PetitGo

– projet de compilation, première partie –

Maxime FLIN

7 décembre 2019

1 Manuel d'utilisation

1.1 Organisation du dépôt

Le compilateur a été réalisé en `Ocaml` avec `Ocamllex` et `Menhir`.
Les principaux fichiers du projet sont dans le dossier `src` :

main Fichier d'entrée du programme

ast Arbre de syntaxe abstraite utilisé construit pendant le parsage

config Fonctions d'utilitées générales et les paramètres passé en ligne de commande (voir section 1.3)

error Ensemble de fonctions permettant de lever des erreurs et de les afficher proprement (voir section 3.2)

graph Implémentation de la recherche de cycle dans un graphe

lexer et parser Fichiers `Ocamllex` et `Menhir`

typer Fonctions de typage

On trouve aussi dans le dossier `tests` les fichiers de test et le script `tester.sh`.

1.2 Compiler le compilateur

Pour compiler le projet, j'ai fait le choix d'utiliser `dune`. Il suffit donc d'entrer les commandes suivantes

```
git clone https://git.eleves.ens.fr/mflin/petitgo.git
cd PetitGo
dune build src/main.exe
```

1.3 Compiler avec le compilateur

L'exécutable est assez simple à utiliser. Pour compiler un fichier `PetitGo` il faut le passer en argument le nom de ce fichier à l'exécutable.
Le programme peut aussi prendre des options en fonction des besoins.

-v Mode verbeux.

-parse-only L'éxécution s'arrête après le parsage.

-type-only L'éxécution s'arrête après le typage.

Ainsi, pour compiler le fichier de test `tests/exec/abr.go` il suffit d'entrer la commande suivante

```
main.exe tests/exec/abr.go
```

2 Implémentation du sujet

2.1 L'arbre de syntaxe abstraite

L'arbre de syntaxe abstrait est représenté par les structures décrites dans le fichier `ast.ml`. Les deux types les plus importants sont les types représentants les expressions et les instructions

```
and expr =
  Enil
| Eident  of string
| Eint    of int64
| Estring of string
| Ebool   of bool
| Etuple  of expr loc list
| Eattr   of expr loc * ident loc
| Ecall   of (ident loc option) * ident loc * (expr loc list)
| Eunop   of unop * expr loc
| Ebinop  of binop * expr loc * expr loc

and instruction =
  Inop
| Iexpr   of expr loc
| Iasgn   of expr loc * expr loc
| Iblock  of instruction list
| Idecl   of ident loc list * ty loc option * expr loc option
| Ireturn of expr loc
| Ifor    of expr loc * instruction
| Iif     of expr loc * instruction * instruction
```

On remarquera le constructeur d'expression `Etuple` qui est simplement une liste d'expression. Le langage `PetitGo` n'a pas de tuple comme `Ocaml`, j'ai tout de même fait le choix d'ajouter ce constructeur pour uniformiser le type dans les autres déclarations. Ce choix est questionnable et présenta des désaventages lors du typage, rien d'insurmontable pour autant.

On notera de plus la présence du type α `loc` plusieurs fois dans les déclarations. Ce type est juste un enregistrement qui permet de se souvenir de la position des éléments retenu dans le fichier.

```
type 'a loc = { v : 'a; position : position }
```

2.2 Le lexer et le parser

Il n'y a pas grand chose à dire sur ces parties du projets, elles sont une implémentation plus ou moins directe de la syntaxe décrite dans le sujet.

