Correction du Compilation, Langages de Programmation

EPITA – Promo 2006 – **Tous documents autorisés** *

Septembre 2004 (1h30)

Le sujet a été écrit par Akim Demaille.

Une copie synthétique, bien orthographiée, avec une présentation claire des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur.

Tout résultat se doit d'être justifié ; une argumentation informelle mais convaincante, même courte, sera souvent suffisante.

Le lecteur sera bien avisé de parcourir l'ensemble du sujet avant de l'attaquer.

1 Contrôle de Connaissances : Piqûres de Rappel

1. Donner la représentation sagittale (i.e., graphique) d'un automate fini (éventuellement non-déterministe, avec ou sans transition spontanée) reconnaissant le langage des crochets imbriqués (i.e., $\{\varepsilon, [], [[]], ..., [^k]^k, ...\}$).

   Correction: C'est impossible (langage hors contexte, pas régulier). Ne pas le savoir est une des fautes les plus impardonnables que je puisse concevoir à la fin de cette année.

2. À quoi sert Bison ?

   Correction: À engendrer un parseur à partir d'une grammaire (LALR (1)).
   Ne pas me répondre que ça fabrique de scanner, ou lexer, ou que ça parse (même si c'est vrai qu'il parse : la grammaire !) etc.

   Best-of:

   bison est un tools.
   bison est un outil qui permet de construire des grammaires.
   bison sert à parser un texte.
   bison est un parseur. Il se programme en Yak.
   bison est un générateur de compilateur.
   bison sert à interpréter du code compilé avec flex.
   bison est un lexer.

3. Qu'est-ce qu'un objet fonction ?

   Correction: Question déjà donnée, et corrigée. Un objet qui dispose d'une méthode `operator()`, i.e., qui a une méthode dont l'invocation ressemble à l'appel d'une fonction.

*"Tout document autorisé" signifie que notes de cours, livres, annales, etc. sont consultables pendant l'épreuve. En cas de zèle des surveillants contacter le LRDE au 01 53 14 59 22.

Best-of:

Une fonction objet est une fonction qui se réfère à elle-même.
Un objet fonction est un objet qui possède dans ses attributs un pointeur sur fonction. Ainsi cet objet peut-être utilisé comme une fonction.

4. Qu'est-ce que la « portée » d'un identificateur (*scope*) ?

   Correction: La zone de programme depuis laquelle il est accessible.

   Best-of:

   La portée d'un scope ...

5. À quoi sert un environnement/table des symboles ?

   Correction: C'est une structure de donnée qui permet de faire le lien entre la définition et les références (utilisations) d'un identificateur. Par exemple, lors du contrôle des types d'une déclaration de variable, c'est dans une table des symboles qu'est stockée le type de ladite variable, de façon à contrôler que les références sont conformes à ce type.

6. Quelle différence notable y a-t-il entre une table des symboles et un tableau associatif (tel que `std::map`) ?

   Correction: La gestion des portées !

7. Donner, dans le langage de votre choix, un court programme justifiant le recourt d'un compilateur à une table des symboles plutôt qu'un tableau associatif.

   Correction: La question précédente l'a souligné, la différence marquante est la gestion des portées qui autorisent l'existence simultanée, sans conflit ni confusion, de plusieurs entités portant le même nom.
   Il s'agit donc de faire un programme avec deux entités (e.g., variables) portant le même nom, leur portée étant imbriquée.
   En C :

```
{
  int c;     /* c1 */
  {
    char *c; /* c2 */
  }
  1 - c;     /* c1 */
}
```

   En Tiger :

```
let var c := 1     /* c1 */
in
  let var c := "2" /* c2 */
  in
  end;
  1 - c            /* c1 */
end
```

   On pouvait aussi tout simplement répondre « voir le programme `goodfnredef.tig`, section 3.4 ».
   Ne seront pas acceptées les réponses données dans des langages à portée dynamique : Perl (sauf avec `my`), Sh, etc.

8. Qu'est-ce qu'un « traits » ?

Correction: Question déjà donnée, et corrigée. Une fonction dont les arguments sont des types. Elle est donc évaluée à la compilation, et permet d'apprendre des choses sur un type, comme par exemple :
– pour les types numériques, leurs caractéristiques (le max, min etc.).
– s'il est const
– sa version const
– sa version pas const
– s'il est pointeur
– ...

Best-of:

Un "trait" est une séquence d'états qui peut-être consécutivement exécuté durant l'exécution du programme.

Un trait c'est l'ensemble des scopes.

Un trait c'est quelque chose de magique qui ...

