Correction du partiel CMP1 de S2

EPITA – AppIng1 promotion 2015
Notes de cours manuscrites autorisées
Machines (calculatrices, ordinateurs, tablettes, téléphones) interdites.

Juin 2013 (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 12 pages. Les pages 1–9 contiennent l'épreuve et son corrigé. Les pages 10–12 sont dédiées aux annexes.

1 Généralités

1. Qu'appelle-t-on la partie frontale d'un compilateur ?

   Correction: *Front end* : tout ce qui dépend du langage source.

2. Rappelez quelles sont les cinq phases de la partie frontale d'un compilateur comme `tc`.

   Correction: (i) scanner, (ii) parser et construction de l'AST, (iii) liaison des noms, (iv) typage, (v) traduction vers une représentation intermédiaire.

3. Qu'est-ce que BNF ?

   Correction: Backus-Naur Form : un langage pour décrire des grammaires hors contextes.

   Best-of:
   – Backus Norme Form
   – Backus Norm Formal

4. Quelle différence faites-vous entre un arbre de dérivation (*parse tree*) et un arbre de syntaxe abstraite (*Abstract Syntax Tree*) ?

   Correction: L'arbre de dérivation ou *parse tree* est un produit direct du parser, et reproduit fidèlement la syntaxe concrète du programme en entrée. Un arbre de syntaxe abstraite ou Abstract Syntax Tree (AST) est une représentation synthétique du programme, débarrassée des artefacts de la syntaxe concrète.

Correction: (suite) On trouvera donc dans un arbre de dérivation des éléments qui sont absents de la syntaxe abstraite :
- tous les non terminaux intermédiaires faisant apparaître toutes les productions utilisées lors de l'analyse syntaxique (par ex. : `program` → `exp` → `INT`) ;
- la ponctuation : virgules, points-virgules (séparateurs ou terminateurs), deux-points, parenthèses, crochets, etc.
- le sucre syntaxique, éventuellement (partiellement) supprimé dans l'AST : opérateurs de logique booléens, `if-then` sans `else`, moins unaire, etc.
- le sel syntaxique, c'est-à-dire des éléments de syntaxe qui pourraient ne pas faire partie de l'expression d'un programme et qui permettent (essentiellement) à l'utilisateur de mieux repérer ses erreurs et aux outils de l'aider dans cette tâche. Les mots-clefs terminateurs du Bourne Shell (`fi`, `done`, `esac`, etc.) remplissent cette fonction (en sus de simplifier l'analyse du langage).

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 ~~t~~ `f` et aux fonctions `a` et `m` ?

   Correction: Il y avait une coquille dans l'énoncé originel : on voulait bien sûr parler du type `f` (le seul du programme), non du type `t` (erronément mentionné).

   Le type `f` représente un type fraction ; les champs `n` et `d` représentent respectivement le numérateur et le dénominateur du quotient ; les fonctions `a` et `m` implémentent resp. l'addition et la multiplication de deux fractions.
   Il s'agit bien entendu d'une implémentation très naïve, car ces fractions ne sont pas réduites ; par exemple, le programme ci-dessus calcule l'opération $\left(\frac{1}{2}+\frac{4}{3}\right)\times\frac{4}{3}$ et donne $\frac{44}{18}$, ce qui est juste mais peu élégant, car on aurait plutôt voulu $\frac{22}{9}$.
   Enfin, le cas de la division par zéro (dénominateur nul) n'est pas traité.

   Best-of: `f` est un type et `a`, `m` sont des fonctions.

2. Quel rôle joue la fonction `f` de la ligne 3 ?

   Correction: Celui de *constructeur* (externe), puisque cette routine sert à construire des valeurs de type `f`.

   Best-of: La fonction `f` joue le rôle de la fonction identité.

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 ?

   Correction: Parce que les noms de types, de fonctions mais aussi de variables vivent dans des espaces de noms différents en Tiger, ce qui autorise des individus homonymes à condition qu'ils appartiennent à des genres (type, fonction, variable) différents.

   Je reconnais que cette question n'était pas très bien posée ; la formulation ci-dessous eût été plus claire :

   > Pour quelle raison le compilateur ne confond-t-il pas dans ce programme les occurrences du *nom de type* `f` (défini à la ligne 2) et celles du *nom de fonction* `f` (définie à la ligne 3) ?