Le point virgule automatique en fin de ligne est géré dans le lexer. J'ai utilisé une référence `is_semi` indiquant s'il faut insérer un point virgule après le retour à la ligne ; une fonction `tok` pour la mettre un jour ; et une fonction `eol` pour insérer le retour à la ligne si besoin.

```
let is_semi = ref false

let eol f lexbuf =
  Lexing.new_line lexbuf;
  if !is_semi
  then begin is_semi := false; SEMI end
  else f lexbuf

let tok t =
  let _ =
    match t with
    | IDENT _ | INT _ | STRING _ | TRUE | FALSE
      | NIL | RETURN | INCR | DECR | RPAR | END ->
       is_semi := true
    | _ -> is_semi := false
  in
  t
```

J'ai légèrement étendu la syntaxe du `PetitGo` pour permettre d'utiliser des fonctions d'autres packages que `main` et `fmt` (voir section 3.1). Plus précisement les modifications sont les suivantes

```
<type> ::= <ident> | <ident> . <ident> | * <type>
<expr> ::= <ident> . <ident> (<expr,>*) | ...
```

2.3 Le typage

L'implémentation du typeur a été beaucoup plus longue que celle du parser et du lexer. J'ai commencé par définir un type représentant les types de valeurs possibles dans le fichier `ast.ml`

```
type typ =
  Tvoid
| Tnil
| Tint
```

```
| Tbool
| Tstring
| Ttuple  of typ list
| Tstruct of ident
| Tref    of typ
```

Je reviens ici rapidement sur la remarque que j'ai faite en section 2.1 à propos du constructeur `Etuple` introduit dans les expressions. Sans ce constructeur, on pourrait se passer du contructeur `Tvoid` dans le type `typ`. En effet, une expression serait alors du type `typ list` et la liste vide représenterait le type `Tvoid`. Il est possible que cette modification simplifie quelques parties du code du typeur mais je n'ai pas eu la volonté de la faire.

Ensuite, j'ai redéfini un type d'ast *typé* dans le fichier `ast.ml`.

```
type texpr =
  Tenil
| Teint    of int64
| Testring of string
| Tebool   of bool
| Tident   of ident
| Tetuple  of texpr list
| Tattr    of texpr * ident
| Tcall    of (ident option) * ident * texpr list
| Tunop    of unop * texpr
| Tbinop   of binop * texpr * texpr
| Tprint   of texpr list
| Tnew     of typ

type tinstruction =
  Tnop
| Texpr   of texpr
| Tasgn   of texpr * texpr
| Tblock  of tinstruction list
| Tdecl   of ident list * typ option * texpr option
| Treturn of texpr
| Tfor    of texpr * tinstruction
| Tif     of texpr * tinstruction * tinstruction
```

On ne trouve plus de localisation dans ce type car passé le typage, le compilateur n'est plus censé pouvoir encore échouer. Les localisations n'étant utiles que pour préciser les messages d'erreurs qu'il renvoie, passé le typage on peut se permettre de les oublier.

Le typage d'un package renvoie un environnement du type

```
type env = {
    structs : tstruct Smap.t;
    types : typ Smap.t;
```

```
funcs : tfunc Smap.t;
vars  : typ Smap.t;
packages : Vset.t }
```

qui retient toutes les informations dont on peut avoir besoin à propos du programme pour pouvoir le compiler plus tard [1]. Cet environnement est ensuite conservé dans une table globale, permettant de le retrouver rapidement, quand il est importé dans un autre package par exemple.

3 Quelques petites extentions du sujet

3.1 Compiler plusieurs packages

J'ai légèrement étendu la syntaxe du `PetitGo` pour avoir la possibilité d'importer d'autres packages que `fmt`. La raison pour laquelle je l'ai fait est simplement que je trouvais amusant de pouvoir construire de petites bibliothèques en `PetitGo` qui s'utiliseraient les unes les autres.

Pour pouvoir importer un autre fichier, il faut compiler ce dernier en même temps que le fichier qui l'importe. Par exemple, pour utiliser des arbres binaires de recherches dans mon programme, je compile avec la commande

```
main.exe abr.go mon_programme.go
```

On notera que les fichiers doivent être importés dans un bon ordre, sans quoi ils ne seront pas compilé. De plus, un fichier définit un package [2] dont le nom est exactement celui donné au debut du fichier par la ligne

```
package abr
```

Le nom de ce package ne dépendra pas du nom du fichier ni de sa position relative dans l'arborescence de fichier [3].

