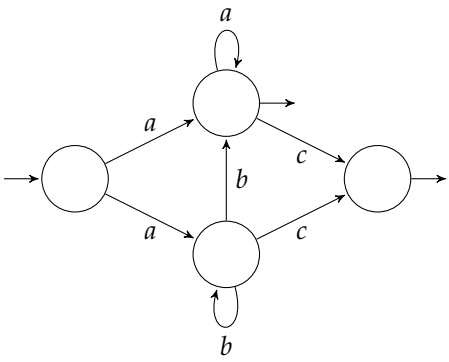
Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants (les fautes de langue sont d'origine).

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.

1 Incontournables

Il n'est pas admissible d'échouer sur une des questions suivantes : **chacune induit une pénalité sur la note finale**. Répondez sur les feuilles de QCM qui vous sont remises (n'oubliez pas d'y inscrire votre nom ou login).

Considérez l'automate ci-dessous, et répondez aux questions qui suivent.



1. Cet automate est déterministe. A. Vrai/B. Faux ?

   Correction: B. Faux. Depuis l'état initial, on peut accéder à deux états différents en utilisant des transitions étiquetées par la même lettre, a.

2. Un ordinateur peut reconnaître un mot du langage décrit par cet automate directement (c'est-à-dire, en simulant une exécution de cet automate sans modification ou ajustement préalable de celui-ci). A. Vrai/B. Faux ?

   Correction: A. Vrai. Oui : on peut simuler l'exécution (l'évaluation) d'un automate fini sur un ordinateur, qu'il soit déterministe ou pas. Dans le second cas, on peut suivre tous les chemins possibles dans l'automate, ou bien utiliser du retour sur trace (*backtracking*).

[*]Calculatrice, téléphone portable, ordinateur, *mentat*, etc.

3. Cet automate reconnaît le mot "*a*". A. Vrai/B. Faux ?

   Correction: A. Vrai. Il suffit de suivre la transition du haut en partant de l'état initial, puis de sortir via l'état atteint, qui est accepteur.

4. Le langage reconnu par cet automate est dans la classe des langages hors-contexte. A. Vrai/B. Faux ?

   Correction: A. Vrai. Bien entendu, puisque la classe des langages hors-contexte (type 2 dans la hiérarchie de Chomsky) inclut la classe des langages réguliers (type 3).

2 Étendons Bison

1. Voici une grammaire simplifiée de la partie "grammaire" que l'on peut trouver dans un fichier d'entrée de Yacc (située entre les deux séparateurs `%%` du fichier), sans les actions (pour ne compliquer l'exercice outre mesure) :

```
grammar −> rules
grammar −> grammar rules

rules −> ID ":" rhses

rhses −> rhs
rhses −> rhses "|" rhs
rhses −> rhses ";"

rhs −> rhs ID      (Typo dans le sujet initial, qui comportait
                    une règle erronée « rhs −> rhs SYMBOL ».)
rhs −> eps
```

   Cette grammaire est-elle LR(1) ? Est-elle LR(k) ? Si oui, pour quel k ? Justifiez votre réponse.

   Correction: Tout d'abord, mes excuses pour une erreur de frappe dans le sujet : il fallait lire *ID* à la place de *SYMBOL* ici. Lorsque la réponse argumentait sensément en s'appuyant sur le sujet (erroné), je l'ai considérée juste, bien entendu.

   Cette grammaire n'est pas LR(1), car il n'est pas possible de construire un automate LR(1) reconnaissant le langage qu'elle engendre. Réécrire tout l'automate est fastidieux, mais on peut montrer "avec les mains" qu'il contient un état exhibant un conflit shift/reduce :

rhses	→	**rhs** .	$
rhses	→	**rhs** .	*ID*
rhses	→	**rhs** .	|
rhses	→	**rhs** .	;
rhs	→	**rhs** . *ID*	$
rhs	→	**rhs** . *ID*	*ID*
rhs	→	**rhs** . *ID*	|
rhs	→	**rhs** . *ID*	;

   Lorsque l'on est dans cet état, et derrière un *ID*, on ne sait pas s'il faut réduire la règle **rhses** → **rhs** ou bien faire passer le token *ID* sur la pile.

