Projet

Version du 19 avril 2013

Ce projet a pour objectif :

- l'implémentation d'une machine virtuelle pour le langage fonctionnel Clap, tel que spécifié dans le cours d'introduction à la compilation MV6. Clap est un langage fonctionnel de type ML muni de types sommes, produits et récursifs ;
- l'écriture de programmes Clap pour la tester ;
- la découverte d'une phrase secrète, le *codex*, qui est le résultat de l'exécution par la machine virtuelle d'un programme fourni.

Dans la suite, on décrit le jeu d'instruction de la machine, qui est très proche de (mais pas identique à) celle de OCaml, ainsi que sa représentation binaire en *bytecode*. On vous fournit un squelette de code comprenant un compilateur de l'AST vers ces instructions, et l'assembleur ; vous devez implémenter le désassembleur et la machine virtuelle, puis choisir une ou plusieurs parties optionnelles du projet et les implémenter.

1 Spécification de la machine

La machine à implémenter est une machine à *a*-pile, plus un tas. Chaque mot mémoire est, comme en OCaml, soit un entier, soit un pointeur vers une zone du tas. Cette zone, un *bloc*, est soit une chaîne de caractères, soit la paire d'un *tag* et d'un tableau contiguë de mots mémoire. Pour ne pas avoir à se soucier de la gestion de la mémoire, on utilise le tas OCaml pour implémenter le tas de notre machine.

1.1 Représentation des données

On représente les valeurs du langage Clap par les mots mémoires correspondant dans le tableau suivant :

valeur	**représentation**
entier	sa représentation binaire 32 bits
caractère	son code ASCII sur 32 bits
n-ème constructeur somme, sans argument	la représentation binaire 32 bits de *n*
n-ème constructeur somme, avec argument	bloc de tag *n* contenant la représentation de l'argument
produit *n*-aire	bloc de tag 0, contenant la représentation des *n* champs
fonction unaire de code *c*	bloc de tag 88, contenant l'adresse de *c* suivi de *n* valeurs

1.2 Jeu d'instructions

Les instructions acceptées par la machine sont les suivantes :

halt (opcode 0) arrête l'exécution de la machine ;

push (opcode 1) empile le contenu de l'accumulateur sur la pile ;

print (opcode 2) affiche le contenu de la chaîne de caractère pointée par l'accumulateur ;

apply (opcode 3) appelle la fonction dont la fermeture est contenue dans l'accumulateur, avec comme argument le contenu du sommet de la pile (voir ci-dessous) ;

acc *n* (opcode 4) copie la valeur de la *n*-ème case du sommet de la pile dans l'accumulateur ;

const *n* (opcode 5) met l'accumulateur à l'entier *n* ;

return *n* (opcode 6) dépile et jette les *n* premières cases de la pile et retourne à la fonction appelante (voir ci-dessous) ;

pop *n* (opcode 7) dépile et jette les *n* premières cases de la pile ;

branchif *o* (opcode 8) saute *o* instructions en avant si l'accumulateur est entier et différent de 0 ;

branch *o* (opcode 9) saute *o* instructions en avant ;

getblock n (opcode 10) met dans l'accumulateur la n-ème case du bloc pointé par l'accumulateur ;

makeblock (t, n) (opcode 11) met dans l'accumulateur un bloc de tag t et contenant les n valeurs au sommet de la pile, qui sont dépilées ;

closure (n, o) (opcode 12) alloue un bloc de taille $n+1$, de tag 88, contenant l'indice de l'instruction située o instructions en avant, suivi des n premières cases de la pile, qui sont dépilées (et forment l'*environnement* de la fermeture) ;

binop b (opcode 13) effectue l'opération binaire dont l' opcode est l'entier b ;

str s (opcode 14) met dans l'accumulateur un pointeur vers la chaîne de caractères s.

