Projet du cours « Introduction à la compilation »
Partie 2 : Interprétation et analyse statique

version numéro ae80893cb705cb4a02f93e6e5bf4fdaecd77dedf

Le sujet du projet du cours d'introduction à la compilation a pour objectif l'implémentation d'un langage de programmation. Le projet est constitué de deux parties. La première est l'implémentation de l'analyse lexicale et syntaxique (déjà traitée), la seconde est l'implémentation d'un interprète et d'un vérificateur de types.

Cette année, le langage est un langage fonctionnel avec types algébriques.

Dans cette seconde partie du projet, on rappelle d'abord la syntaxe abstraite du langage et on introduit la sémantique à grands pas du langage. Ensuite, vous pouvez faire choisir d'implémenter une optimisation de cette sémantique à grands ou bien un algorithme de typage (ou bien faire les deux si vous le souhaitez !).

Encore une fois, on vous fournit une série de fichiers qui constituent un squelette d'une implémentation de ce langage. Vous devez de nouveau compléter ce squelette en important les fichiers que vous avez écrits dans la première partie et en définissant les parties manquantes de l'interprète et du vérificateur de types.

1 Syntaxe abstraite

On vous rappelle ici la syntaxe abstraite du langage (qui est en correspondance directe avec les types définis dans le module `AST`). Pour une certaine métavariable X, on écrit $\vec{X}$ pour représenter une séquence, potentiellement vide, de X et on écrit $X^?$ pour réprésenter un sous-terme optionnel.

p	$::=$	$\vec{d}$	***Programme***
d	$::=$		***Définitions***
	$\mid$	d_v	*Valeur*
	$\mid$	`type` $t(\vec{\alpha}) = T$	*Type défini par l'utilisateur*
d_v	$::=$		***Définitions de valeur***
	$\mid$	`val` $b = e$	*Valeur*
	$\mid$	`def` $\vec{d_f}$	*Fonctions mutuellement récursives*
d_f	$::=$	$b =$ `fun` $\vec{b}\,(: T)^? \to e$	***Définitions de fonction***
e	$::=$		***Expressions***
	$\mid$	n	*Entier*
	$\mid$	c	*Caractère*
	$\mid$	s	*Chaîne de caractères*
	$\mid$	x	*Variable*
	$\mid$	$K_{T^?}[e^?]$	*Somme*
	$\mid$	$\{\vec{t}\}_{T^?}$	*Produit*
	$\mid$	$(e : T)$	*Annotation de type*
	$\mid$	$e; e$	*Séquencement*
	$\mid$	d_v `in` e	*Définition locale*
	$\mid$	$e\ e$	*Application*
	$\mid$	`case`$_{T^?}\{\vec{c}\}$	*Analyse de motifs*
	$\mid$	`fun` $b \to e$	*Fonction anonyme*
b	$::=$	$x\ (: T)^?$	***Liaison d'une variable***
t	$::=$	$K \leftarrow e^?$	***Définition de composante de produit***
c	$::=$	$p \Rightarrow e$	***Cas d'analyse de motifs***
p	$::=$		***Motifs***
	$\mid$	$K_{T^?}[p^?]$	*Somme*
	$\mid$	$\{\vec{q}\}_{T^?}$	*Produit*
	$\mid$	p `and` p	*Conjonction*
	$\mid$	p `or` p	*Disjonction*
	$\mid$	`not` p	*Négation*
	$\mid$	1	*Universel*
	$\mid$	x	*Variable*
	$\mid$	0	*Vide*
q	$::=$	$K \leftarrow p^?$	

2 Sémantique operationnelle

2.1 Évaluation des expressions

Les expressions du langage s'évaluent vers des valeurs closes v dont la syntaxe est :

v	::=		***Valeurs closes***
	|	δ	*Primitives*
	|	n	*Entier*
	|	c	*Caractère*
	|	s	*Chaîne de caractères*
	|	$(\texttt{case}\{\vec{c}\})[E]$	*Fermeture*
	|	$\{\vec{v_c}\}$	*Valeur structurée*
v_c	::=	$K \to v^?$	***Valeurs étiquetées***

