Nicolas YING
M. Alain CHILLÈS
Theory of Programming Language
17 December 2016

Compilateur, RE & REPL

Ce projet consiste à developper un environnement de développement pour le langage AC, un langage mathématique créé par le prof Alain CHILLÈS.

Ce projet adopte la norme Hugolonicolasienne sur les critères du compilation, créée par Hugo BI et Nicolas YING, disponible via `https://github.com/SPEITCoder/VM_LAC_hugolonicolasien`.

Ce projet consiste de trois parties :

- un compilateur qui prend un ficher de code `**.lac`, génère une fichier `**.lacc` comme LAC compilé ;
- un runtime (RE comme Runtime Environment) qui exécute un ficher `**.lacc` dans une machine virtuelle ;
- un interpréteur (REPL comme Read-Evaluate-Print Loop) qui soutient une partie des instructions définies dans LAC, et renvoie des résultats.

Organisation des fichiers

Les fichiers de C se trouvent tous dans le répertoire racine :

- Analyse lexicale (Projet 1) : `analyse_lexical.c, analyse_lexical.h`
- Analyse syntaxique (Projet 2, intégré dans les programmes sortants) : `Syntax_calculate/BNF_C.c, Syntax_calculate/BNF_C.h, Syntax_calculate/calculate.c`
- Interpréteur (Projet 3) : `mode_calculatrice.c, mode_calculatrice.h`
- Compilateur (Projets 4 et 5) : `compiler.c, compiler.h`
- Runtime (Projets 4) : `runtime.c`
- Les composants communs : `common_component.c, common_component.h, processeur.c, processor.h`

Il y a de plus un `makefile` (dépendance d'installation `make`), deux dossiers `Release` et `Debug` pour contenir les programmes sortants, un dossier `Test` comprenant les codes pour tester.

Compilation et Exécution (niveau C)

- `make` compile l'interpréteur, le runtime et le compilateur en les mettant sous le dossier Release;

- `make Interpreter`, `make Runtime`, `make Compiler` compilent respectivement les trois différents programmes ;
- `make verbose` compile les trois programmes avec beaucoup plus d'information à afficher pendant son exécution, pour les utilisateurs curieux, les programmes se trouvent sous le dossier `Debug` ;
- `make clean` supprime les programmes sous le dossier `Release`.

Tous ces instructions ont activé le flag `-Wall` pour générer des warnings pendant la compilation des codes en C.

À l'exécution des programmes, on lance l'interpréteur sans argument supplémentaire.

```
$: Release/interpreter
```

Par contre, pour compiler une fichier, il faut fournir l'adresse du code LAC

```
$: Release/compiler ./factorielle.lac
```

Le code sera compilé et sauvegardé dans le même répertoire que le code, avec le même nom mais avec l'extension `.lacc` .

Pour exécuter un code LAC compilé, il faut fournir l'adresse du code compilé

```
$: Release/runtime ./factorielle.lacc
```

Description des composants

Analyse lexicale

Pour réaliser l'analyse lexicale, la méthode utilisée dans le projet s'agit de couper le texte d'entré vers plusieurs morceau, chacun validant l'expression régulière unique, et puis un autre analyse plus mécanique qui reconnaitre les différents types d'éléments du langage puisque l'on peut y appliquer les différents traitements.

L'écriture de l'expression régulier est une simple logique de « ou », j'ai écrit d'abord des expressions régulières pour les chaines de caractères, un commentaire en ligne, un commentaire multi-ligne et les autres formes d'identificateurs autorisé par le langue,

Voici les expressions :

```
((^|[ \t\n])\" ([^()\"]*[\\].*|[^()\\]*[\\([^()\\]*[\\)][^()\\]*)\")
((^|[ \t\n])[\\]( |\t)[^\n]*($|\n)) // commentaire en ligne
((^|[ \t\n])[(] [^)]*[)]) // commentaire multi-ligne
((^|[ \t\n])[\"] [^\"]*[\"]) // chaine de caractères
([-+/:;.\\=\\*0-9a-zA-Z[:punct:]]+) // le reste
```

et je les ai connecté par les `|` en respectant l'ordre.

Une **hypothèse** remarquante est que l'assortiment de l'expression régulière favorise les parties à gauche, car je suppose que l'expression régulière réalise les mécanismes d'automate, c'est donc

les parties (connectées par les « ou ») à gauche qui sont trouvées d'abord et le recherche se termine en arrivant à un état acceptable.