   J'ai mis des points pour les réponses intelligentes.

4. Dans l'expression de la ligne 9, la première occurrence de `p` est une *l-value*. Que signifie ce terme ?

   Correction: Une l-value (*left-value*) est une valeur qui se situe à gauche d'une affectation. Par extension (dans les langages qui offrent ce service), c'est une valeur dont l'utilisateur peut prendre l'adresse.

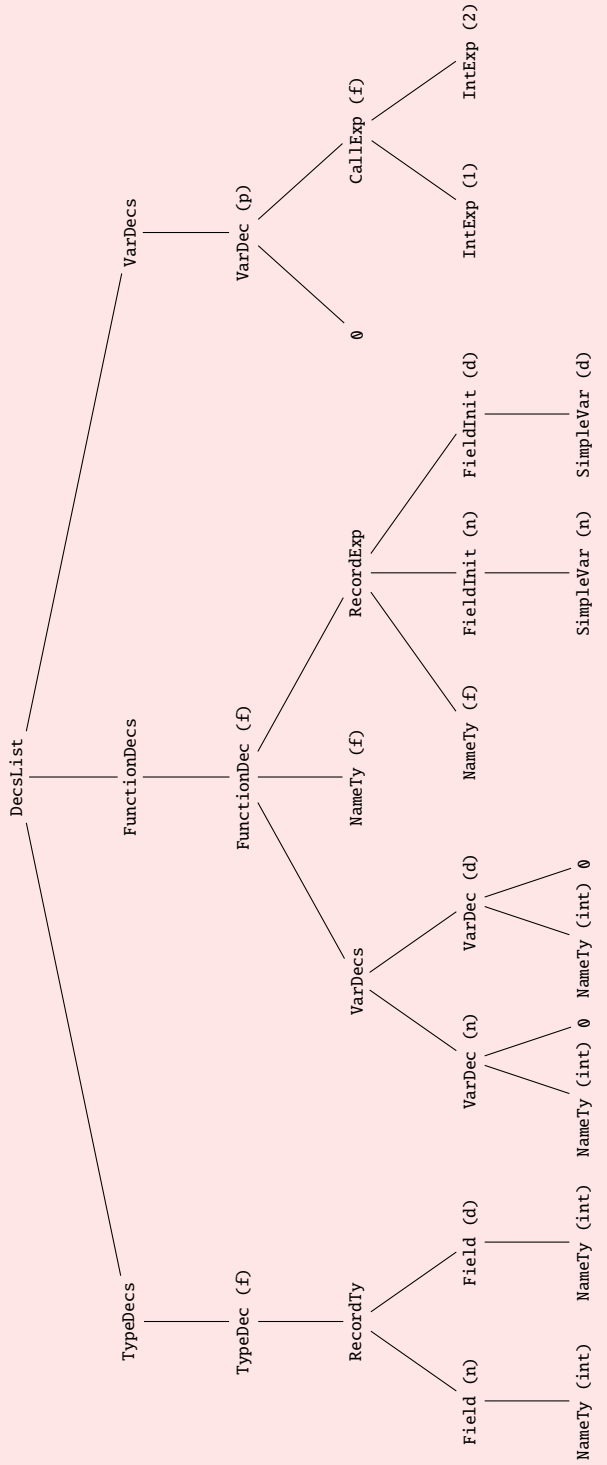
   Best-of:
   - Ce terme signifie "litteral value".
   - Le terme 'l-value' signifie variable locale.
   - La l-value correspond au nom de la fonction appelée.
   - Une l-value est le nom d'une variable, d'une fonction ou d'un type.
   - Cela signifie que la première occurrence de `p` est une valeur à gauche d'un élément terminal.

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.