Opérations binaires Les opérations binaires prennent leurs deux opérandes respectivement dans l'accumulateur et au sommet de la pile. Elles sont les suivantes :

add addition entière (opcode 15)

sub soustraction entière (opcode 16)

mul multiplication entière (opcode 17)

div division entière (opcode 18)

eqi égalité d'entiers (opcode 19)

cat concaténation de chaînes (opcode 20)

Le résultat est stocké dans l'accumulateur, et le sommet de la pile lu est dépilé.

Convention d'appel Dans cette machine, tous les appels de fonction sont unaires. L'appelant empile l'argument de la fonction à appeler, met sa fermeture dans l'accumulateur, puis exécute **apply**. Cette instruction prépare la pile de façon à y mettre, du bas vers le haut : l'indice i de l'instruction à laquelle retourner après l'appel, puis les n valeurs d'environnement contenue dans la fermeture, puis l'argument. Enfin, elle saute à l'instruction dont l'indice est le premier champ de la fermeture (le code de la fonction appelée). Ce code se termine par l'instruction **return** n. Celle-ci dépile et jette $n+1$ cases au sommet de la pile (n valeurs d'environnements $+$ 1 argument), puis dépile une case contenant l'indice i d'une instruction, et y saute.

1.3 Bytecode

Chaque instruction est représenté par une suite d'octet comme suit :
- un entier argument d'une instruction a pour représentation les 4 octets de sa représentation non signée petit-boutiste ;
- une instruction sans argument (**halt**...**apply**) a pour représentation binaire son opcode, sur 8 bits ;
- un instruction à n arguments entiers (**acc**...**binop**) a pour représentation la concaténation de son opcode sur 8 bits et des représentations de ses arguments ;
- l'instruction **str** s a pour représentation : l'opcode 14 sur 8 bits, suivi de la représentation de la taille de la chaîne de caractère s sur 4 octets, suivi de la chaîne elle-même.

Une suite d'instruction est représenté en bytecode par la concaténation de la représentation de chacune de ses instructions.

2 Code fourni

Un squelette de code vous est fourni, il est disponible sur la page web du cours. Il contient tous les modules de la première partie du projet de MV6, plus les modules OCaml suivants (les modules à compléter sont précédés d'une étoile) :
- `Util` : quelques fonctions utilitaires complétant la librairie standard OCaml ;
- `Instr` : spécification des instructions, du bytecode et du modèle mémoire de la machine, tels que décrits en 1.2 ;
- `Printer` : fonctions d'affichage des expressions, du bytecode, de l'état mémoire...
- ⋆ `Prim` : la liste des primitives du langage, c'est-à-dire les identifiants définis par du bytecode qui seront inclus dans tout programme compilés ;
- `Compil` : le compilateur de l'AST Clap vers du bytecode ;
- ⋆ `Asm` : les fonction de (dés)assemblage ((dé)sérialisation) de bytecode, tels que décrits en 1.3 ;
- `Stk` : une implémentation naïve de pile ;
- ⋆ `Machine` : la machine virtuelle.

La commande `make` produit un exécutable appelé `clap`, et qui sert à la fois de compilateur et de machine virtuelle. On l'appelle avec une liste de fichiers, d'extension `.clap` (fichiers sources) qui seront compilés en un bytecode `.clvm`, ou d'extension `.clvm` qui seront exécutés par la machine. Les options de la commande `clap` sont :

`-x` : au lieu de générer un fichier `.clvm` à partir des source, le passe directement à la machine virtuelle pour exécution ;
`-dinterm` : affiche le code intermédiaire compilé ;
`-dinstr` : affiche le bytecode compilé ;
`-ddasm` : affiche le bytecode désassemblé d'un `.clvm` ;
`-dtrace` : affiche la trace d'exécution, c'est-à-dire l'état mémoire avant l'exécution de chaque instruction.

L'exécutable a besoin d'un analyseur syntaxique pour fonctionner. Il appelle pour cela un exécutable nommé `parser.byte` (ou le nom de votre choix fourni avec l'option `-pp`). Des analyseurs syntaxiques sont fournis pour diverses versions d'OCaml à la racine du projet ; si besoin, changer le lien `parser.byte` vers celui pour votre version d'OCaml. Pour que `clap` trouve `parser.byte`, exécutez-le toujours depuis le répertoire qui contient ce dernier.