Les résultats de test convient avec cette hypothèse.

Un problème potentiel de cette écriture de l'expression, c'est qu'elle ne tient pas compte des structures de commentaires multi-ligne dedans les guillemets. La première partie au-dessus résolve ce problème par une détection dédiée à telle structure.

À chaque assortiment de l'expression, le résultat `regmatch_t` donne la position du début et de la fin de cette élément, ceci facilite l'examination du lexème en regardant les premiers caractères et les derniers. À la fin de ce processus, lexème va être sauvegardé dans une liste de lexèmes qui contient le type du lexème et les positions du début et de la fin de ce lexème dans le texte entrée. Les commentaires détectés ne sont pas sauvegardés, donc les types sont identificateurs et chaine de caractères.

Les lexèmes sont :

```
typedef enum lexeme_Type {
    C, // chain de character
    I // identificateur
} lexeme_Type;

typedef struct lexeme_Element {
    lexeme_Type type;
    int begin, end;
} lexeme_Element;
```

La fonction réalisée dans analyse_lexique.c modifie donc la liste de lexème, passée par le processus (soit l'interpréteur soit le compilateur).

Mémorisation des chaines de caractère

Réalisée dans l'interpréteur et le runtime, la mémorisation des chaines de caractère est soutenu par un tableau d'entier `static` en C qui est assez grand, où il y a un pointeur dirigeant vers l'indice le plus petit qui n'est pas utilisé. L'accès au tableau est circulaire, la position d'accès est toujours un modulo de la taille.

Donc il n'y a pas de mécanisme de recyclage d'espace où l'évitement, la stabilité de cette structure est basée sur la taille de ce tableau.

Interpréteur

L'interpréteur est un REPL, qui lit les commandes depuis le `stdin`, les passe vers l'analyse lexicale, et les interprète en exécutant les commandes s'il n y a pas d'erreur.

Le programme soutien les opérations arithmétiques `+, −, ∗, /` (avec remarque de division sur zéro), et `cr, dup, drop, swap, ., count, type, =, calculate, &&, ||, ! (négation), ≤` et `catenate`.

L'environment LAC fourni par l'interpréteur est simple : un pile de donnée, un pile de type, un tableau de symbol statique et un tableau de pointeurs machine virtuelle VM.

Pour chaque entrée (fin d'entrée à carriage), l'interpréteur la passe vers l'analyse lexicale, avec la liste de lexème, il vérifie si une identificateur est bien une fonction définie en traversant le tableau de symbol à partir de la fin (fonction `findFunction` comme un composant commun), si la fonction n'est pas trouvée, il va essayer de l'interpreter comme un nombre entier (négatif ou positif). Une erreur apparaitra si le lexème ne se transforme pas en un nombre entier, c'est un lexème invalide.

Pour les lexèmes qui réussissent à transformer (depuis `char[ ]` à `int`) et les chaines de caractères, l'interpréteur les empile correctement.

Pour les lexèmes qui sont définies comme des fonctions de bases (celles autorisées dans ce mode), on exécute les fonctions, et les erreurs seront générées par le processeur de VM comme les erreurs du typage.

Pour chaque itération, l'interpréteur vérifie si l'entrée est « QUIT » puisqu'il peut se terminer, à la fin d'exécution il affiche les données dans les piles, inspiré par l'exemple de REPL de Forth.

Pour le moment, l'interpréteur ne soutient pas un fichier d'entrée au démarrage parce que les fichier qui seraient soutenus par l'interpréteur sont aussi utilisables en mode compilé.

Compilateur

Le compilateur réalisé dans le projet se conforme à la norme Hugolonicolasienne, un standard qui définit l'ordre des fonctions de processeur et un format du fichier de sortie disons LAC compilé, `**.lacc`.

Le compilateur réalise toutes les fonctions de l'interpréteur, mais aussi des branches conditionnelles, les fonctions composées et les fonctions récursives.

Au moment d'erreur, le compilateur va s'arrêter à compiler, et imprime le code où le problème génère.

TYPAGE

Une différence et difficulté majeure de la réalisation, c'est le contrôle du typage. Pour examiner la validité du code en terme de typage, il y a des variables et les structures comptant le nombre, le type et l'ordre des données pour imiter l'exécution du code.