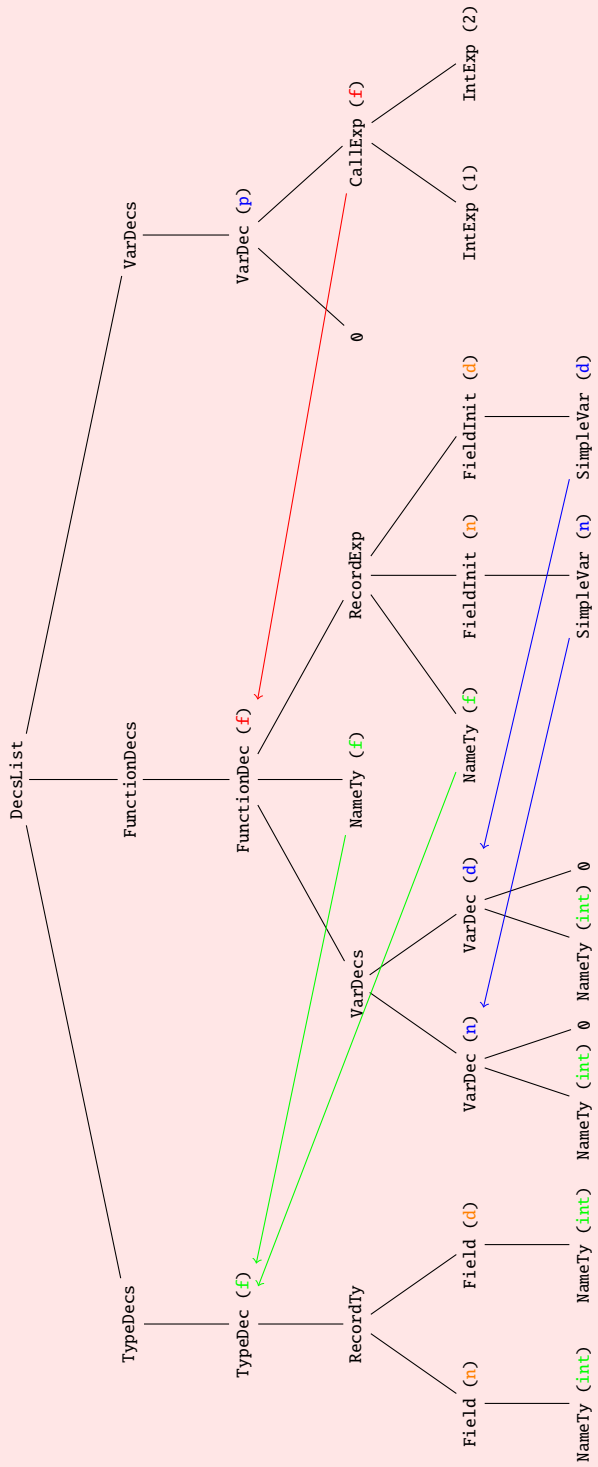
Correction:



Best-of: *Parmi les réponses proposées, j'ai trouvé un arbre curieux contenant une multitude de nœuds de type* `StringExp`*, alors qu'il n'y avait aucune chaîne de caractères littérale dans le programme à traiter (ni dans le programme original, d'ailleurs). . .*

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).

Correction:



Correction: (suite) On notera que les nœuds `'NameTy (int)'` n'ont pas de cible matérialisée, puisque c'est la sémantique du langage qui fournit cette liaison « par défaut ».
Enfin, les champs d'enregistrements ne sont pas liés ici, car cette opération nécessite des informations de typage.

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.

Correction: La question était un peu imprécise (mais j'en ai tenu compte dans la notation) : on souhaite ici une fonction de comparaison au sens de l'égalité (=). Par « intelligente », on entendait une comparaison des quotients, non des couples numérateurs/dénominateur :

```
function eq (x : f, y : f) : int = x.n * y.d = y.n * x.d
```

Best-of:

```
"comparaison"  return parse::parser::make_COMPARAISON(tp.location_);
```

Attention à ne pas confondre « ajouter au programme *Tiger (de l'exercice) » et « ajouter au* langage *Tiger » !*

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` ?

Correction: À garantir que toute destruction d'arbre libère correctement l'intégralité des données allouées dynamiquement et tenues par les nœuds de cet arbre. Le fait de rendre `virtual` le destructeur de `Exp` assure que la sélection du destructeur à l'exécution sera dynamique. En particulier, cela garantit qu'une instance de `Op` vue comme une `Exp` sera bien détruite comme une `Op`.

2. À quoi servent les lignes 5 à 8 dans la classe `Exp` ?

   Correction: À rendre `Exp` non instanciable, en empêchant sa construction ou sa copie depuis l'extérieur de la hiérarchie de classe, ceci afin de rendre explicite le caractère abstrait de cette classe.