Un trait est lorsque l'on trace une ligne avec un stylo ou un crayon et que l'on n'est pas doué en orthographe. (Ce n'est pas drôle, désolé)

9. Expliquer pourquoi remplacer `l := (s; e)` par `s; l := e` n'est pas toujours correct (`l` désigne une l-value, `s` une instruction, et `e` une expression).

Correction: Cette question avait été incorrectement posée (mais correctement corrigée) dans le partiel précédent.
`a[i] := (i := i + 1; 42)` n'a pas le même sens que `(i := i + 1; a[i] := 42)`.

2 Contrôle de Compétences : Résolution de noms et Liaisons

Correction: L'idée directrice de cette question fait suite à la lecture de l'article « A Research C# Compiler », par David R. Hanson ; Todd A. Proebsting `http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2003-32`.
Cette idée, conjuguée à deux autres (un fichier Prelude pour les primitives comme en Haskell, et une annotation de l'AST par les types des expressions) devrait simplifier l'implémentation de `tc`, avec comme but principal l'alléger tc-5.

1. Pour chacune des phases de compilation suivantes, justifier si une table des symboles s'impose.
   a. L'analyse syntaxique.

      Correction: Non, le parsing n'est pas concerné par les portées. Il s'attache aux aspects hors-contexte, et les portées et les liaisons sont précisément des aspects sensibles au contexte.

   b. Le marquage des déclarations de variables non locales.

      Correction: Oui, bien sûr, puisque le caractère non local d'une variable s'établit par l'étude de ses utilisations : il faut donc pouvoir distinguer les définition et utilisation(s) d'une variable indépendemment des problèmes de portée d'identificateur.

   c. Le contrôle de type.

      Correction: C'est l'exemple le plus clair du besoin d'un environnement : deux variables peuvent exister simultanément, et avoir des types différents. Le contrôle de leurs utilisations nécessite des les distinguer. Voir la question 1.7.

   d. La traduction de l'AST en code intermédiaire.

      Correction: Puisque cette étape place les variables (dans des temporaires ou sur la pile) lors de leur déclarations, il faut bien savoir retrouver ces emplacements lors de l'utilisation de ces variables.

   e. L'allocation des registres.

      Correction: Ici les identificateurs sont à portée globale ! Un identificateur (de temporaire ou de registre) ne désigne qu'une et une seule entité. Pas de portée, donc pas de nécessité d'environnement.

On constate que les tables de symboles sont plusieurs fois utilisées, toujours pour rejoindre une définition (ou une information portée par elle) à partir d'une utilisation. C'est ce que l'on appelle la « résolution de nom ».

Étudions la possibilité de factoriser cette résolution de nom en annotant les utilisations (d'identificateurs) d'un pointeur vers leur définition. Ce pointeur se nomme la « liaison » (*binding*).

2. Quels nœuds de l'AST de Tiger définissent des identificateurs ?

   Correction: `VarDec`, `FunctionDec`, `TypeDec`. Il y aussi les `Field` qui introduisent les arguments de fonction, voir l'AST donné en section 4.

3. Dans un programme Tiger correct de syntaxe, quelle erreur purement liée au problème de noms (i.e., *pas* les conflits de type) peut exister dans la définition d'un identificateur ?

   Correction: Un identificateur ne peut pas être défini deux fois de suite pour une catégorie donnée (fonction, type, variable).

4. Quels sont les nœuds de l'AST de Tiger qui référencent (« utilisent ») des identificateurs ?

   Correction: `SimpleVar`, `CallExp`, `NameTy`.

5. Dans un programme Tiger correct de syntaxe (et de typage), quelle erreur peut exister dans l'utilisation d'un identificateur ? Comme pour la question 3, ne considérez pas les problèmes liés à des conflits de type, mais seulement ceux liés à la référence à un nom.

   Correction: Identificateur pas défini.

6. Dans un programme Tiger correct de syntaxe et de typage, se peut-il que l'on réfère une variable qui n'est pas définie dans l'AST ?

   Correction: Pas de variable prédéfinie.

7. En est-il de même pour les fonctions et types ?

   Correction: On a des types (int, string) et des fonctions primitives (print, print_int etc.). Il va bien falloir les gérer quelque part. Ça commence à devenir lourd : les primitives sont introduites pour le TypeVisitor, le BindVisitor et pour le TranslateVisitor : on va vouloir factoriser ça aussi. L'idéal pour simplifier serait que les primitives soient définies dans l'AST, d'où l'idée d'un Prélude à la Haskell.

8. Considérons une phase supplémentaire dans ce compilateur qui annote toutes ces utilisations. À quels noeuds de l'AST doit-elle prêter attention ? Attention, il n'y a pas que les définitions et utilisations d'identificateurs qui interviennent.