La commande `make check` lance une batterie de tests sur votre programme, non fournis. Pour écrire un test positif/négatif, mettre un fichier d'extension `.clap` dans le répertoire `tests/{good|bad}/semantic`. Un test positif `test.clap` doit être accompagné d'un fichier `test.expected` contenant la sortie attendue de `./clap test.clap -x test.clap`.

Le fichier `codex.mlvm` contient le bytecode du programme à exécuter pour découvrir la phrase secrète.

3 Travail à effectuer

3.1 Partie obligatoire

La première partie du projet est l'écriture de la machine virtuelle et du désassembleur de bytecode Clap. Le compilateur ainsi que l'assembleur est fourni. Pour cela, vous devez compléter les parties du code fourni de la forme :

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

Vous fournirez en outre une batterie *conséquente* de tests (une dizaine minimum) sous la forme de fichiers source Clap. Une fois ces tâches effectuées, vous découvrirez alors le *codex* en exécutant `codex.clvm`.

3.2 Parties optionnelles

Pour augmenter votre note, vos connaissances ou votre plaisir, vous pouvez choisir une ou plusieurs tâches parmi les suivantes (de la plus facile à la plus stimulante) :

Complétion des primitives du langage Vous ajouterez les instructions à la machine et les primitives du langage correspondant aux opérateurs suivants :
- `&&` la conjonction booléenne ;
- `||` la disjonction booléenne ;
- `~` la négation booléenne ;
- `<=`, `=>`, `<`, `>`, `!=` les comparaisons entières ;

Optimisation de la structure de pile La structure de donnée utilisée pour représenter la pile dans le module `Stk`, une simple liste, est inefficace pour exécuter du code, car on fait souvent des lectures profondément dans la pile (instruction **acc**). Proposer une implémentation efficace du module `Stk`, sans changer son interface. Bonus si elle est purement fonctionnelle.

Optimisation des appels terminaux Vous ajouterez une ou plusieurs instructions à la machine dédiées à optimiser les appels terminaux et les faire exécuter en taille de pile constante. Après la compilation, vous rajouterez une passe d'optimisation du bytecode produit utilisant cette instruction.

Un débogueur Implémenter votre machine virtuelle de façon purement fonctionnelle. Puis implémenter un *débogueur*, c'est-à-dire une option particulière de `clap` qui affiche après l'exécution de chaque instruction l'état de la mémoire et demande si l'on veut, soit avancer, soit revenir en arrière, soit sauter à un numéro d'instruction donné.

Optimisation des appels *n*-aires Vous ajouterez une ou plusieurs instructions à la machine dédiées au traitement en bloc des appels *n*-aires, comme vu en cours. Vous modifierez le compilateur ou rajouterez une passe d'optimisation en conséquence.

Vérification de bytecode Un bytecode est composé de plusieurs *segments*, un segment étant défini comme une série d'instructions contiguë finissant par **halt**, **branch**, **branchif** ou **return**. Le compilateur est censé produire du bytecode tel que chaque segment de code, si executé entièrement, ne modifie pas la taille de la pile. Écrire une fonction qui vérifie *statiquement* cet invariant, c'est-à-dire sans exécuter le code en question.

3.3 Modalités

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
```

Le projet est à rendre **avant le** :

jeudi 25 avril 2013 à 23h59

Votre travail, à effectuer par groupe de 2, devra comprendre un fichier `README` expliquant synthétiquement votre implémentation, les choix de design, les difficultés rencontrées, et le codex. Il devra être construit par la commande « `make dist` ». Elle produit une archive de la forme `clap-$(STUDENTS)-13.1.tar.gz` contenant tous les fichiers indiqués dans le fichier `distributed_files`. Vous devez l'envoyer par mail à l'adresse `puech@pps.univ-paris-diderot.fr`. Toute question relative au projet devra être adressée à la liste de diffusion du cours.