L'examination du typage sert à générer des erreurs au moment de compilation quand une fonction (de base ou définie dans LAC) prend les entrées depuis les piles, et vérifier si les branches conditionnelles produisent tous des types compatibles. Les variables et les structures gérant les types sont chaque fois vérifiées en comparant le nombre d'entrée et de sortie, vérifie si les types sont compatible (en particulier le type **ANY**).

Il y existe, comme défini dans la programme, trois environments du code, la compilation d'une fonction définie en LAC, la compilation de la fonction main, et la compilation des branches conditionnelles. Les différents environments sont emboités, il y a un structure similaire au pile, qui sauvegarde les informations de comptage de type, qui permet d'utiliser les mêmes lignes de code pour vérifier les types dans ces environments, et retrouver les information au niveau inférieur. (E.g.

Une structure SIVRAI dans les branches conditionnelles est de niveau inférieur par 1 à celui de SIFAUX s'il y existe, mais supérieur par 1 à celui de la fonction contenant cette branche.)

ÉVALUATION DU TYPE ANY

Comme le nom du type indique, ANY est souvent compatible avec les autres types, la dynamique du type est une difficulté pour programmer sa vérification.

Il y a deux cas où on peut determiner précisément le type, un au comparaison de deux branches conditionnelles, la vérification de compatibilité de chaque donnée peut donner un type plus exact si nécessaire ; un autre à la compilation générale, où on peut effectivement numéroter chaque type **ANY** depuis le tableau de symbol en utilisant les entiers négatifs : il y a de plus un tableau qui génère toute les type **ANY** pendant la compilation, chaque indice présente une évaluation du type **ANY**. Ceci permet de garder le mécanisme de vérification du typage, en seulement modifiant la fonction de conversion du type, et les types sont finalement évalués en accédant le tableau (et les types associés, sont de cette façon modifiées en même temps, garantissant la cohérence).

Malgré tout, cette dynamique n'est pas réalisée dans la programmation, car le prototype du compilateur est déjà stable, et puis une autre raison importante, c'est la renonce de vérification du typage à l'apparition du lexème « recurse ». Si éventuellement on ne peut pas déterminer le type d'entrée (du tout) au pire cas, ce n'est pas nécessairement util de réaliser une telle évaluation du type dynamique.

STYLE DE CODAGE RECOMMANDÉ PAR CE COMPILATEUR

Limité par un nombre maximale d'espace du tableau de symbol, du tableau de VM et du tableau d'information du type, le code LAC ne peut pas dépasser une certaine quantité de fonction définie dans une fichier, une quantité de opération et de chaine de caractères et de nombre, et un maximum degré d'emboitage des branches conditionnelles. (Pour l'instance le compilateur permet de réaliser 20 emboîtes, modifiable dans les programmes aux tous premières lignes.)

Le compilateur distincte les fonctions définies dans LAC et une fonction main, dont l'idée est similaire au fonction main en C. La fonction main est la partie du code LAC dehors les définitions de fonction. Il est fortement conseillé, que les codes pour tester le compilateurs mettent bien les opérations dehors les définitions ensemble, i.e. pas de définition du fonction au cours des codes d'opérations.

Il est, toutefois, réalisable de définir les fonctions d'abord et compiler les codes après par manipulation des positions du code avant la compilation du code commence. Je le considère comme une optimisation du codage, ainsi le compilateur aujourd'hui va générer des erreurs indiquant d'existence de définition de fonction en main.

UN PEU PLUS SUR LA NORME HUGOLONICOLASIENNE

La norme définit le format du fichier compilé, qui contient principalement le tableau de VM, pour que les runtimes conformant à la même norme puissent échanger les fichiers compilé.

1. Les deux premières éléments dans le tableau sont réservées aux information général du code compilé. La première est la version de norme utilisée par le compilateur pour que les runtimes

déterminent si le code est acceptable et compatible. La deuxième est la position de point d'entrée de la fonction main.

2. La fonction main est donc toujours les dernières éléments dans le tableau, le programme se termine à la fin du tableau VM.
3. La norme recommande d'utiliser le type de C `int32_t` si le compilateur de C est gcc (et pour la plupart des machine, ce type est en effet `int` normal). Le tableau est sauvegardé dans un fichier binaire dont l'extension du nom est `.lacc`. Puisque le type int32_t est vérifié compatible sur macOS, Linux (Ubuntu x64) et Windows, la portabilité est ainsi promue.

