Construction des Compilateurs

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

Septembre 2004 (1h30)

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., $\{\varepsilon, [], [[]], ..., [^k]^k, ...\}$).
2. À quoi sert Bison ?
3. Qu'est-ce qu'un objet fonction ?
4. Qu'est-ce que la « portée » d'un identificateur (*scope*) ?
5. À quoi sert un environnement/table des symboles ?
6. Quelle différence notable y a-t-il entre une table des symboles et un tableau associatif (tel que `std::map`) ?
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.
8. Qu'est-ce qu'un « traits » ?
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).

*"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.

2 Contrôle de Compétences : Résolution de noms et Liaisons

1. Pour chacune des phases de compilation suivantes, justifier si une table des symboles s'impose.
   a. L'analyse syntaxique.
   b. Le marquage des déclarations de variables non locales.
   c. Le contrôle de type.
   d. La traduction de l'AST en code intermédiaire.
   e. L'allocation des registres.

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 ?
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 ?
4. Quels sont les nœuds de l'AST de Tiger qui référencent (« utilisent ») des identificateurs ?
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.
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 ?
7. En est-il de même pour les fonctions et types ?
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.
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).
10. Comment permette à l'utilisateur de vérifier cette annotation ?
11. Quelles simplifications du TypeVisitor la présence d'une telle phase permet-elle ?
12. Quel est l'intérêt d'une telle factorisation ?
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 ?
14. L'introduction de la surcharge de fonctions (*overloading*) compromet-elle cette approche ? Si non, pourquoi ? Si oui, que peut-on en sauver.
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 ?
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 ?

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>)])]))
```