Correction du partiel CMP1 de S2

EPITA – Promo 2016
Notes de cours personnelles autorisées.
Livres, polycopiés et planches de cours imprimées interdits.
Machines (calculatrices, ordinateurs, tablettes, téléphones) interdites.

Avril 2014 (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 11 pages.

1 Starters

1. Combien de caractères (au sens « glyphe ») différents peut-on théoriquement représenter en UTF-32, un encodage à taille fixe qui utilise exactement 32 bits pour coder chaque élément ?

   Correction: Sans trop de surprise, 2^{32}.

   Best-of:
   - 8
   - 256
   - 32^{33}
   - 2^4
   - 1024
   - 32 bits = 2^5 ⇔ on peut coder 5 caractères
   - 32
   - 16^4
   - $2^{32} = 2^{2\times4} = 4^{16} = 4^{4\times2} = 16^4$ caractères différents
   - 101
   - 2^{33}
   - 255
   - $2^{33} - 1$

2. Quel est l'intérêt d'un parser réentrant ?

   Correction: Un parser réentrant (encore dit « pur ») supporte des exécutions multiples et concurrentes. En particulier, il autorise la récursion, ce qui permet de déclencher le parsing d'un nouveau fichier depuis un parser en cours d'exécution. Certaines copies citent le désucrage comme application, mais cette réponse n'a de sens que s'il est précisé que ce dernier est réalisé *en syntaxe concrète*.

   Best-of:
   - Il permet de revenir en arrière.
   - Moins d'erreur possible car plusieurs passages.

3. Citez un bénéfice apporté par la liaison statique de noms dans un langage compilé.

   Correction: Au choix : la sûreté, la clarté ou encore l'efficacité du code produit (rapidité, taille).

   Best-of: Recherche en temps constant.

4. Comment s'appelle la construction employée dans la définition de `incr` dans le code Objective Caml ci-dessous ?

```
let start x =
  let incr y = x + y
  in incr
in
  let x = -42 in
  let y = start x in
  y 51
;;
```

Correction: Une fermeture (*closure*) : une fonction qui capture des références à des variables libres dans l'environnement lexical.

Best-of: La définition de `incr` utilise la cloture transitive.

2 Un peu d'imagination

On considère la grammaire suivante au format BNF, engendrant un langage d'expressions complexes (au sens de $\mathbb{C}$, l'ensemble des nombres complexes).

```
r0  ⟨exp⟩ ::= ⟨num⟩                # A literal number.
r1          | "i"                  # The imaginary unit number.
r2          | ⟨exp⟩ "+" ⟨exp⟩      # Complex addition.
r3          | ⟨exp⟩ "-" ⟨exp⟩      # Complex subtraction.
r4          | ⟨exp⟩ "*" ⟨exp⟩      # Complex multiplication.
r5          | ⟨exp⟩ "/" ⟨exp⟩      # Complex division.
r6          | "re" ⟨exp⟩           # Real part of a complex number.
r7          | "im" ⟨exp⟩           # Imaginary part of a complex number.
r8          | "(" ⟨exp⟩ ")"        # Explicit grouping.
```

On souhaitera dans l'ensemble de cet exercice suivre les règles usuelles de priorité et d'associativité classiquement utilisées en mathématiques. On considérera aussi que les opérateurs unaires (`re`, `im`) sont plus prioritaires que les opérateurs binaires.

1. Que signifie BNF ?

Correction: Backus-Naur Form.

Best-of:
- Bacchus No Form
- Baccus Nand Format
- Back Naur Form
- Back Naus Form
- Backaur Naus Form
- Backup-Naur farm
- Backus No form
- Backus Normalize Format
- Backus norm form
- Backus-Nohr-Form
- Bauer N... F...
- Baur Naur Formalisation
- Baur Naur Format
- Berckley Normal Form
- Berkley Notation Format
- Best Node Feed
- Bibliothèque Normalisée en Forme
- Binary Extend Form
- Binary Formalized Form
- Binary Format
- Binary Native Format
- Binary Nested Format
- Binary Node Form
- Bison and Flex
- Boolean Nuclear Farmville
- À peu de choses près : Baker Naus-Frauman...

2. On s'intéresse à l'implémentation de ⟨*num*⟩, qui désigne un nombre littéral (entier ou réel, non signé) du point de vue du scanner. Écrivez une expression rationnelle décrivant de tels nombres.