Correction: En revanche, si l'on construit un automate LR(2) relatif à cette grammaire, on va lever cette ambiguïté vis-à-vis de *ID*. En LR(2), l'état précédent devient :

rhses	→	**rhs** .	$
rhses	→	**rhs** .	| $
rhses	→	**rhs** .	| *ID*
rhses	→	**rhs** .	; $
rhses	→	**rhs** .	; *ID*
rhses	→	**rhs** .	*ID* :
rhs	→	**rhs** . *ID*	$
rhs	→	**rhs** . *ID*	| $
rhs	→	**rhs** . *ID*	| *ID*
rhs	→	**rhs** . *ID*	; $
rhs	→	**rhs** . *ID*	; *ID*
rhs	→	**rhs** . *ID*	*ID* :
rhs	→	**rhs** . *ID*	*ID* $
rhs	→	**rhs** . *ID*	*ID* |
rhs	→	**rhs** . *ID*	*ID* ;
rhs	→	**rhs** . *ID*	*ID ID*

On voit désormais que l'on réduit la règle **rhses** → **rhs** uniquement si le lookahead est "*ID* :" (ou l'un des autres lookaheads correspondant à cette réduction dans l'état précédent). Cette grammaire est donc LR(2), donc LR(k).
Le problème de cette grammaire provient du ";" optionnel de la grammaire des fichiers d'entrée de Yacc. Celle-ci permet d'écrire :

```
ID : ID ID : ID
   ↑
```

Or lorsque que l'on se situe au niveau de la flèche, et avec un lookahead de 1 symbole terminal, il n'est pas possible de savoir si l'on doit réduire la règle **rhses** → **rhs** ou bien empiler le token *ID*. Avec un lookahead de deux symboles, on peut désambiguïser ; si le symbole qui suit le "*ID*" est un ":", alors on peut réduire : on vient de démarrer un nouvelle règle de production (*ID*: ...) ; sinon, on empile *ID*.

Best-of:

- Cette grammaire est LR(k) avec $k = 2$. Ici le k représente le nombre de retour récursif existant dans les règles et on voit ici qu'il s'agit de 2.
- Étant donné que cette grammaire se trouve dans un fichier d'entrée de Yacc et que Yacc repose sur un algorithme LALR(1), cette grammaire est LR1.
- Elle est LR(1) car récursif à gauche.
- Elle est LR(2) car le scanner a besoin de voir 1 token plus loin pour être efficace.
- Cette grammaire est LR(1) car elle a des symboles terminaux.
- Cette grammaire est LR(1) parce qu'il y'a un seul symbole.
- Cette grammaire est récursive à gauche, elle ne peut donc être directement parsée par Yacc qui est par défaut LL(1).
- LR(k) avec k = 2, car on à une récursion gauche et droite.
- Elle semble LR(3).
- Cette grammaire es LR(1), mais pas LR(k).
- Il s'agit d'une grammaire LR(k) car il y a une lecture en une passe de gauche à droite avec 3 symboles de regard avant.

2. Vous avez vu en cours de Théorie des Langages qu'en pratique, Yacc ne peut pas utiliser cette grammaire *directement*[1]. Pour quelle raison ?

Correction: Parce que Yacc utilise un parser LALR(1), pas assez puissant pour lire la grammaire précédente, qui est LR(2).

Best-of:

- Yacc ne connaît pas le ε, symbole du neutre (...).
- Yacc devrait faire deux passes.
- Il faut qu'il n'y est qu'un point d'entrée pour que Yacc puisse utiliser cette grammaire.
- Akim Demaille n'a pas travaillé à la réalisation de Yacc.
- Car elle [la grammaire] est trop abstraite pour lui [Yacc].
- Car Yacc ne possède pas de dictionnaire, donc il ne peut comprendre cette grammaire.
- La syntaxe n'est pas la bonne pour la flèche par exemple.

3. Quelle modification peut-on introduire pour faire accepter cette grammaire à Yacc ?

[1] Les entrées de Yacc sont donc définies avec une grammaire que Yacc ne sait pas parser, ce qui est *a priori* paradoxal.

Correction: L'astuce consiste à transformer la séquence de tokens "*ID* :" en un seul symbole, de sorte à lever l'ambiguïté soulevée dans la question 1 tout en rendant la grammaire (LA)LR(1) :

```
grammar → rules              rhses → rhs
grammar → grammar rules      rhses → rhses "|" rhs
                             rhses → rhses ";"
rules → ID_COLON rhses
                             rhs → rhs ID
                             rhs → ε
```

En effet, l'état qui provoquait le conflit shift/reduce devient alors :

rhses	→	**rhs** .	$
rhses	→	**rhs** .	*ID_COLON*
rhses	→	**rhs** .	|
rhses	→	**rhs** .	;
rhs	→	**rhs** . *ID*	$
rhs	→	**rhs** . *ID*	*ID_COLON*
rhs	→	**rhs** . *ID*	*ID*
rhs	→	**rhs** . *ID*	|
rhs	→	**rhs** . *ID*	;

Le conflit shift/reduce a disparu.