3. Le code C++ ci-dessus est volontairement court par souci de simplicité et pourrait être amélioré. Quels changements apporteriez-vous ?

   Correction:
   – On pourrait masquer les données des classes en utilisant `private`/`protected`, ainsi que `class` (au lieu de `struct`), et en ajoutant des accesseurs (au moins en lecture seule).
   – L'attribut `oper` de la classe `Op` (ligne 41) sert à stocker un opérateur (binaire) sous la forme d'un `char`. Ce choix de conception est simple mais peu robuste. Il vaudrait mieux le remplacer par une énumération (`enum`) ou mieux : une énumération fortement typée (`enum class`) du C++ 2011.

   Best-of: Ecrire des templates.

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 ?

   Correction: Il s'agit des *multiméthodes*, qui permettent l'aiguillage (*dispatch*) multiple, basé sur les types dynamiques de un ou plusieurs argument(s). Les méthodes (virtuelles) « simples » n'autorisent en effet qu'un dispatch dynamique sur un argument (la cible de la méthode).

   Best-of:
   – Le parse tree
   – PrettyPrinter

5. Écrivez le code de la classe abstraite `Visitor` dont dériveront toutes les classes de visiteurs parcourant nos ASTs.

   Correction:

```
class Visitor
{
public:
  virtual void operator()(const Int& exp) = 0;
  virtual void operator()(const Op& exp) = 0;
};
```

   Note : par soucis de simplicité, on n'a traité que le cas des parcours en lecture seule (les nœuds visités passés par référence constante).

6. Que faut-il ajouter aux classes `Exp`, `Int` et `Op` pour qu'elle puissent coopérer avec un `Visitor` ?

Correction: Une méthode virtuelle `accept`, abstraite dans `Exp`, concrète dans `Int` et `Op` :

```
7  struct Exp {
8    virtual void accept(Visitor& v) = 0;  // ...
9  };
10 struct Int {
11   virtual void accept(Visitor& v) { v(*this); }  // ...
12 };
13 struct Op {
14   virtual void accept(Visitor& v) { v(*this); }  // ...
15 };
```

Best-of:
- `extends Visitor`
- `implements 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.

Correction:

```
class Evaluator : public Visitor
{
public:
  virtual int eval(const Exp& e) {
    e.accept(*this); return value;
  }
  virtual void operator()(const Exp& e) {
    e.accept(*this);
  }
  virtual void operator()(const Op& e) {
    switch (oper)
      {
      case '+': value_ = eval(e.lhs_) + eval(e.rhs_); break
      case '-': value_ = eval(e.lhs_) - eval(e.rhs_); break
      case '*': value_ = eval(e.lhs_) * eval(e.rhs_); break
      case '/': value_ = eval(e.lhs_) / eval(e.rhs_); break
      default: abort();
      }
  }
  virtual void operator()(const Int& e) {
    value_ = e.val;
  }
  int value_get() const {
    return value_;
  }

private:
  int value_;
};
```

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 ?

Correction: Il y a au moins deux solutions possibles :
- Manipuler les nœuds fils via des « pointeurs intelligents » (*smart pointers*), par exemple via `std::shared_ptr<T>` (ou `std::unique_ptr<T>`, si on n'a pas besoin de partage).
- Utiliser un « ramasse-miettes » (*garbage collector*) pour manipuler ces données. Pour le C++ et le C, il existe par exemple le garbage collector Boehm-Demers-Weiser, qui fournit cette fonctionnalité sous forme de bibliothèque. Il est aussi possible d'obtenir ce service via une variante du langage, comme « Managed C++ » de Microsoft ou encore C++/CLI, visant tous les deux la plateforme .NET et permettant d'instancier des objets sous la coupe du garbage collector .NET.

Dans les deux cas, on supprimera les appels à `delete` dans les destructeurs des nœuds d'ASTs, puisque la gestion mémoire sera prise en charge par le mécanisme choisi.

Une copie suggère de surcharger le destructeur afin de ne supprimer que le nœud courant ; l'idée est intéressante, mais n'est réalisable (en tout cas pas directement) car la surcharge de destructeurs n'existe pas en C++.

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)
```