   Correction: Quelque chose comme `[0-9]+(\.[0-9]*)?` était acceptable (même s'il était possible de proposer des expressions plus complexes, capable d'accepter des entrées comme '`.51`').

   Best-of: `[7-9]+[12-15]`

   Par la suite, et afin de ne pas compliquer l'exercice outre mesure, on considérera que les nombres littéraux traités par le scanner seront restreints aux entiers – c'est-à-dire que l'on posera ⟨*num*⟩ = `INT`, où `INT` est un token représentant un entier littéral non signé.

3. Quelles sont les symboles terminaux de cette grammaire ?

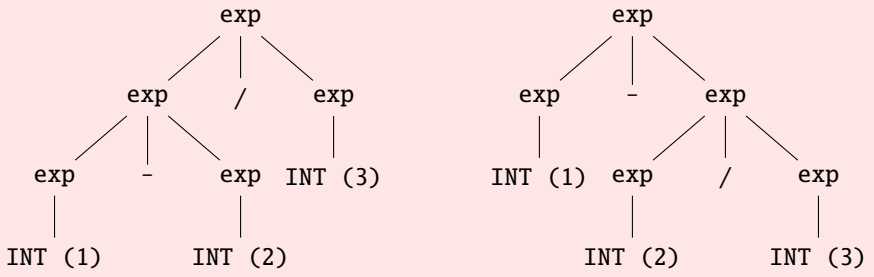
   Correction: `INT` (ou `num`), '`i`', '`+`', '`-`', '`*`', '`/`', '`re`', '`im`', '`(`' et '`)`'.

4. Quelles sont les symboles non-terminaux de cette grammaire ?

   Correction: Le seul non-terminal est `exp`.

5. Cette grammaire est-elle ambiguë ? Justifiez votre réponse.

   Correction: Elle est bien entendu ambiguë, car il est possible de parser des entrées valides de plusieurs façons. Par exemple, `1 - 2 / 3` :



   Best-of:
   – Il n'y a aucune ambiguïté, les règles peuvent se reduce sans avoir besoin de shifter afin de déterminer la règle où nous nous trouvons.
   – Les [opérateurs] unaires sont associatifs à droite [...]

6. Qu'est-ce qu'une valeur sémantique ?

   Correction: Une information auxiliaire attachée à certains symboles. Elle peut être produite par le scanner (par exemple, la valeur d'un entier littéral représentée par un token `INT`) ou par le parser (un morceau d'AST associé au non-terminal `exp`).

   Best-of: Une valeur sémantique est un symbole terminal.

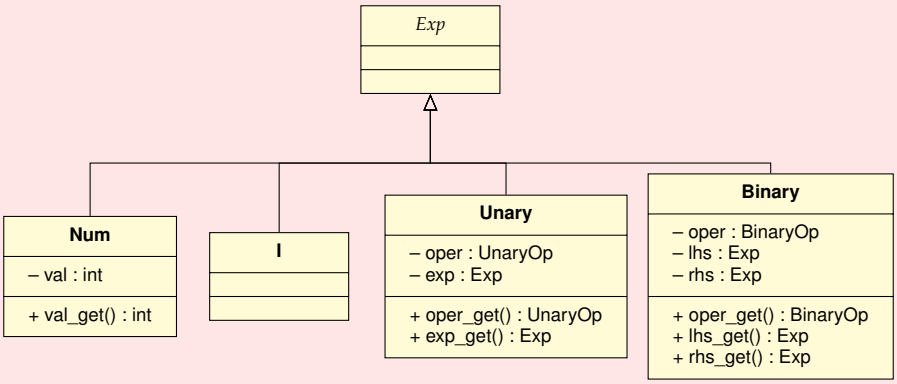
7. Proposez une syntaxe abstraite permettant de représenter les entrées du langage engendré par la grammaire ci-dessus, soit sous la forme d'une grammaire (de syntaxe abstraite), soit sous la forme d'un diagramme de classes.

Correction: Voici une possibilité :

```
Exp ::= Num(int)
      | I
      | Unary(UnaryOp, Exp)
      | Binary(BinaryOp, Exp, Exp)

UnaryOp  ::= re | im
BinaryOp ::= add | sub | mul | div
```

Ou encore :



où `UnaryOp` désigne une énumération contenant les valeurs `re` et `im` ; et `BinaryOp` une énumération contenant les valeurs `add`, `sub`, `mul` et `div`.