Une autre possibilité, si l'on s'autorisait de modifier le langage reconnu par cette grammaire serait de rendre le ";" obligatoire : ainsi, il n'y aurait aucune ambiguïté quant à la fin d'une définition de règle de production.

Une dernière possibilité, *si l'on disposait de Bison* (au lieu de Yacc) serait d'utiliser l'algorithme GLR, qui différerait le résolution du conflit à l'exécution.

Best-of:

- Il faut passer cette grammaire à un lexeur.
- Transformer les récursions à gauche en récursions à droite.
- On peut introduire une option dont je ne me souviens pas le nom en haut du fichier[2].
- On supprime tout.

4. On décide d'étendre cette grammaire pour pouvoir *nommer* les symboles dans les règles, plutôt que d'y faire référence avec des numéros. Concrètement, cela signifie qu'au lieu d'écrire :

```
exp: "identifier" "[" exp "]" "of" exp
{
 $$ = new Array (@$, new Type (@1, $1), $3, $6)
}
```

dans un fichier d'entrée de Bison, on veut pouvoir écrire :

```
exp$res : "identifier"$type "[" exp$size "]" "of" exp$init
{
 $res = new Array(@res, new Type(@type, $type), $size, $init)
}
```

[2] Il s'agit de l'option `%glr-parser` (cf. la correction également).

À partir de la grammaire donnée au début de l'exercice, et corrigée à la question 3, proposez une nouvelle grammaire **LR(1)** prenant en compte cette syntaxe étendue (sans les actions).

Correction: Voici un exemple de réponse.

grammar –> **rules**
grammar –> **grammar rules**

rules –> *ID_COLON* **rhses**
rules –> *ID_DOLLAR_ID_COLON* **rhses**

rhses –> **rhs**
rhses –> **rhses** "|" **rhs** (Nouvelle règle)
rhses –> **rhses** ";"

rhs –> **rhs** *ID*
rhs –> **rhs** *ID* "$" *ID* (Nouvelle règle)
rhs –> eps

L'élément important à prendre en compte est de ne pas réintroduire un *autre* conflit shift/reduce qui rendrait cette nouvelle grammaire LR(k), avec $k > 1$: c'est pourquoi on a introduit un autre "token multiple" ici ("*ID_DOLLAR_ID_COLON*" au lieu de "*ID* $ *ID* :").

3 Tigris fluctuat nec mergitur

Best-of: *[Commentaire/traductions (non demandées !) lus en tête de l'exercice sur des copies :]*

- "Le tigre change mais ne meurt pas"
- "Le tigre prend l'eau mais ne sombre pas."
- In principio erat Tigrum...

Dans cet exercice, on se propose de rajouter un nouveau type de données à notre langage de programmation préféré (Tiger) : les nombres à virgule flottante ou flottants (*floats*).

~~Nous allons~~ Vous allez (c'est votre épreuve, après tout) donc passer en revue les différentes parties de notre compilateur afin de déterminer quels sont les aménagements nécessaires.

Rappel : le langage Tiger utilisé dans cet exercice est celui qui est décrit dans le Manuel de Référence du Compilateur Tiger, qui a été utilisé comme référence tout au long du projet du même nom.

1. **Spécification lexicales.** Que faut-il ajouter aux spécifications *lexicales* du langage pour supporter les flottants ?

   Correction: Il suffit d'ajouter un token `FLOAT` à la grammaire reconnaissant les mots correspondants à l'expression rationnelle `[0-9]+\.[0-9]*`.

   Best-of:
   - On choisis les flottants décris dans IEEE 1394[3].

[3] Attention à ne pas confondre IEEE 1394 (http://en.wikipedia.org/wiki/IEEE_1394), une norme de bus série également appelée "Firewire" ou "i.LINK" ; et IEEE 754 (http://en.wikipedia.org/wiki/IEEE_754-1985), une norme de représentation des nombres à virgule flottante !

2. **Scanner.** Comment étendre le scanner, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`scantiger.ll`' pour supporter ces nouvelles spécifications lexicales ?