   Correction: Bien sûr il y a les définitions/utilisations de variables, fonctions et types (VarDec/SimpleVar, FunctionDec/CallExp, TypeDec/NameTy) ainsi que Field qui sert à définir les arguments formels d'une fonction. Mais en plus il faut prêter attention aux portées ! Elles sont introduites par LetExp, ForExp, ainsi que FunctionDec.

9. Décrire l'implémentation d'une phase supplémentaire dans un compilateur tel que `tc` qui annote toutes ces utilisations. On ne négligera pas de décrire les structures de données qui seront impliquées. On n'oubliera pas de préciser où s'insère cette phase, et quels sont les cas particuliers à gérer (cf. les questions précédentes).

   Correction: Il s'agira bien sûr d'un visiteur d'AST qui passe avant les EscapeVisitor, TypeVisitor et TranslateVisitor. Il lui faudra trois tables de symboles (variable, type, fonction) qui associent toutes à un identificateur un pointeur sur le nœud qui l'a défini. Les nœuds de définition servent à charger ces tables, les nœuds d'utilisation les consultent pour obtenir leur annotation. En cas d'erreur (voir les questions précédentes) les reporter.
   Enfin, les nœuds liés aux portées doivent les ouvrir et les fermer comme il convient : LetExp, ForExp, FunctionDec.

10. Comment permette à l'utilisateur de vérifier cette annotation ?

    Correction: Une option `--bindings-display` qui appelle un PrintVisitor qui commente les définitions par leur adresse, et les utilisations par l'adresse de la définition à laquelle elles réfèrent. C'est tout-à-fait comparable à `--escapes-display`.

    Best-of:

    En sortant du texte en couleur dans le PrintVisitor, avec la même couleur pour une définition et les utilisations d'un même identificateur.

    En faisant des applications.

11. Quelles simplifications du TypeVisitor la présence d'une telle phase permet-elle ?

    Correction:
    – Une partie des erreurs (celles liées aux identificateurs) n'est plus à sa charge.
    – Plus de gestion des portées.
    – Donc plus de tables : de simples `std\:\:map` suffisent.

12. Quel est l'intérêt d'une telle factorisation ?

    Correction:
    – Les phases suivantes (EscapeVisitor, TypeVisitor, TranslateVisitor), plus complexes, sont simplifiées : plus de portée à gérer. Plus besoin de tables de symbole, des tableaux associatifs suffisent.
    – Facilite le débogage : un typage incorrect du fait d'une liaison mal faite sera plus facile à diagnostiquer.
    – Moins de risque d'erreur ! Beaucoup de groupes avaient correctement implémenté les espaces de noms dans l'un des trois visiteurs (Escape, Type, Translate), mais pas dans tous. Ici, c'est impossible.

13. Le langage Tiger décrit par Appel n'a que deux espaces de noms : types d'une part, et variables et fonctions d'autre part (une portée ne peut avoir une fonction et une variable de même nom). Est-ce que cela compromet cette approche ? Si non, pourquoi ? Si oui, que peut-on sauver ?

    Correction: Ça ne change rien : ce qui est critique est de pouvoir apparier un nom de fonction avec une déclaration de fonction, et idem pour les variables. Cet aspect n'est pas lié aux espaces de noms, mais à la propreté de la syntaxe du langage.

14. L'introduction de la surcharge de fonctions (*overloading*) compromet-elle cette approche ? Si non, pourquoi ? Si oui, que peut-on en sauver.

    Correction: Là, c'est beaucoup plus problématique, puisque pour identifier la déclaration de fonction à laquelle un appel fait référence, il faut avoir résolu le type des arguments. Autrement dit, le typage devient partie intégrale de l'identification : impossible alors de séparer liaison de typage.
    Que peut-on préserver ?
    Remarquons qu'on peut toujours établir la liaison pour les types et les variables. On peut ajouter un drapeau au `BindVisitor` qui dise si l'on veut, ou pas, lier les fonctions.
    Puis le `TypeVisitor`, utilisant les liaisons, pourrait être sous-classé par un `OverloadedTypeVisitor` qui s'occupe de tous les noms en relations avec les fonctions, et qui gère, à nouveau, les portées... Du coup, il lui faut aussi gérer `LetExp`.
    Roland Levillain propose une idée intéressante qui dispense cet `OverloadedTypeVisitor` de gérer portée (et `LetExp`) : le `BindVisitor` peut annoter les `CallExp` par l'*ensemble* des `FunctionDec` possibles. Reste à la charge de l'OverloadedTypeVisitor de choisir le bon candidat dans cet ensemble.