(Note : les indications de méthodes n'étaient pas demandées.)

8. Continuez (sur votre copie) le schéma du parsing de l'entrée `2 + 3 * i`, commencé ci-dessous, en affichant les états successifs de la pile (symbolique et sémantique) de l'automate LR engendré par la grammaire ci-dessus. Le symbole '`$`' désigne le marqueur de fin d'entrée. Cette analyse doit *in fine* produire un arbre de syntaxe abstraite (AST).

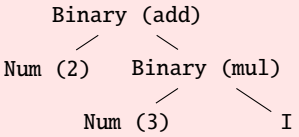
action	**stack**	**input**
	⊢	`2 + 3 * i $` ⊣
shift (2)		
	⊢ INT 2	`+ 3 * i $` ⊣
reduce (r_0)		
	⊢ exp Num(2)	`+ 3 * i $` ⊣
...		

Correction:

action	stack	input
	⊢	2 + 3 * i $ ⊣
shift (2)		
	⊢ INT 2	+ 3 * i $ ⊣
reduce (r_0)		
	⊢ exp Num(2)	+ 3 * i $ ⊣
shift (+)		
	⊢ exp '+' Num(2)	3 * i $ ⊣
shift (3)		
	⊢ exp '+' INT Num(2) 3	* i $ ⊣
reduce (r_0)		
	⊢ exp '+' exp Num(2) Num(3)	* i $ ⊣
shift (*)		
	⊢ exp '+' exp '*' Num(2) Num(3)	i $ ⊣
shift (i)		
	⊢ exp '+' exp '*' 'i' Num(2) Num(3)	$ ⊣
reduce (r_1)		
	⊢ exp '+' exp '*' exp Num(2) Num(3) I	$ ⊣
reduce (r_4)		
	⊢ exp '+' exp Num(2) Binary(mul, Num(3), I)	$ ⊣
reduce (r_2)		
	⊢ exp Binary(add, Num(2), Binary(mul, Num(3), I))	$ ⊣
accept		

Écrivez ou dessinez l'AST issu de ce parsing.

Correction: Il s'agit simplement de la valeur sémantique associée au dernier non-terminal (`exp`), juste avant l'action de décalage du marqueur de fin d'entrée (*accept*) :



Attention, beaucoup de copies contenaient un arbre de dérivation ou encore un arbre laissant apparaître des détails de syntaxe concrète inutiles (typiquement, des parenthèses).

9. Même question pour l'entrée `im (51 * i + 43)`.

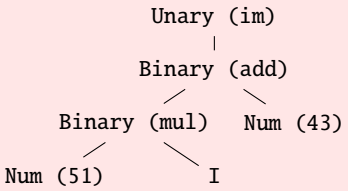
Correction:

action	stack	input
	⊢	im (51 * i + 43) $ ⊣
shift (im)	⊢ 'im'	(51 * i + 43) $ ⊣
shift (()	⊢ 'im' '('	51 * i + 43) $ ⊣
shift (51)	⊢ 'im' '(' INT 51	51 * i + 43) $ ⊣
reduce (r_0)	⊢ 'im' '(' exp Num(51)	* i + 43) $ ⊣
shift (*)	⊢ 'im' '(' exp Num(51) '*'	i + 43) $ ⊣
shift (i)	⊢ 'im' '(' exp Num(51) '*' 'i'	+ 43) $ ⊣
reduce (r_1)	⊢ 'im' '(' exp Num(51) '*' exp I	+ 43) $ ⊣
reduce (r_4)	⊢ 'im' '(' exp Binary(mul, Num(51), I)	+ 43) $ ⊣
shift (+)	⊢ 'im' '(' exp Binary(mul, Num(51), I) '+'	43) $ ⊣
shift (43)	⊢ 'im' '(' exp Binary(mul, Num(51), I) '+' INT 43	) $ ⊣
reduce (r_0)	⊢ 'im' '(' exp Binary(mul, Num(51), I) '+' exp Num(43)	) $ ⊣