Correction: Quelque chose de ce genre :

```
/* ... */
float   [0-9]+\.[0-9]*
/* ... */
%%
/* ... */
{float} {
        float val = 0;
        try
          {
            val = boost::lexical_cast<float> (yytext);
          }
        catch (boost::bad_lexical_cast&)
          {
            tp.error_ << misc::error::scan
                    << *lloc
                    << ": invalid literal float: "
                    << yytext << std::endl;
            val = FLT_MAX;
          }
        lval->fval = val; return token::FLOAT;
      }
/* ... */
%%
/* ... */
```

Il faudra également penser à définir un nouveau token `FLOAT` dans '`parsetiger.yy`' ainsi qu'étendre l'union `lval` avec un champ `fval` de type `float` pour pouvoir y stocker la valeur sémantique associée à `FLOAT` (cf. réponse à la question 4).
Attention à ne pas ajouter de tokens existant déjà : beaucoup proposent l'ajout d'un token pour "," ou "." alors qu'ils existent déjà (`COMMA` et `DOT` respectivement).

Best-of:

- Il faut rajouter des tokens (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).
- Il faut ajouter un token `VIRGULE`.
- (. . .) définir l'expression rationnelle correspondante :

  FLOAT [0-9]+'.'[0-9]{2}

- yylval.fval = myfloatatoi (yytext);
- Il faut ajouter une pile pour les flottants (. . .).

3. **Spécification syntaxiques.** Que faut-il ajouter aux spécifications *syntaxiques* du langage pour supporter les flottants ?

Correction: Il suffit d'étendre la grammaire pour les flottants littéraux soient acceptés comme expressions :

$$\text{exp} \rightarrow \text{float}$$

où `float` est un nouveau terminal.

Best-of:
- (. . .) spécifications synthaxiques.

4. **Parser.** Comment étendre le parser, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`parsetiger.yy`' pour supporter ces nouvelles spécifications syntaxiques ?

Correction: Les ajouts nécessaires sont :
- un champ `fval` dans l'union de Bison pour y stocker la valeur sémantique associée à `FLOAT` (un flottant littéral),
- la définition du token `FLOAT`,
- la règle de production de la question précédente.

```
%union
{
  // ...
  float fval;
  // ...
}

// ...

%token <fval> FLOAT  "float"

// ...

%%
// ...
exp: FLOAT { $$ = new ast::FloatExp (@$, $1); }
// ...
```

où `ast::FloatExp` désigne la classe des flottants littéraux dans l'AST, très similaire à `ast::IntExp`. On pourrait d'ailleurs envisager une classe paramétrée `ast::NumExp<T>` pour factoriser `ast::FloatExp` et `ast::IntExp` (avec `T = int` et `T = float` respectivement).

5. **Liaison des noms.** Que faut-il changer dans le `Binder` vis-à-vis des flottants ?

Correction: *Presque* rien : les modifications introduites par les questions précédentes n'ont en effet pas d'incidence sur les noms, mais il faut que le `Binder` sache que `float` est un nom de type prédéfini (*builtin*). Très concrètement, cela signifie que `bind::Binder::operator() (ast::NameTy&)` doit être équipé pour accepter l'identifiant `float` comme un nom de type valide.

Best-of:
- La place que prend un float est plus grande que celle d'un int. Il faut adapter ça.

6. **Sémantique des flottants.** Le nouveau type de données flottant nécessite des règles de sémantique précises (manipulation de variables ou littéraux flottants). Au moins deux éléments essentiels sont concernés, qui sont liés à deux types de *polymorphismes* que vous connaissez.

   (a) Citez ces deux polymorphismes.

Correction:

Le polymorphisme de coercition En effet, nous souhaitons pouvoir convertir (explicitement ou implicitement) des expressions de type entier vers des expressions de type flottant (et vice versa). Si nous décidons de rendre ces conversion implicites, alors nous incluons une forme de polymorphisme de coercition, permettant de voir un entier comme un flottant, et vice versa (comme en C et C++). Si nous décidons que ces conversions doivent être explicites, nous devons pourvoir le langage de constructions (mots-clefs ou routines) pour ce faire (comme en Objective Caml).

