Compilateur Pascal avec filtrage
troisième partie : production de code

à rendre avant le lundi 12 décembre 10 heures

Le but de cette dernière partie est de produire un code exécutable, étant donné un fichier Pascal, une fois les analyses syntaxiques et sémantiques passées avec succès.

1 Sémantique

1.1 Pascal traditionnel

On donne ici les principaux aspects de la sémantique du Pascal :

- Pascal est un langage strict : toutes les opérandes ou arguments de fonctions sont évalués. Dans une expression, l'ordre d'évaluation des sous-expressions n'est pas spécifié.
- La seule exception au point précédent est l'évaluation *paresseuse* des expressions e_1 `and` e_2 et e_1 `or` e_2 : pour l'opérateur `and` (resp. `or`), e_2 n'est évaluée que si e_1 est vrai (resp. faux).
- Passage des arguments : quel que soit leur type, les arguments sont passés par référence lorsque le mot clé `var` est présent, par valeur sinon.
- La valeur retournée par une fonction est la *dernière* valeur affectée à la variable portant le nom de la fonction lorsque le flot de contrôle atteint la fin de la fonction.
- Dans une boucle `for` x `:=` e_1 `to` e_2 `do` i, les expressions e_1 et e_2 ne sont évaluées qu'une seule fois. Aucune évaluation de i n'est faite si la valeur de e_2 est strictement inférieure à celle de e_1.
- La procédure `writeln` affiche chacun de ses arguments sur une ligne différente (c'est un point sur lequel on diffère légèrement du Pascal). Les booléens seront affichés `true` et `false` respectivement.

Pour tous les autres points de la sémantique du Pascal, on vous renvoie aux tests que vous pourrez effectuer avec un vrai compilateur Pascal ou à la littérature.

Les primitives graphiques sont compilées vers les primitives graphiques de la machine virtuelle de manière immédiate.

1.2 Types construits et filtrage

Concernant l'extension considérée dans ce projet, à savoir les types construits et le filtrage, voici quelle en est la sémantique :

- Dans un filtrage, la *première* branche qui filtre la valeur est sélectionnée (comme en Ocaml).
- Les variables apparaissant dans le motif sont de *nouvelles* variables locales, initialisées *par valeur* avec les composants de la valeur filtrée (comme en Ocaml).

– À la différence d'Ocaml, cependant, ces variables sont modifiables en place (comme toutes les autres variables du Pascal), mais leur modification n'entraîne pas la modification de la valeur filtrée puisqu'elles ont été initialisées par valeur. De manière générale, il n'y a aucun moyen de modifier en place les valeurs des types construits (comme en Ocaml). Ainsi, le programme suivant

```
program test;
type list = Nil | Cons of integer * list;
var l : list;
begin
  l := Cons(1, Cons(2, Nil));
  case l of Nil: | Cons(x, _): x := 0; writeln(x); end;
  case l of Nil: | Cons(x, _): writeln(x); end;
end.
```

  est correct et affiche successivement `0` et `1`.

– Enfin, la comparaison sur des types construits, avec `=` ou `<>`, correspond à l'égalité *structurelle* (comme en Ocaml).

2 Production de code

2.1 Langage cible

La machine abstraite à pile servant de langage cible est une légère extension de la machine décrite et utilisée dans le cours. Elle est détaillée, ainsi que les interprètes qui en sont fournis, dans le document annexe *Machine virtuelle pour le projet de compilation*.

2.2 Représentation des valeurs

Toutes les valeurs sont allouées sur la pile, à l'exception des constructeurs non constants qui sont alloués sur le tas.

2.2.1 Types primitifs

Les entiers et les flottants sont représentés de manière immédiate par les entiers et les flottants de la machine virtuelle.

Les booléens sont représentés par des entiers, avec la même convention que la machine virtuelle : 0 représente `false` et 1 représente `true`.