Les valeurs incluent les entiers, caractères, chaînes de caractères, fermetures, valeurs structurées et les primitives. Les primitives sont décrites dans la section suivante. Les fermetures servent à représenter les fonctions de première classe. Une fermeture est une paire composée du code de la fonction à représenter et de l'environnement d'évaluation E dans lequel la fonction a été définie. Dans ce langage, les fonctions sont définies par cas sur la forme de l'entrée. La syntaxe des environnement d'évaluation est :

E	::=		***Environnement d'évaluation***
	|	$\bullet$	*Environnement vide*
	|	$E; x \mapsto v$	*Environnement étendu*

La sémantique à grands pas des expressions est spécifiée par le jugement

$$E \vdash e \Downarrow v$$

qui se lit « Sous l'environnement E, l'expression e s'évalue en une valeur v. »

Ce jugement s'appuie sur quatre jugements auxiliaires :

- L'interprétation des définitions :

  $$E \vdash d_v \Downarrow E'$$

  qui se lit « La définition d_v étend l'environnement E en l'environnement E'. »
- La capture par un motif :

  $$\vdash p \sim v \Downarrow E$$

  qui se lit « Le motif p capture la valeur v en produisant un environnement E. »
- La non-capture par un motif :

  $$p \not\sim v$$

  qui se lit « Le motif p ne capture pas la valeur v. ».
- L'analyse par cas d'une valeur :

  $$E \vdash \vec{c} \diamond v \Downarrow v'$$

  qui se lit « L'analyse par les cas $\vec{c}$ de la valeur v s'évalue en la valeur v' sous l'environnement E. »

Dans les règles qui suivent, les produits sont à considérer à permutation des composantes près. De plus, comme les types n'influent pas sur l'évaluation des programmes, on ignore les annotations de type dans les expressions.

Pour finir, le jugement d'évaluation est spécifié par les règles suivantes :

$$\frac{E(x) = v}{E \vdash x \Downarrow v} \qquad \frac{}{E \vdash n \Downarrow n} \qquad \frac{}{E \vdash c \Downarrow c} \qquad \frac{}{E \vdash s \Downarrow s} \qquad \frac{E \vdash e \Downarrow v}{E \vdash K[e] \Downarrow \{K \to v\}}$$

$$\frac{}{E \vdash \{K_1 \leftarrow e_1, \ldots, K_n \leftarrow e_n\} \Downarrow \{K_1 \to v_1, \ldots, K_n \to v_n\}} \qquad \frac{}{E \vdash \texttt{fun}\ x \to e \Downarrow (x \Rightarrow e)[E]} \qquad \frac{}{E \vdash \texttt{case}\{\vec{c}\} \Downarrow (\texttt{case}\{\vec{c}\})[E]}$$

$$\frac{E \vdash e_1 \Downarrow (\texttt{case}\{\vec{c}\})[E'] \qquad E \vdash e_2 \Downarrow v' \qquad E' \vdash \vec{c} \diamond v' \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow v_1 \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1; e_2 \Downarrow v_2} \qquad \frac{E \vdash d_v \Downarrow E' \qquad E' \vdash e \Downarrow v}{E \vdash d_v\ \texttt{in}\ e \Downarrow v}$$

$$\frac{\vdash p \sim v' \Downarrow E' \qquad EE' \vdash e \Downarrow v}{E \vdash p \Rightarrow e, \vec{c} \diamond v' \Downarrow v} \qquad \frac{\vdash p \not\sim v' \qquad E \vdash \vec{c} \diamond v' \Downarrow v}{E \vdash p \Rightarrow e, \vec{c} \diamond v' \Downarrow v} \qquad \frac{\forall E, \vdash p \sim v \Downarrow E \text{ n'est pas dérivable}}{p \not\sim v} \qquad \frac{}{\vdash 1 \sim v \Downarrow \bullet}$$

$$\frac{\vdash p_1 \sim v \Downarrow E \qquad \vdash p_2 \sim v \Downarrow E' \qquad E, E' \text{ disjoints}}{\vdash p_1\ \texttt{and}\ p_2 \sim v \Downarrow EE'} \qquad \frac{\vdash p_1 \sim v \Downarrow E}{\vdash p_1\ \texttt{or}\ p_2 \sim v \Downarrow E} \qquad \frac{\vdash p_2 \sim v \Downarrow E}{\vdash p_1\ \texttt{or}\ p_2 \sim v \Downarrow E} \qquad \frac{p_1 \not\sim v}{\vdash \texttt{not}\ p_1 \sim v \Downarrow \bullet}$$

