Compilateur Mini Ada

Lucas Willems

17 janvier 2017

1 Choix techniques

1.1 Analyseur lexical

Pour réduire le nombre d'états de l'automate, un seul cas a été utilisé pour les identifieurs. Les mots clés sont enregistrés dans une liste de couples d'identifieur et de lexeme.

Cependant, des cas spécifiques ont été ajoutés pour certaines suites de caractères (`..` ou `:=` par exemple) qui ne peuvent comprendre de blancs entre les caractères.

1.2 Analyseur syntaxique

Pour permettre la localisation des erreurs, les déclarations, les instructions et les expressions ont des types comprenant un champs pour la description de l'élément et un champs pour la position (récupérée grâce à `($startpos, $endpos)`) comme présenté en classe.

De plus, plusieurs fonctions de Menhir comme `*`, `+`, `?`, `separated_nonempty_list`... ont été utilisées.

1.3 Typeur

Le principal choix technique pour le typeur s'est trouvé au niveau du choix de la structure de donnée pour stocker la déclaration de variables, de types, de fonctions et de procédures, et au niveau du choix de la représentation des types.

Initialement, j'avais opté pour représenter les types avec le type `typna` (pour type naïf car il ne stocke que l'identifieur) défini par :

```
TNArec of string | TNAaccOFrec of string | Ttypenull
```

et de stocker, dans une variable `env`, un 4-uplet de maps :

- un map `Vmap` pour les variables associant, à un `ident`, un couple `mode * typna`
- un map `Tmap` pour les types associant, à un `ident`, soit une liste de couples `(ident * typna) list`, soit un lien vers un autre type enregistré, soit un type indéfini.
- un map `Pmap` pour les procédures associant, à un `ident`, une liste de couples `(mode * typna) list`
- un map `Fmap` pour les fonctions associant, à un `ident`, une liste de couples `(mode * typna) list` et un `typna` de retour

Cependant, après avoir implémenté les règles de typage des déclarations, instructions et expressions, j'ai lu les vérifications supplémentaires et lu que "toutes les déclarations d'un même niveau doivent porter des noms différents", ce que je n'avais pas du tout pris en compte.

Par conséquent, j'ai modifié ma structure de donnée en rajoutant un map `Nmap` associant, à un `ident`, un couple `int * string` représentant le niveau de la déclaration d'un identifieur et la catégorie (variable, type, fonction ou procédure). Avec cette structure, j'ai réussi à passer tous vos tests.

Toutefois, après avoir testé les programmes mis sur Github par des élèves de la classe, j'ai découvert deux problèmes avec ma structure de donnée :

- D'autres types `integer`, `character` et `boolean` ne pouvaient pas être définis.
- Le "shadowing" de type n'était pas pris en compte, les types, variables, fonctions, procédures pouvant être écrasés à chaque nouveau niveau.

Pour illustrer le deuxième point, le programme suivant était alors mal typé pour mon compilateur :

```
with Ada.Text_IO; use Ada.Text_IO;
procedure niveaux_shadow is
   type r is record a : boolean; end record;
   procedure f is
      y : r;
      type r is record b : integer; end record;
   begin
      -- y fait reference a un type r shadow a ce niveau
      y.a := true;
   end;
begin New_Line; End;
```

Pour corriger les deux problèmes, j'ai totalement modifié ma structure de donnée et ma représentation des types. Je suis passé de 4 maps à une liste de couples `key * def` où :

- `key` est un couple `ident * int` correspondant à un identifieur et son niveau
- `def` est la définition d'une variable, d'une fonction, d'une procédure et d'un type (qui peut être, en plus, défini comme `primitive`)

Avec cette structure, les types des niveaux précédents sont conservés et il est possible d'y faire référence. Au niveau des types, ceux-ci sont maintenant représentés avec le type `typpr` (pour type précis car il stocke la profondeur et l'identifieur) suivant :

`TPRrec of key | TPRaccOFrec of key | Ttypenull`

Ainsi, les types "record" et "access" stockent à la fois le nom et le niveau du type, résolvant le problème de "shadowing". Quant à la rédéfinition des types `integer`, `character` et `boolean`, j'ai simplement déclaré des types `primitive` de clés `(integer, 0)`, `(character, 0)` et `(boolean, 0)`.

1.4 Production de code

Le principal choix technique s'est trouvé au niveau du choix de l'arbre de syntaxe abstraite. Lors de la réalisation du typeur, n'étant pas du tout familier avec le langage assembleur, j'avais beaucoup de mal à imaginer ce qu'il était bon de garder pour la production de code.

Dans un premier temps, n'ayant pas trop d'idées, j'ai gardé l'arbre de syntaxe abstraite du typeur. J'ai seulement modifié la partie déclaration en suivant celle présentée dans le cours 8, contenant une déclaration pour les variables, pour les fonctions et pour les procédures.

Puis, à forcer de coder, j'ai compris que la seule information utile à conserver d'un type est sa taille. Ainsi, j'ai modifié mon typeur pour qu'il construise un nouvel arbre de syntaxe abstraite ne contenant plus aucune trace de l'environnement du typeur et plus les positions obtenues par le lexeur :

```
type expr =
        Eint of int
        | Echar of char
        | Etrue
        | Efalse
        | Enull
        | Eaddr of expr (* pour le passage par parametre comme dans le cours 8 *)
        | Eacces of acces
        | Ebinop of expr * binop * expr
        | Enot of expr
        | Enew of int (* 'int' correspond a la taille du type associe *)
        | Efunction of string * expr list
        | Echarval of expr

and acces =
        Aident of string
        | Afield of expr * (int * int * int)
```

```
        (* 'int * int * int' correspond :
                - au nombre de pointeur a suivre
                - a la position du champs dans le type
                - a la taille du champs *)

type instr =
        Iassign of acces * expr
        | Iif of (expr * instr) list
        | Iwhile of expr * instr
        | Ifor of string * bool * expr * expr * instr
        | Iprocedure of string * expr list
        | Ireturn of expr option
        | Iblock of instr list

type decl =
        Dvar of string list * int * expr option
        | Dpf of (string * int) * (string * bool * int) list * decl list * instr
        | Dtype (* N'apparait pas dans l'arbre *)
```

Ce nouvel arbre de syntaxe abstraite a permis de simplifier mon code, de le rendre plus clair et d'enlever plusieurs problèmes. Toutefois, quelques petits problèmes ont persisté, comme par exemple, les deux problèmes suivants, avec les solutions trouvées :

- Comment renvoyer la valeur d'une fonction (si la taille du type de retour est grande) ?
  Ajouter un premier paramètre (que j'ai nommé `.return` pour ne pas avoir de collision avec un nom possible de paramètre) où stocker la valeur de retour. Ainsi, un `return` correspond alors à une affectation, et de cette manière, les fonctions peuvent être considérées comme des procédures.
- Comment stocker la valeur de l'indice dans une boucle `for` ?
  Agrandir la taille à allouer pour une procédure / fonction lorsqu'une boucle `for` est rencontrée.

2 Tests supplémentaires

Pour vérifier la correction de mon typeur, j'ai d'abord utilisé vos tests (présents dans le dossier `tests-prof`) puis ceux mis en ligne (version de mardi 07/12/16) par Florentin, Martin, Alexis et Lionel (présents dans le dossier `tests-eleves`) qui relèvent plusieurs subtilités du language.