Le polymorphisme de surcharge (dit *ad hoc*) Les opérateurs d'arithmétique unaire (opposé) et binaire (addition, soustraction, multiplication et division) sont désormais valables pour deux types de valeurs : les entiers et les flottants. Là encore, deux possibilités :

- Utiliser les mêmes symboles pour ces opérations, quels que soient les types des opérandes ; on décide donc de fournir des versions *surchargées* de ces opérateurs, ce qui permet d'écrire

  ```
  42 + 51;     /* 'int + int' overload. */
  3.14 + 2.718 /* 'float + float' overload. */
  ```

  sans ambiguïté. Ce choix de surcharge se marie bien avec celui de la coercition (à condition de fournir des règles de conversions non ambiguës vis-à-vis de la surcharge), ce qui permet de mélanger entiers et flottants comme opérandes :

  ```
  42 * 3.14    /* Cast LHS to 'float' and select
                  the 'float + float' overload. */
  ```

- L'autre choix consiste à ne pas fournir de surcharge pour `+`, `-`, `*` et `/` et donc fournir des opérateurs nommés différemment, selon le type des opérandes (par exemple, "`+`" et "`+.`" pour additionner deux entiers ou deux flottants respectivement, comme en Objective Caml). Ce choix se combine assez naturellement avec celui de l'absence de coercition.

Best-of:

- Boxing et Unboxing.
- Le polymorphisme avec les entiers.
- Polymorphisme de type.
- Polymorphismes païens et occultes.
- Polymorphisme implicite.
- Polymorphisme (celui qui englobe les autres).
- Polymorphisme d'inclusion, $\mathbb{Z} \subset \mathbb{R}$.
- Polymorphisme lexical et syntaxique.
- Polymorphisme (...) de substitution.
- Polymorphisme de type de sortie.
- Polymorphisme de classe.

(b) Quelles règles décidez-vous d'appliquer concernant les éléments suivants du langage dans lesquels les flottants sont impliqués :

Correction: Dans ce corrigé, on fait le choix que l'introduction des flottants vient avec la coercition et la surcharge des opérateurs sur entiers et flottants, mais comme rien n'obligeait à faire ce choix dans le sujet, toutes les autres proposition cohérentes ont été acceptées.

i. affectation & initialisation des variables locales,

ii. initialisation des arguments effectifs des fonctions,

Correction: Pour ces deux items, au vu du choix qui a été fait ci-avant (coercition), on autorise donc des variables (ou des arguments effectifs) de type entier et flottant à être initialisé(e)s au choix par des variables entières ou flottantes.
En conséquence, le type-checker devra considérer les types `Int` et `Float` comme *compatibles*.

iii. expressions binaires arithmétiques,

Correction: Comme évoqué plus haut, si on fait le choix de la surcharge et de la coercition, on pourra considérer que le compilateur fournit deux surcharges par opérateur `+`, `-`, `*` et `/`, l'une prenant en argument deux entiers, l'autre deux flottants. On pourra accepter de typer ainsi les nœuds `OpExp` `+`, `-`, `*` et `/` :

- On note respectivement *lhs* et *rhs* les opérandes gauche et droite de l'opération binaire arithmétiques évaluée, elle-même notée *op*.
- On note type(x) le type associé au nœud x.
- Si type(*lhs*) = type(*rhs*) = `Int`, alors type(*op*) ← `Int` et la surcharge choisie sera celle qui prend deux entiers.
- Si type(*lhs*) = type(*rhs*) = `Float`, alors type(*op*) ← `Float` et la surcharge choisie sera celle qui prend deux flottants.
- Si type(*lhs*) = `Int` et type(*rhs*) = `Float`, alors type(*op*) ← `Float` et la surcharge choisie sera celle qui prend deux flottants, après conversion de *lhs* en flottant.
- Idem pour type(*lhs*) = `Float` et type(*rhs*) = `Int`, *mutatis mutandis*.

Best-of:

- Il faut donner une forte priorité à "," par rapport à l'affectation et initialisation.

iv. expressions binaires logiques.

Correction: Tiger ne possède pas de type spécifique pour les booléens et utilise `Int` pour les représenter. Considérer que `Float` peut être utilisé pour coder une valeur booléenne n'est pas très sain, aussi décidons-nous de ne pas modifier le type-checker vis-à-vis des opérateurs binaires logiques `&` et `|`, ce qui revient à dire que la condition d'un nœud `IfExp` devra toujours être de type entier, puisque `&` et `|` sont désucrés en `if-then-else`. (Idem pour `WhileExp`.)

Best-of:

- Entre deux flottants bit à bit avant et après la virgule.