$$\frac{\vdash p \sim v \Downarrow E}{\vdash K[p] \sim \{K \leftarrow v\} \Downarrow E} \qquad \frac{}{\vdash K[] \sim \{K\} \Downarrow \bullet} \qquad \frac{\forall i, \vdash p_i \sim v_i \Downarrow E_i \qquad (E_i)_{i \in [1 \ldots n]} \text{ disjoints deux à deux}}{\vdash \{K_1 \leftarrow p_1 \ldots K_n \leftarrow p_n\} \sim \{K_1 \to v_1 \ldots K_n \to v_n\} \Downarrow \cup_{i=1}^{n} E_i}$$

$$\frac{}{\vdash x \sim v \Downarrow x \mapsto v} \qquad \frac{E \vdash e \Downarrow v}{E \vdash x = e \Downarrow E; x \mapsto v} \qquad \frac{E' \equiv (E; f_1 \mapsto v_1; \ldots; f_n \mapsto v_n) \qquad \forall i, E' \vdash e_i \Downarrow v_i}{E \vdash \texttt{def}\ f_1 = e_1, \ldots, f_n = e_n \Downarrow E'}$$

Quelques commentaires sur ces règles Pour implémenter ces règles, vous devez utiliser le module `Runtime` qui fournit une implémentation des environnements E et des valeurs v. L'implémentation peut simplement paraphraser "mot à mot" la spécification de la plupart des règles exceptée la règle des fonctions mutuellement récursives. Pour celle-ci, il y a un cycle que vous devez briser : pour calculer les valeurs v_i, il faut utiliser l'environnement E' mais cet environnement est lui-même construit avec les valeurs v_i !

Concours de l'interprète le plus rapide De nombreuses optimisations sont envisageables pour implémenter un interprète plus rapide que celui obtenu en paraphrasant les règles de sémantique. Vous pouvez par exemple améliorer la complexité des opérations primitives des environnements ou bien avoir une phase liminaire de réécriture du programme, à appliquer avant l'évaluation. Vous êtes libres de modifier le programme autant que nécessaire ! Par contre, vous devez vous assurer que l'afficheur de résultats n'est pas impacté par vos modifications de façon à ce que les tests continuent à être validés.

2.2 Évaluation des primitives du langage

Les primitives du langage sont implémentées par le module `Primitive`. On y trouve la définition des opérateurs déclarés dans le module `Operator` ainsi que des primitives associées à certains identificateurs de programme. Votre interprète doit détecter quand une primitive du langage est utilisée et délégué son traitement au module `Primitive`. Notez que vous n'avez pas le droit de modifier l'interface de ce module.

2.3 Évaluation des programmes

Les programmes sont des listes de définitions de valeurs et de types. Pendant la phase d'évaluation, on ignore les définitions de types. Le résultat de l'évaluation d'un programme est tout simplement constitué des valeurs qu'il définit.

3 Option 1 : Évaluation des expressions avec partage maximal et mémoïsation

Deux calculs distincts peuvent mener à la même valeur v. Plutôt que d'allouer un espace mémoire distinct pour stocker deux fois la même valeur, on peut de façon systématique vérifier à chaque fois que l'on produit une valeur, si elle n'est pas déjà présente en mémoire et le cas échéant, renvoyer cette valeur déjà représentée en mémoire plutôt que de réallouer un nouvel espace. Cette technique dit "du partage maximal" permet de minimiser l'empreinte mémoire de l'exécution du programme et de tester si deux valeurs sont égales en testant si elles sont représentées par la même zone mémoire.

Un calcul peut répéter l'évaluation d'une fonction f pour une même entrée. Lors de ces répétitions, il peut alors être utile de ne pas réévaluer le corps de la fonction mais de renvoyer la valeur que l'on a calculée lors du premier appel

de f sur cette entrée. Cette technique dite de "mémoïsation" permet très souvent de réduire le temps d'exécution de certains algorithmes.

Votre interprète doit implémenter le partage maximal et la mémoïsation lorsque l'option `--memo` est activée.

4 Option 2 : Typage

Comme nous l'avons vu en cours, le typage est une analyse statique qui approche la forme des résultats à l'aide d'un terme que l'on appelle type. Cette approximation est construite *via* une classification des expressions et éventuellement à l'aide d'annotations de type spécifiées par l'utilisateur.

