CMP1 – Construction des compilateurs – S2

EPITA – AppIng1 promotion 2015
Notes de cours manuscrites autorisées
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 6 pages. Les pages 1–3 contiennent l'épreuve elle-même. Les pages 4–6 sont dédiées aux annexes.

1 Généralités

1. Qu'appelle-t-on la partie frontale d'un compilateur ?
2. Rappelez quelles sont les cinq phases de la partie frontale d'un compilateur comme `tc`.
3. Qu'est-ce que BNF ?
4. Quelle différence faites-vous entre un arbre de dérivation (*parse tree*) et un arbre de syntaxe abstraite (*Abstract Syntax Tree*) ?

2 Front end Tiger

Soit le programme Tiger suivant :

```
1  let
2    type f = { n : int, d : int }
3    function f (n : int, d : int) : f = f { n = n, d = d }
4    function a (x : f, y : f) : f = f (x.n * y.d + x.d * y.n, x.d * y.d)
5    function m (x : f, y : f) : f = f (x.n * y.n, x.d * y.d)
6    var p := f(1, 2)
7    var q := f(4, 3)
8  in
9    p := m(a(p, q), q)
10 end
```

La grammaire du langage Tiger est rappelée dans l'annexe A.

1. Quel signification peut-on attribuer au type `f` et aux fonctions `a` et `m` ?
2. Quel rôle joue la fonction `f` de la ligne 3 ?
3. Pour quelle raison le compilateur ne confond-t-il pas le type `f` de la ligne 2 et la fonction `f` de la ligne 3 ?
4. Dans l'expression de la ligne 9, la première occurrence de `p` est une *l-value*. Que signifie ce terme ?

5. On considère à présent une version raccourcie du programme précédent, réduite au jeu de déclarations suivantes :

```
type f = { n : int, d : int }
function f (n : int, d : int) : f = f { n = n, d = d }
var p := f(1, 2)
```

   Représentez l'arbre de syntaxe abstraite de ce programme. Pour vous aider, un récapitulatif des nœuds de la syntaxe abstraite du langage Tiger vous est fourni dans l'annexe B.

6. Sur le schéma de la question précédente, indiquez les liens entre sites d'utilisation et sites de définition à l'aide de flèches partant des premiers et allant vers les seconds (de préférence en utilisant une autre couleur de stylo).
7. On souhaiterait ajouter au programme Tiger originel une fonction de comparaison « intelligente » entre deux variables de type `f`. Proposez une définition d'une telle fonction.

3 Syntaxe abstraite

Dans cet exercice, on s'intéresse à la syntaxe abstraite d'un langage permettant de représenter des expressions arithmétiques. Les nœuds des arbres de syntaxe abstraite (ou Abstract Syntax Trees (ASTs)) de ce langage sont représentés par la hiérarchie de classes C++ ci-dessous :

```
1  struct Exp
2  {
3    virtual ~Exp () {};
4
5  protected:
6    Exp() {};
7    Exp(const Exp& rhs) {};
8    Exp& operator=(const Exp& rhs) {};
9  };
10
11 struct Int : Exp
12 {
13   Num(int val)
14     : Exp(), val_(val)
15   {}
16
17   int val_;
18 };
```

```
19 struct Op : Exp
20 {
21   Op(Exp* lhs, char oper, Exp* rhs)
22     : Exp(),
23       lhs_(lhs), oper_(oper), rhs_(rhs)
24   {}
25
26   virtual ~Op()
27   {
28     delete lhs_;
29     delete rhs_;
30   }
31
32   Exp* lhs;
33   char oper;
34   Exp* rhs;
35 };
```

1. À quoi sert la ligne 3 dans la classe `Exp` ?
2. À quoi servent les lignes 5 à 8 dans la classe `Exp` ?
3. Le code C++ ci-dessus est volontairement court par souci de simplicité et pourrait être amélioré. Quels changements apporteriez-vous ?
4. Pour manipuler ces ASTs, on utilise le design pattern *Visiteur*. Certains langages disposent d'une fonctionnalité qui permet (entre autres) de parcourir des arbres polymorphes comme les ASTs de cet exercice. Quelle est le nom de cette fonctionnalité et que permet-elle de faire ?
5. Écrivez le code de la classe abstraite `Visitor` dont dériveront toutes les classes de visiteurs parcourant nos ASTs.
6. Que faut-il ajouter aux classes `Exp`, `Int` et `Op` pour qu'elle puissent coopérer avec un `Visitor` ?

7. Écrivez un visiteur concret `Evaluator` dérivant du `Visitor` de la question 5, réalisant l'évaluation d'un AST et produisant un résultat numérique. Celui-ci sera accessible via une méthode `value_get()` du visiteur après réalisation du parcours.
8. Le mécanisme de destruction des nœuds des ASTs ci-dessus n'est pas très flexible. Par exemple, il n'est pas possible de supprimer isolément un nœud d'un arbre : l'intégralité du sous-arbre tenu par ce nœud sera également détruit récursivement. Comment pourrait-on modifier les classes de notre AST pour rendre cette gestion mémoire plus souple ?