À LA COMPILATION DU LAC

Je présente ici l'algorithme de la compilation réalisé dans le programme.

La compilation commence par itérations des traitements des lexèmes identifiés en phase de l'analyse lexicale.

```
// Pseudo code simplifié
si lexème.type est chaine de caractère
      examiner l'environment (compilation de fonction? main? branche?)
      indiquer une entrée de chaine au mécanisme comptant le type
      noter dans VM
sinon si lexème est « : »
      examiner l'environment (compilation de fonction? main? branche?)
      // pas de définition de fonction emboitée
      examiner si la fonction est une fois définie
      configurer l'environment de la compilation d'une fonction
      noter dans VM
sinon si lexème est « ; »
      examiner l'environment (compilation de fonction? main? branche?)
      // erreur si l'environment n'est pas compilation d'une fonction
      ajouter l'information de la fonction dans le tableau de symbol
      // info donné par le mécanisme de type
      noter dans VM
      restaurer l'environment avant cette compilation de fonction
sinon // alors c'est un lexème registré dans le tableau, ou un nombre
      examiner l'environment (compilation de fonction? main? branche?)
      essayer de trouver la fonction définie dans le tableau
      si fonction trouvée est if
            sauvegarder l'info de type dans le pile
            initialiser l'info de type
            noter dans VM
      sinon si fonction trouvée est else
            examiner l'environment (compilation de fonction? main? branche?)
            sauvegarder l'info de type dans le pile
            initialiser l'info de type
            noter dans VM
      sinon si fonction trouvée est then
            examiner l'environment (compilation de fonction? main? branche?)
            si SIVRAI et SIFAUX existent
                  comparer les info de type à partir du pile
            si SIFAUX n'existe pas
                  vérifier si la branche ne prend rien et ne produit rien
            fin
            restaurer l'info de type avant la branche
```

```
            noter les changements de types dans l'info
            noter dans VM
      sinon si fonction trouvée est recurse
            configurer l'environnement à ne plus compter l'info de type
            noter dans VM
      sinon si une fonction est trouvée
            imiter l'exécution
            noter les changements de types dans l'info de type
            noter dans VM
      sinon // c'est un nombre
            convertir à un nombre
            noter une entrée d'entier
            noter dans VM
      fin
fin
```

À la fin de compilation, on met à jour la position de point d'entré, vérifier si la fonction main a suffisants entrées pour démarrer l'exécution.

Runtime

Le runtime prend un fichier compilée, le lire comme un fichier binaire à fin d'obtenir le tableau de VM, vérifie si la version de norme est compatible, et commencer à exécuter la fonction à la position indiquée par point d'entrée.

Pour l'exécution, le runtime partage le même processeur que l'interpréteur sauf le runtime connait plus de fonctions fournies par le processeur. Le runtime réalise des pile de données, de types et de retournes. Le principe de ces piles convient celui présenté dans les cours. Le runtime réalise aussi un mémoire dédié aux chaine de caractère circulaire, comme celui de l'interpréteur.

```
// Pseudo code simplifié
lire le fichier
vérifier la version de norme
si la version est compatible
      initialiser l'environnement
      ajuster position d'exécution au point d'entrée
      le type de fonction en train d'être exécuter est main
      début de boucle infini
            si c'est une fonction définie dans LAC
                  sauvegarder le type de fonction emboitant cette fonction
                  empile le pile de retourne
                  si la position dépasse la taille de table
                        sortir de boucle
                  fin
                  aller à la fonction
                  mettre à jour le type de fonction comme une fonction définie
            sinon // une fonction de base
                  exécuter la fonction
                  dépile le pile de retourne
                  restaurer le type de fonction sauvegardé
            fin
      fin de boucle infini
sinon
      générer des erreurs
fin
```

Comme l'interpréteur, s'il y existe l'erreur de type, l'erreur est générer par le processeur.

Processeur

Le processeur conforme aussi à la norme. Même l'ordre de fonction est défini dans le compilateur et le runtime.

Pour communiquer avec différent composant de programme, il y a une fonction particulière `linkProcessor` qui rattache le processeur soit à l'interpréteur soit au runtime, pour avoir l'accès aux piles, au mémoire et aux tableau.