On spécifie généralement un système de type à l'aide d'un jugement de la forme $\Gamma \vdash e : T$ qui se lit « Sous l'environnement de typage Γ, l'expression e a le type T. »

Vous devez dans un premier temps spécifier les règles de typage du système de type sur papier et dans un second temps les implémenter dans votre programme de façon à ce que l'option `--type` active la vérification du bon typage des programmes.

5 Code fourni

Un squelette de code vous est fourni, il est disponible sur la page web du cours. Il contient des `Makefile`s ainsi que des modules O'Caml à compléter. Voici une description rapide de ces modules (les modules à compléter sont précédés d'une étoile) :

- `Clap` : analyse les arguments fournis en ligne de commande et lance les processus correspondants.
- `AST` : définit le type de l'arbre de syntaxe abstraite.
- ⋆ `Lexer` : implémente l'analyse lexicale à l'aide de l'outil `ocamllex`. (Réutilisez votre travail de la première partie.)
- ⋆ `Parser` : implémente l'analyse syntaxique à l'aide de l'outil `menhir`. (Réutilisez votre travail de la première partie.)
- `Printer` : implémente un afficheur d'arbre de syntaxe.
- `Runtime` : implémente un environnement lexical et les valeurs du langage.
- `Primitive` : implémente les primitives du langage.
- `Memo` : implémente le flag d'activation du partage maximal et de la mémoïsation.
- ⋆ `Interpreter` : implémente un interprète pour la sémantique à grands pas décrite plus haut.
- ⋆ `TypeCheck` : implémente un vérificateur de types pour le langage.
- `Position` : définit des types et des opérations pour décrire un emplacement dans un fichier.
- `Error` : définit des opérations pour arrêter le programme en cas d'erreur et afficher un message informatif.

La commande `make` produit un exécutable appelé `clap`. On doit pouvoir l'appeler avec un nom de fichier en argument. En cas de réussite (de l'analyse syntaxique, de l'évaluation et du typage), le code de retour de ce programme doit être 0. Dans le cas d'un échec, le code de retour doit être 1.

Par ailleurs, la commande `make check` lance une batterie de tests sur votre programme. Peu de tests sont fournis, vous devez en écrire vous-même ! (Il suffit de rajouter de nouveaux fichiers dans les répertoires idoines, ils seront automatiquement pris en compte.)

6 Travail à effectuer

Comme pour la première partie, vous devez compléter les parties du code fourni de la forme :

```
failwith "Students, this is your job."
```

Avant toute chose, vous devez définir la variable `$(STUDENTS)` dans le `Makefile.local` du code fourni et compléter le fichier `AUTEURS` à la racine du répertoire en suivant le format suivant :

```
nom1, prenom1, numero-etudiant1, email1
nom2, prenom2, numero-etudiant2, email2
```

La compilation s'effectue par la commande « `make` ».

Vous pouvez lancer une batterie de tests sur votre projet à l'aide de la commande « `make check` ».

Le projet est à rendre **avant le** :

24 mai 2013 à 23h59

Votre travail, à effectuer par groupe de 2, devra être construit par la commande « `make dist` ». Cette commande produit une archive de la forme `clap-$(STUDENTS)-13.2.tar.gz`. Vous devez le déposer à l'URL `http://www.regis-gianas.org/~clap2013`.

Votre projet sera traité de façon automatique : un mail de confirmation vous sera envoyé contenant, en particulier, le résultat de votre travail sur notre propre batterie de tests. Il n'est pas interdit de soumettre votre travail plusieurs fois (le système de soumission sera mis en place à partir du 8 mai 2013). Attention, si vous n'avez pas respecté le format du fichier `AUTEURS` décrit plus haut, la correction automatique échouera, et vous aurez 0.

Pour finir, vous devez vous assurer des points suivants :

- Le projet contenu dans cette archive **doit compiler**.
- Vous devez **être les auteurs** de ce projet.
- Il doit être rendu **à temps**.

Si l'un de ces points n'est pas respecté, la note de 0 vous sera affectée.

7 Log

```
commit ae80893cb705cb4a02f93e6e5bf4fdaecd77dedf
Date:   Wed Apr 24 18:18:38 2013 +0200

    * Partie 2. Version initiale.
```