Correction: (suite)

```
reduce (r2)
            ⊢   'im'    '('    exp                              ) $ ⊣
                                  Binary(add,
                                    Binary(mul,
                                      Num(51),
                                      I),
                                    Num(43))
shift  ())
            ⊢   'im'    '('    exp    ')'                         $ ⊣
                                 Binary(add,
                                   Binary(mul,
                                     Num(51),
                                     I),
                                   Num(43))
reduce (r8)
            ⊢   'im'    exp                                      $ ⊣
                      Binary(add,
                        Binary(mul,
                          Num(51),
                          I),
                        Num(43))
reduce (r7)
            ⊢   exp                                              $ ⊣
                Unary(im,
                  Binary(add,
                    Binary(mul,
                      Num(51),
                      I),
                    Num(43))
accept
```

L'AST issu de ce parsing est :



10. On considère l'implémentation d'un parser pour le langage utilisé dans cet exercice avec Bison, et la question de la reprise sur erreur face à une entrée syntaxiquement incorrecte. Qu'écririez-vous dans le fichier passé à Bison pour afin d'effectuer une telle reprise sur erreur ?

    Correction: Le mécanisme de reprise sur erreur (par suppression) utilise le token `error` fourni par Bison. Un bon emplacement pour en faire usage est au sein d'un contexte borné par des symboles à gauche et à droite, par exemple à l'intérieur de parenthèses :

    ```
    exp : "(" exp ")"    { $$ = $1; }
        | "(" error ")"  { $$ = Num(0);  /* Dummy semantic value. */ }
    ;
    ```

    On prendra également soin d'ajouter des directives `%destructor` pour libérer correctement la mémoire associée aux symboles défaussés lors de la reprise sur erreur.

    Best-of:
    - `%include GLR`
    - `%expect (1)`

11. On souhaite écrire en C++ un évaluateur des AST produits par notre parser. Comment s'appelle la technique objet usuelle pour mettre en œuvre un tel traitement ?

    Correction: Le design pattern *Visiteur*.

    Best-of: Un lexer

12. Considérez l'implémentation partielle de l'évaluateur ci-dessous, où les nombres complexes sont implémentés par la classe standard C++ `std::complex<float>`, qui fonctionne comme vous l'imaginez. Complétez (sur votre copie) le code manquant identifié par le commentaire bien connu.

```
class Evaluator : public Base
{
public:
  void operator()(const Num& e)
  {
    value_ = e.val_get();
  }

  void operator()(const I&)
  {
    value_ = std::complex<float>(0.f, 1.f);
  }

  void operator()(const Unary& e)
  {
    e.exp_get().accept(*this);
    switch (e.oper_get())
    {
    case Unary::re: value_ = std::real(value_); break;
    case Unary::im: value_ = std::imag(value_); break;
    }
  }

  void operator()(const Binary& e)
  {
    /* FIXME: Some code was deleted here.  */
  }

  std::complex<float> value_get() const
  {
    return value_;
  }

private:
  std::complex<float> value_;
};
```

Correction:

```
class Evaluator : public Base
{

  // ...

  void operator()(const Binary& e)
  {
    e.lhs_get().accept(*this); std::complex<float> lhs = value_;
    e.rhs_get().accept(*this); std::complex<float> rhs = value_;
    switch (e.oper_get())
    {
    case Binary::add: value_ = lhs + rhs; break;
    case Binary::sub: value_ = lhs - rhs; break;
    case Binary::mul: value_ = lhs * rhs; break;
    case Binary::div: value_ = lhs / rhs; break;
    }
  }

  // ...

};
```

13. Écrivez le code de la classe `Base`, dont dérive `Evaluator`.

Correction: L'élément important ici est que les méthodes soient virtuelles pures (polymorphes abstraites).

```
class Base
{
public:
  virtual void operator()(const Num& e) = 0;
  virtual void operator()(const I& e) = 0;
  virtual void operator()(const Unary& e) = 0;
  virtual void operator()(const Binary& e) = 0;

  virtual ~Base() = default;
};
```

On rappelle que le mot clef `virtual` est facultatif devant les méthodes virtuelles redéfinies (*overriding*) dans les sous-classes de Base, telle `Evaluator`.

14. Quelle nom meilleur pourrait-on donner à la classe `Base` du listing précédent ?

Correction: `Visitor`, bien entendu.

Best-of:
- Getter
- Headquarter
- Integer