Le processeur réalise des fonctions qui examinent leur-même les types, les erreurs de types sont générées ici, mais traitées différemment, par le composant utilisant le processeur. Voir l'information donnée en cours d'exécution pour être sûr qu'est-ce qui se passe.

À partir du cours, où les fonctions de base sont définies, comme `lit, str, fin, ., +, −, *, =, dup drop, swap, count, type, if, else, then, calculate` et `concatenate`, la norme définit de plus `&&, ||, /(division), !(not), ≤`. Ces fonctions ne sont pas forcément réalisés dans les différents projets, mais le runtime sera capable d'y identifier et générer les informations.

NOTE SUR RECURSE ET THEN

À cause du différentes interprétations du texte, les fonctions `recurse` et `then` sont aussi réalisées dans le processeur, mais le but de ces réalisations est de rendre le processeur compatible avec différentes compréhension et réalisation de ces deux fonctions, où on met les informations de fonction dans le VM, suivis par des adresses à sauter.

NOTE SUR `.`

Je réalise la fonction d'affichage en étendant la compétence de cette fonction, elle est capable d'imprimer les infos sur l'écrans de tous les types. Ce qui va rendre le résultat sorti du code `factorielle.lac` un peu différent que ce que l'on attend.

Calculate

Pour réaliser l'analyse syntaxique, la méthode utilisée est l'analyse BNF.

Le ficher est initialement écrit en utilisant les strings de C, il est maintenant adapté aux chaine de caractère de LAC, en d'abord convertissant la chaine de LAC vers la chaine de C.

Pour cette fonction, les espaces ne sont pas autorisés, et les multi

```
// Pseudo code simplifié
convertir la chaine vers une chaine de C
analyse lexical de la chaine en lisant chaque caractère (automate)
      stocker les lexèmes dans une liste (de type nombre, opérateur de faible
priorité, opérateur de haute priorité, parenthèse ouvrante, parenthèse fer-
mante)
BNF_somme // renvoie un arbre syntaxique
évaluation du valeur // traverser l'arbre syntaxique
```

Les détailles d'algorithm BNF se trouvent dans le fichier `calculate.c` en bas.

Les tests

Les compilations du programmes C sont testés sur macOS, Ubuntu x64 16.04 et Bash on Windows. Auprès les standards C89 et C99, la compilateur n'aurait généré aucune erreur.

Les fichiers de tests sont présentés dans le dossier `test`.

J'ai testé factorielle.lac, calculate, les branches conditionnelles balancée, non balancée (avec la vérification des types, dans le fichier `newtest.lac`), et les codes simples fournis par Hugo qui est très sympas.

Pour l'interpréter j'ai testé toutes les fonctions de base, qui marchent normalement en fournissant des résultats attendus (après debugging bien sûr).

La performance de l'analyse lexicale n'est pas toujours garantie, quelque fois si le code devient assez compliqué l'analyse tendent à générer une grosse chaine de caractère incluant les opérations LAC. Les raisons ne sont pas trouvées, l'expression régulière sont modifiées plusieurs fois, le cause de l'erreur n'est pas trouvé comme l'expression est déjà illisible.

Quelques mots finaux

La programmation de ces programmes ont duré longtemps, c'est même moi plus gros projet écrit seul.

Il y a sûrement beaucoup d'amélioration sur le style de codage et la réutilisation des lignes, ce sont les défauts au départ du projet.

Un regret principal c'est que je n'ai pas fini l'évaluation dynamique de ANY, qui serait plus performant sur le restraint du type au moment du compilation mais le temps est limité.

L'hypothèse faite pour la réalisation de l'analyse lexicale en utilisant une seule expression n'est toujours pas formellement vérifiée.

Il faut mieux faire la taille de tableau de symbol et la taille de VM, mémoire et degré de emboîtage des branches dynamique car statistiquement ces tailles sont proportionnelles à la longueur du code.

Il est aussi mieux si le traitement de recursive devient plus rationnelle. Parce que quelque fois le type est prédicable, la façon de le traiter à l'instant est trop imprécise.

Un répertoire complet avec toutes les modifications historiques sont disponibles via https://github.com/nicolasying/Compiler-L_AC

Finalement, merci à M. Alain CHILLÈS pour ses cours et ses conseilles, à Hugo BI pour ses idées innovantes et les débats sur la réalisation, aux autres camarades pour leurs aides.