Vous êtes libres des choix de sémantique de cette extension, mais la cohérence de ceux-ci sera bien sûr prise en compte.

Vous n'êtes pas obligés de calquer la sémantique d'un autre langage, mais il peut-être utile de se rappeler ce que vous connaissez dans d'autres langages au sujet des flottants.

7. **Vérification des types.** Qu'allez-vous changer dans le module `type` du compilateur ? Réfléchissez bien, et n'oubliez rien !

Correction: Ici aussi, plusieurs choix étaient possibles. Nous donnons *une* réponse, qui découle logiquement des choix précédents. Toutes les réponses sensées ont été acceptées.

- Ajouter nouvelle classe singleton `type::Float` pour représenter le type des flottants, dont la méthode `compatible_with` renvoie "vrai" lorsque l'opérande est `Float` ou `Int`.
- Étendre la méthode `type::Int::compatible_with` pour qu'elle réponde "vrai" lorsque son opérande est `Float`.
- Modifier `type::TypeChecker::operator() (ast::OpExp&)` comme indiqué dans la correction de la question 6(b)iii.
- Ajouter une méthode `type::TypeChecker::operator() (ast::FloatExp& e)` affectant le type `Float` à `e`.
- Accepter `float` comme nom de type (builtin) valide dans `type::TypeChecker::operator() (ast::NameTy&)`, et lui associer le type `Float`.

Best-of:

- J'ai pas de mirroir...

8. **Représentation intermédiaire** Faut-il ajouter/modifier quelque chose dans le langage Tree ?

Correction: Oui : actuellement nous ne disposons d'aucune instruction pour

- coder un flottant littéral ;
- construire des opérations binaires arithmétiques flottantes (c'est-à-dire, ayant des opérandes flottants.

Il faut donc étendre TREE avec deux opérateurs (expression) supplémentaires :

FCONST(f) La constante flottante f.

FBINOP(o, e_1, e_2) L'application de l'opérateur arithmétique binaire flottant o sur les opérande e_1 et e_2.

Les conversions entier → flottant et flottant → entier peuvent soit faire l'objet de deux opérateurs supplémentaires dans TREE, soit être implémentés sous forme de routines (builtins) dans le *runtime* Tiger étendu.

9. **Traduction (vers HIR).** Y'a-t-il des changements à apporter au visiteur chargé de la traduction vers la représentation intermédiaire de haut niveau ? Justifiez votre réponse.

Correction: Oui :

- il faut implémenter `translate::Translator::operator() (const ast::FloatExp&)` pour générer une constante FCONST ;
- `translate::Translator::operator() (const ast::CallExp&)` doit effectuer les conversions entier → flottant et flottant → entier nécessaires sur les arguments effectifs le cas échéant, pour que ceux-ci aient les types attendus par la fonction ;
- idem pour `translate::Translator::operator() (const ast::OpExp&)`, `translate::Translator::operator() (const ast::AssignExp&)`, `translate::Translator::operator() (const ast::VarDec&)` et `translate::Translator::operator() (const ast::FunctionDec&)` (valeur de retour).

Best-of:

- Non, le désucrage se charge de tout !!

10. **Canonisation (HIR vers LIR).** La canonisation est-elle affecté par cette extension ? Justifiez votre réponse.

Correction: Il suffira d'étendre la canonisation des arbres pour que FCONST et FBINOP soient considérés de la même façon que CONST et BINOP dans le processus de réécriture.

Best-of:

- [La canonisation] supprime les instructions "MOVE" inutiles (la source et la destination sont les mêmes).

11. **Langage *Assem*.** Nous allons considérer que nous ciblons l'architecture MIPS, comme dans le projet (par défaut). Que doit-on ajouter dans le langage d'assemblage à registres infinis *Assem* ?

Correction: Il faut que le langage d'assemblage supporte les instructions sur les flottants, et éventuellement des registres dédiés (c'est le cas de MIPS).

Best-of:

- (. . .) tenir compte de la représentation des flottants sur deux nombre (partie flottante, partie entière).
- Dans un premier temps, il faudrait pouvoir dissocier la partie entière et décimale et pouvoir les calculer.

12. **Sélection d'instructions (génération de code).** Quels sont les changements à apporter au générateur de code en langage d'assemblage ?

Correction: Les opérateurs FCONST et FBINOP doivent être prises en compte dans la sélection d'instruction, afin de générer les instructions MIPS correspondantes.