2.2.2 Tableaux

Les tableaux sont représentés par l'empilement de leurs éléments dans l'ordre lexicographique croissant des indices. En particulier, un tableau multi-dimensionnel tel que

```
t : array [0..1] of array [1..3] of integer;
```

sera représenté de la manière suivante :

⋮
`t[1][3]`
`t[1][2]`
`t[1][1]`
`t[0][3]`
`t[0][2]`
`t[0][1]`
⋮

2.2.3 Types construits

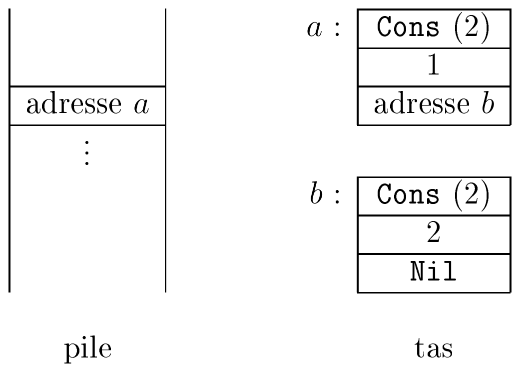
Les constructeurs constants (i.e. sans argument) seront représentés par des entiers tous distincts (au minimum pour un même type). Les constructeurs non constants seront représentés par des adresses vers des blocs alloués sur le tas contenant, d'une part une identification du constructeur (nécessaire pour distinguer les constructeurs dans le filtrage et ainsi que pour implanter l'égalité structurelle), et d'autre part autant de champs qu'il y a d'arguments au constructeur.

Ainsi, avec la déclaration du type construit suivant

```
type list = Nil | Cons of integer * list;
```

le constructeur `Nil` est représenté par un entier, et le constructeur `Cons` par un bloc alloué sur le tas, de taille 3, dont le premier champ identifie `Cons` et indique le nombre de ses arguments (ici 2), le deuxième champ contient un entier et le troisième une valeur de type `list` (soit à nouveau l'entier représentant `Nil` ou l'adresse d'un bloc sur le tas représentant `Cons`).

La valeur `Cons(1, Cons(2, Nil))` sera donc représentée sur la pile par une adresse a correspondant à la situation suivante :



Le nombre d'arguments du constructeur stocké dans le premier champ n'est utile que pour l'égalité structurelle.

Remarque importante : on supposera l'existence d'un GC et l'on se contentera donc d'allouer les blocs sur le tas sans chercher à les libérer.

2.3 Fonctions

Une fonction est compilée en un morceau de code appelé par un `call`. L'appelant empile successivement

1. une place pour la valeur de retour ;
2. les arguments e_1, …, e_n.

Au retour, l'appelant se charge de dépiler ces valeurs (sauf la valeur de retour).

L'appelé se retrouve donc dans la situation (générale) suivante :

FP $\rightarrow$	
	e_n
	$\vdots$
	e_1
	résultat

et il alloue ensuite au dessus de FP la place nécessaire aux variables locales de son code.

3 Travail demandé

Adapter votre compilateur pour qu'il effectue la production de code. Votre compilateur devra reconnaître les deux options existantes `-parse-only` et `-type-only` (cette dernière indiquant désormais de s'arrêter juste après la phase d'analyse sémantique).

En l'absence d'erreur lexicale, syntaxique ou sémantique, la production de code ne doit pas échouer, et le code doit être produit dans le fichier `fichier.vm` si le fichier source s'appelle `fichier.pas`.

Votre code devra comprendre au moins un fichier `compile.ml` contenant les principales fonctions de production de code. L'introduction d'autres fichiers devra être justifiée dans le rapport.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `compilo`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Votre projet doit également s'accompagner d'un rapport (5 pages maximum) expliquant les difficultés rencontrées et les solutions techniques adoptées. Le rapport pourra être rendu soit directement dans l'archive contenant les sources de votre projet (formats pdf, postscript ou openoffice), soit imprimé et remis au secrétariat du master M1.

Votre projet est à rendre à votre chargé de TP par e-mail[1] au plus tard le **lundi 12 décembre à 10h00**.

[1] `sylvain.conchon@lri.fr`, `jean-christophe.filliatre@lri.fr`, `marc.pouzet@lri.fr`, `matthieu.sozeau@lri.fr`, `benjamin.monate@cea.fr`