Annexes

A Grammaire du langage Tiger

Cette grammaire utilise le formalisme Extended Backus-Naur Form (EBNF), où '[' et ']' sont utilisés pour représenter zéro (0) or une (1) occurrence du terme encadré, tandis que '{' et '}' désignent un nombre arbitraire de répétitions (y compris zéro).

```
⟨program⟩ ::=
    ⟨exp⟩
  | ⟨decs⟩

⟨exp⟩ ::=
  # Literals.
    "nil"
  | integer
  | string

  # Array and record creations.
  | ⟨type-id⟩ "[" ⟨exp⟩ "]" "of" ⟨exp⟩
  | ⟨type-id⟩ "{" [ id "=" ⟨exp⟩ { "," id "=" ⟨exp⟩ } ] "}"

  # Object creation.
  | "new" ⟨type-id⟩

  # Variables, field, elements of an array.
  | ⟨lvalue⟩

  # Function call.
  | id "(" [ ⟨exp⟩ { "," ⟨exp⟩ } ] ")"

  # Method call.
  | ⟨lvalue⟩ "." id "(" [ ⟨exp⟩ { "," ⟨exp⟩ } ] ")"

  # Operations.
  | "-" ⟨exp⟩
  | ⟨exp⟩ ⟨op⟩ ⟨exp⟩
  | "(" ⟨exps⟩ ")"

  # Assignment.
  | ⟨lvalue⟩ ":=" ⟨exp⟩

  # Control structures.
  | "if" ⟨exp⟩ "then" ⟨exp⟩ [ "else" ⟨exp⟩ ]
  | "while" ⟨exp⟩ "do" ⟨exp⟩
  | "for" id ":=" ⟨exp⟩ "to" ⟨exp⟩ "do" ⟨exp⟩
  | "break"
  | "let" ⟨decs⟩ "in" ⟨exps⟩ "end"

⟨lvalue⟩ ::= id
  | ⟨lvalue⟩ "." id
  | ⟨lvalue⟩ "[" ⟨exp⟩ "]"
```

```
⟨exps⟩ ::= [ ⟨exp⟩ { ";" ⟨exp⟩ } ]

⟨decs⟩ ::= { ⟨dec⟩ }
⟨dec⟩ ::=
  # Type declaration.
    "type" id "=" ⟨ty⟩
  # Class definition (alternative form).
  | "class" id [ "extends" ⟨type-id⟩ ] "{" ⟨classfields⟩ "}"
  # Variable declaration.
  | ⟨vardec⟩
  # Function declaration.
  | "function" id "(" ⟨tyfields⟩ ")" [ ":" ⟨type-id⟩ ] "=" ⟨exp⟩
  # Primitive declaration.
  | "primitive" id "(" ⟨tyfields⟩ ")" [ ":" ⟨type-id⟩ ]
  # Importing a set of declarations.
  | "import" string

⟨vardec⟩ ::= "var" id [ ":" ⟨type-id⟩ ] ":=" ⟨exp⟩

⟨classfields⟩ ::= { ⟨classfield⟩ }
# Class fields.
⟨classfield⟩ ::=
  # Attribute declaration.
    ⟨vardec⟩
  # Method declaration.
  | "method" id "(" ⟨tyfields⟩ ")" [ ":" ⟨type-id⟩ ] "=" ⟨exp⟩

# Types.
⟨ty⟩ ::=
   # Type alias.
     ⟨type-id⟩
   # Record type definition.
   | "{" ⟨tyfields⟩ "}"
   # Array type definition.
   | "array" "of" ⟨type-id⟩
   # Class definition (canonical form).
   | "class" [ "extends" ⟨type-id⟩ ] "{" ⟨classfields⟩ "}"
⟨tyfields⟩ ::= [ id ":" ⟨type-id⟩ { "," id ":" ⟨type-id⟩ } ]
⟨type-id⟩ ::= id

⟨op⟩ ::= "+" | "−" | "*" | "/" | "=" | "<>" | ">" | "<" | ">=" | "<=" | "&" | "|"
```

Les priorités des opérateurs binaires (op), de la plus haute à la plus basse, sont comme suit :

```
* /
+ -
>= <= = <> < >
&
|
```

Les opérateurs de comparaison (<, <=, =, <>, >, >=) ne sont pas associatifs. Tous les autres opérateurs listés dans op sont associatifs à gauche.

B Classes de la syntaxe abstraite du langage Tiger

Voici une copie du fichier 'src/ast/README' fourni avec le code de tc.

```
Tiger Abstract Syntax Tree nodes with their principal members.
Incomplete classes are tagged with a '*'.

/Ast/               (Location location)
  /Dec/             (symbol name)
    FunctionDec     (VarDecs formals, NameTy result, Exp body)
      MethodDec     ()
    TypeDec         (Ty ty)
    VarDec          (NameTy type_name, Exp init)

  /Exp/             ()
*   /Var/
      CastVar       (Var var, Ty ty)
*     FieldVar
      SimpleVar     (symbol name)
      SubscriptVar  (Var var, Exp index)

*   ArrayExp
*   AssignExp
*   BreakExp
*   CallExp
*     MethodCallExp
    CastExp         (Exp exp, Ty ty)
    ForExp          (VarDec vardec, Exp hi, Exp body)
*   IfExp
    IntExp          (int value)
*   LetExp
    MetavarExp      ()
    NilExp          ()
*   ObjectExp
    OpExp           (Exp left, Oper oper, Exp right)
*   RecordExp
*   SeqExp
*   StringExp
    WhileExp        (Exp test, Exp body)

  /Ty/              ()
    ArrayTy         (NameTy base_type)
    ClassTy         (NameTy super, DecsList decs)
    NameTy          (symbol name)
*   RecordTy

  DecsList          (decs_type decs)

  Field             (symbol name, NameTy type_name)

  FieldInit         (symbol name, Exp init)
```