13. **Analyse de vivacité.** L'analyse de vivacité est-elle modifiée par cette extension ? Justifiez votre réponse.

Correction: Dans le cas de MIPS, oui : en effet, l'architecture MIPS dispose de 32 registres flottants 32-bit. Il faudra donc calculer un graphe de vivacité pour les variables flottantes en sus de celui des variables entières.

Best-of:

- Je ne vois toujours pas de changement, mais je commence à m'inquiéter de répéter à chaque question cette réponse.
- On aura une perte de vivacité, car la gestion de flottants est lourde sur le plan de l'utilisation de la mémoire et des opérations à effectuer.
- Oui, les calculs sont plus longs.

14. **Allocation de registres.** L'allocation de registres est-elle modifiée par cette extension ? Justifiez votre réponse.

Correction: Oui, en conséquence de la réponse précédente : il faudra allouer les temporaires flottantes dans des registres flottants.

Best-of:

- Non, car les registres ne peuvent être des nombres flottants.

15. **Bibliothèque standard.** Quelles routines pourrait-on souhaiter ajouter à la bibliothèque standard du langage Tiger dans le cadre de cette extension ?

Correction: Par exemple :

- les conversions entier → flottant et flottant → entier (cf. réponse à la question 8) ;
- `print_float`, `atof`/`strtod` ;
- des fonctions trigonométriques: `cos`, `sin`, etc.
- des fonctions d'arrondi comme `ceil`, `floor`, `trunc`, etc.
- d'autres fonctions mathématiques comme `sqrt`, `log`, `exp`, etc.
- des constantes usuelles : `pi`, `e`, etc.

Best-of:

- troncage, arondissement, ...
- c $(2,98 \cdot 10^8 \ \ \mathrm{m} \cdot \mathrm{s}^{-1})$, g, G, m_N (masse d'un neutron), m_p, m_e.
- Un convertisseur de monnaie.

16. **Questions Bonux™.** Après les nombres flottants (qui sont une représentation "approchée" de $\mathbb{R}$), on souhaite ajouter un type de données pour représenter les nombre complexes ($\mathbb{C}$). Dans cette question, on admet que l'on dispose d'un compilateur Tiger avec le support des flottants.

    Cette nouvelle extension peut se traiter de deux façons différentes au moins :

    - de façon *intrusive*, en modifiant le langage lui-même (ce qui implique une modification du compilateur) ;
    - de façon *non intrusive*, sans affecter le langage (en étendant la bibliothèque standard par exemple).

    Les questions suivantes donnent lieu à des points de bonus :

    (a) Quels sont les changements à apporter au compilateur dans la première approche ? (Vous pouvez reprendre le plan des questions 1 à 15 pour répondre.)

Correction: Quelques idées :

- Ajouter le mot-clef `i` (ou `j`) pour représenter le nombre imaginaire pur tel que $i^2 = -1$ (ou $j^2 = -1$) au spécifications lexicales. Par la suite, on utilisera la notation `j` (comme les physiciens), car `i` est un nom de variable très utilisé.
- Lui associer un token (par exemple `J`) dans le couple scanner/parser.
- Faire de `j` une expression, en ajoutant une règle

  exp → J

  dans la grammaire.
- Ajouter à la syntaxe abstraite une classe `ast::JExp` pour représenter j ;
- Faire de `complex` un nom et un type valides.
- Ajouter un type `type::Complex` à la hiérarchie des types de Tiger ;
- Attribuer ce type `type::Complex` aux nœuds `ast::JExp`.
- Autoriser dans le type-checker des conversions entier → complexe et flottant → complexe, mais pas leurs réciproques ; de même, autoriser les surcharges d'opérateurs arithmétiques en promouvant les types dans le sens suivant : entier → flottant → complexe. Fournir les surcharges d'opérateurs pour les complexes.
- Traduire les variables complexes en couples de flottants (en utilisant un enregistrement comme étape intermédiaire, par exemple).
- Traduire les opérations arithmétiques sur les complexes en appels de fonctions (`complex_add`, complex_sub, etc.).
- Le reste du back-end est inchangé.
- Fournir les opérations usuelles sur les complexes sous forme de fonctions de la bibliothèque standard du langage :
  - parties réelles et imaginaires : `re`, `im` ;
  - module et argument : `mod`, `arg` ;
  - etc.