Pour utiliser une fonction ou une structure du package importé, il faut les faire précéder du nom du package et d'un point. Comme, par exemple, dans le code suivant utilisant le fichier de test `abr.go`

```
var dico *abr.BST = nil
abr.add(&dico, 42)
abr.add(&dico, -1)
abr.print(dico); fmt.Print("\n")
```

1. du moins toute les informations que j'estime utiles pour l'instant, il est fort probable que viennent s'y ajouter de nouveaux champs plus tard dans le projet.
2. contrairement au langage `Go`
3. pas de package `math/rand` par exemple

3.2 Gestion des erreurs

J'ai travaillé un petit peu plus pour afficher de belles erreurs, en particulier lors d'une erreur de typage. J'ai consigné dans le fichier `error.ml` un ensemble de fonctions qui gèrent le rendu des erreurs. L'ensemble des exceptions que l'éxécution du programe est suceptible de lever à un moment sont les suivantes. Elles sont chacune accompagnées d'un ensemble de fonctions levant ces exceptions avec un message personnalisé.

```
exception Error of string
exception Compile_error of position * string
exception Hint_error of position * string * string
exception Double_pos_error of position * position * string
exception Cycle_struct of string list
```

Lorsqu'une erreur survient, elle est localisé à une position du fichier passé en argument du compilateur. J'affiche donc la position au format demandé, suivit d'un message d'erreur et d'un petit bout du fichier correspondant au passage de l'erreur.

```
File "typing/bad/testfile-leftvalue-2.go", line 3, characters 24-25:
Error: invalid argument for &: has to be a left value.

 2:
 3: func main() { var x = &1 }
    ------------------------^--
```

Parfois, on a plus d'information lors de l'erreur. Par exemple, lorsqu'une variable ou une structure est déclarée plusieurs fois.

```
File "typing/bad/testfile-redeclared-1.go", line 4, characters 6-7:
Error: a structure with name ‘T‘ already exists.

 1: package main
 2: type T struct {}
    ^---------------
 3: func main() {}
 4: type T struct {}


 3: func main() {}
 4: type T struct {}
    -----^----------
```

Pour ces types d'erreurs, il peut y avoir un problème quand la seconde position est originairement d'un autre fichier. Le problème n'est pas très diffile à résoudre et le sera peut être d'ici janvier, mais compte tenu du peu de pertinence que cela avait pour le projet je ne l'ai pas fait pour l'instant.

Quand l'erreur est sur un nom de variable (ou de structure ou de packages ou...) je regarde parmis les noms existants celui qui est le plus proche [4] et je propose un nom de variable qui pourrait convenir.

```
File "test.go", line 32, characters 7-8:
Error: unknown function ‘a‘.
Hint: did you mean ‘add‘ ?

 31:        x := (55 * i) % 34
 32:        abr.a(&dico, x)
     -----------^----------
 33:        abr.prt(dico)
```

4 Conclusion, suite du projet et améliorations possibles

Mon typeur passe bien tous les tests. J'aurais aimé que le code soit moins long et plus simple par endroits mais j'en suis relativement satisfait. Bien que j'ai terminé le typage depuis une semaine environ, je n'ai pas encore commencé la production de code : préferant implémenter les fonctionnalitées décrites dans la section 3. Au regard de tout le travail à fournir pour la production de code, c'était sans doute une erreur. D'autant plus que ces deux extentions me semble, au final, assez peu pertinentes dans un projet de compilation (en tout cas pas aussi pertinente que l'utilisation efficace des registres et/ou d'un garbage collector). Cela dit, maintenant que c'est je ne pense pas qu'elles me poseront de difficultées particulières. Il faudra sans doute revoir les structures de données produites après le typage pour y ajouter des informations dont j'aurais besoin dans la production de code.

4. pour la distance minimum d'édition