15. La question 7 montre que le TypeVisitor, le TranslateVisitor et cette phase supplémentaire ont un point commun aisément factorisable. Lequel, et comment factoriser *et* simplifier ?

    Correction: La gestion des primitives ! On a très envie d'un prélude, i.e., une sorte de fichier include qui définisse toutes les primitives. Là, beaucoup de code serait simplifié.
    Une autre réponse serait d'avoir un fichier supplémentaire qui gère/décrive les primitives. On a bien factorisation, mais la simplification est plus discutable.

16. Il existe dans le langage Tiger une autre paire de définition/utilisation, plus discrète parce qu'elle ne porte pas sur un identificateur explicite. Laquelle ? Suggéreriez-vous un changement par rapport à l'implémentation actuelle ?

    Correction: Les boucles (while et for) définissent ce que le break veut utiliser. Comme pour les cas basés sur des identificateurs, il semble souhaitable d'installer un lien depuis le break vers sa boucle dans le BindVisitor. Qui du même coup devient responsable de diagnostiquer les break sans boucle.

3 Quelques programmes Tiger

Quelques programmes pour aider ceux qui n'ont pas implémenté de compilateur.

3.1 `three.tig` : Trois espaces de nom

```
let type foo = string
    var foo : foo := "foo"
    function foo (foo : foo) : foo = foo
in
  print (foo (foo))
end
```

Correct à l'EPITA, incorrect pour Appel : la fonction `foo` masque la variable `foo`, si bien que l'appel dans le `print` est incorrect.

3.2 `fnundef.tig` : Fonction non définie

```
foo ()
```

Incorrect : `foo` non définie.

3.3 `badfnredef.tig` : Redéfinition de fonction invalide

```
let function foo () = ()
    function foo () = ()
in
  foo ()
end
```

Incorrect : redéfinition de `foo`.

3.4 `goodfnredef.tig` : Redéfinition de fonction valide

```
let function foo () = ()
in
  let function foo () = ()
  in
    foo ()
  end;
  foo ()
end
```

Redéfinition correcte de `foo`.

3.5 `fnoverload.tig` : Surcharge de fonctions

```
let function add (a : int, b : int) : int = a + b
    function add (a : string, b : string) : string = concat (a, b)
in
  print_int (add (1, 2));
  print (add ("1", "2"))
end
```

Incorrect en Tiger standard, mais valide en Tiger étendu avec la surcharge de fonction.

4 AST de Tiger

À titre d'information, voici l'AST de l'implémentation de Tiger de l'EPITA. Il est simplifié des « détails » tels que les listes de déclarations de LetExp, `const`, `&`, `*` et autre `::`.

```
/Ast/           (Location loc)

  /Exp/         ()
    ArrayExp    (NameTy type, Exp size, Exp init)
    RecordExp   (NameTy type, list<FieldExp> fieldvars)

    IntExp      (int value)
    StringExp   (string value)
    NilExp      ()

    AssignExp   (Var lvalue, Exp exp)
    CallExp     (Symbol func, list<Exp> args)
    OpExp       (Exp left, Oper oper, Exp right)

    LetExp      (list<Dec> decs, Exp body)
    SeqExp      (list<Exp> body)
    IfExp       (Exp test, Exp then, Exp else)
    ForExp      (VarDec vardec, Exp hi, Exp body)
    WhileExp    (Exp test, Exp body)
    BreakExp    ()

    /Var/          ()
      SimpleVar    (Symbol var)
      FieldVar     (Var var, Symbol field)
      SubscriptVar (Var var, Exp index)

  /Ty/          ()
    NameTy      (Symbol name)
    ArrayTy     (NameTy basetype)
    RecordTy    (list<Field> fields)

  /Dec/         (Symbol name)
    FunctionDec (list<Fields> formals, NameTy return_type, Exp body)
    TypeDec     (Ty type)
    VarDec      (NameTy type, Exp init)

  Field         (Symbol name, NameTy type)
  FieldExp      (Symbol field, Exp init)
```

Voici l'AST du programme 3.1. Afin de simplifier les notations, les `symbol::Symbol` sont représentés `<ainsi>`, et les `std::list` le sont `[comme, cela]`.

```
LetExp (
   [
    TypeDec     (<foo>, NameTy (<foo>)),
    VarDec      (<foo>, NameTy (<foo>), StringExp ("foo")),
    FunctionDec (<foo>, NameTy (<foo>), [Field (<foo>, NameTy (<foo>))],
                 SimpleVar (<foo>))
   ],
   CallExp (<print>, [CallExp (<foo>, [SimpleVar (<foo>)])]))
```