Best-of:

- *[Au sujet du support des complexes :]*
  - Très compliqué… Mais D le fait[4] !
- *[Au sujet des fonctionnalités ajoutées à la bibliothèque standard :]*
  - Trace le cercle trigonométrique représentant le complexe.
  - Animation 3D avec OpenGL pour représenter le complexe.

(b) Comment implémenteriez-vous le second cas ? (Les autres langages peuvent donner des idées.)

[4] Pour info, C++ aussi (http://www.ddj.com/web-development/184401491), mais son approche serait à ranger dans le second cas, non intrusif (NDC).

Correction: Deux propositions, selon que l'on décide d'utiliser les constructions OO ou pas.

Panther (sans POO)

- Introduire dans le prélude un type `complex`, qui n'est autre qu'un enregistrement contenant les parties réelles et imaginaires sous forme de deux flottants :

```
type complex = { re : float, im : float }
```

- Ajouter les opérations usuelles sous forme de builtins (écrites en Panther ou fournies part le runtime) dans le prélude :

```
primitive addc (a : complex, b : complex) : complex
primitive subc (a : complex, b : complex) : complex
primitive mulc (a : complex, b : complex) : complex
primitive divc (a : complex, b : complex) : complex

primitive re (c : complex) : float
primitive im (c : complex) : float

primitive mod (c : complex) : float
primitive arg (c : complex) : float

/* ... */
```

Tiger (avec POO) Réappliquer la même idée, mais en tirant parti de l'encapsulation que fournissent les classes :

```
class complex =
{
  var re : complex = 0.
  var im : complex = 0.

  method re () : float = re
  method im () : float = im

  method add (rhs : complex) : complex = /* ... */
  method sub (rhs : complex) : complex = /* ... */
  method mul (rhs : complex) : complex = /* ... */
  method div (rhs : complex) : complex = /* ... */

  method mod () : float = /* ... */
  method arg () : float = /* ... */

  /* ... */
}
```

(Rien n'empêche de fournir tout ou partie de ces services comme fonctions externes également.)

Correction: On peut encore étendre en ajoutant un type supplémentaire fournissant une représentation polaire des complexes, etc.

Cette seconde approche a l'avantage d'être plus simple (et non intrusive), mais elle fournit des complexes moins "sucrés" que la première :
`a + b * c` *vs* `addc (a, mulc (b, c))` *vs* `a.add (b.mul (c))`

Best-of:

- Le Klingon[5]ou Klindon (je ne sais plus comment on dit) m'inspire fortement. En effet, il est complexe et imaginaire (. . .).

4 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire, comme lors de l'épreuve du premier semestre.

Les renseignements ci-dessous ne serons 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 les feuilles de QCM qui vous sont remises. N'y passez pas plus de dix minutes.

Le cours

5. Quelle a été votre implication dans les cours (THL, CCMP, TYLA) ?

   A Rien.

   B Bachotage récent.

   C Relu les notes entre chaque cours.

   D Fait les annales.

   E Lu d'autres sources.

6. 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.

7. 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.

8. La charge générale du cours (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).

[5]Voir http://fr.wikipedia.org/wiki/Klingon_(langue) et http://en.wikipedia.org/wiki/Klingon_language (NDC).

Les formateurs

9. 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.

10. 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

11. 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.

12. 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.

13. 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 14-20 (1 pt) 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

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

Questions 22-37 (1 pt) Comment furent les étapes du projet (ne pas répondre à celles que vous n'avez pas faites). Répondre selon la grille suivante. (Une seule réponse attendue par question.)

A Trop facile.

B Facile.

C Nickel.

d Difficile.

E Trop difficile.

21. Mini-projet en Tiger (LZW)

22. `TC-0`, Scanner & Parser.
23. `TC-1`, Scanner & Parser en C++, Autotools.
24. `TC-2`, Construction de l'AST.
25. `TC-3`, Liaison des noms.
26. `TC-4`, Typage.
27. Desucrage des constructions objets (transformation Tiger → Panther)
28. `TC-5`, Traduction vers représentation intermédiaire.

29. `TC-6`, Simplification de la représentation intermédiaire.
30. `TC-7`, Sélection des instructions.
31. `TC-8`, Analyse du flot de contrôle.

32. `TC-9`, Allocation de registres.

33. Option `TC-E`, Calcul des échappements.
34. Option `TC-A`, Surcharge des fonctions.
35. Option `TC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
36. Option `TC-B`, Vérification dynamique des bornes de tableaux.
37. Option `TC-I`, Mise en ligne du corps des fonctions.