Projet de Compilation Avancée
Interpréteur et Compilateur pour la Machine Universelle
(Version 2)

Darius Mercadier

25 février 2021

Présentation du sujet

Vue d'ensemble. Ce projet est composé de deux parties. Dans la première, vous implémenterez un intepréteur de bytecode de la Machine Universelle. Dans la seconde, vous développerez un compilateur pour le langage microJS, qui génèrera du bytecode de la Machine Universelle.

Rendu. Vous devrez rendre, avant le **21 mars 2021, 23h59 (UTC+1)**, une archive au format `.tar.gz` contenant :

- Le code de votre intepréteur, dans le dossier `um` (Section 1).
- Le code de votre compilateur, dans le dossier `cc` (Section 2).
- Un rapport nommé `rapport.pdf`, `rapport.md` ou `rapport.txt` (Section 3).
- Un fichier nommé `README.md` expliquant comment compiler et exécuter votre interpéteur et compilateur.

Le rendu du projet se fera par courrier électronique, aux adresses `darius.mercadier@gmail.com` et `emmanuel.chailloux@lip6.fr`.

Languages de programmation. Les langages de programmation sont laissés libres pour ce projet. Cependant, un langage "performant" (*e.g.,* C, Java, OCaml, Rust) est recommandé pour l'interpréteur (Section 1) ; un langage de script (*e.g.,* Perl, PHP) risquant d'être trop lent pour executer convenablement les fichiers de bytecode. De plus, vous êtes libres de ne pas utiliser le même langage pour l'interpréteur et le compilateur. Quel que soit le langage que vous utilisez, vous accompagnerez votre code d'un fichier permettant de le compiler (par exemple, un `Makefile` pour C ou OCaml, un fichier `Cargo` pour Rust...).

1 Machine universelle

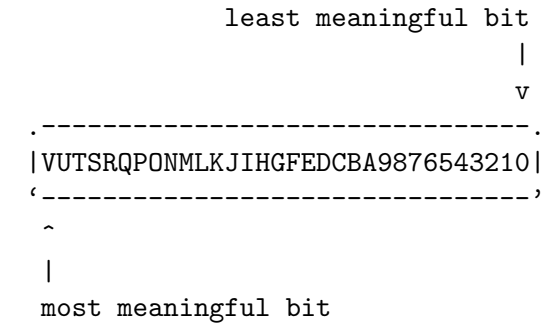
1.1 Introduction

Cette partie est inspirée d'un sujet de la conférence *ACM International Conference on Functional Programming* (ICFP) de 2006. La version originale du sujet peut se trouver via le lien : `http://www.boundvariable.org/task.shtml`. L'interpréteur de cette section sera contenu dans le répertoire `um` de votre projet.

1.2 La machine

On nous demande de créer la machine universelle qui est composée de :

— Une quantité infinie de plateaux de sable avec des cases pour 32 marques (en langage plus usuel : 32 *bits*). Un plateau ressemble à :



Chaque bit peut prendre les valeurs 0 ou 1. En utilisant le système de numération 32 bits, ces marques peuvent aussi encoder des entiers.

— 8 registres à usage général capables chacun de contenir un plateau.
— Une collection de tableaux de plateaux, chacun étant référencé par un identificateur 32 bits. On distinguera le tableau 0 des autres, il contiendra le programme de la machine. Ce tableau sera référencé comme étant le tableau '0'.
— Une console de 1 caractère capable d'afficher les glyphes du code ASCII et faisant l'input et l'output de `unsigned 8-bit characters`.

1.3 Comportement

La machine sera initialisée avec un tableau '0' dont le contenu sera lu depuis le parchemin de programme. Tous les registres seront initialisés à '0'. L'indice d'exécution pointera sur le premier plateau de sable qui devra avoir l'indice zéro.

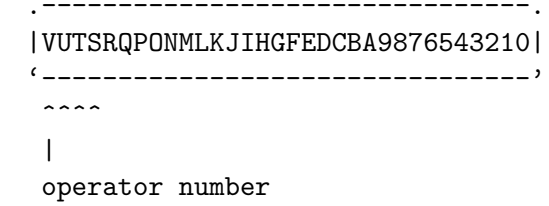
Quand on lit des programmes depuis des parchemins codés sur 8 bits, une série de 4 bytes (A,B,C,D) devra être interprétée comme suit :

— A sera le byte de poids fort.
— D le byte de poids faible.
— B et C seront intercalés dans cet ordre.

Une fois initialisée, la machine débute son cycle. À chaque cycle de la machine universelle, un opérateur devra être lu depuis le plateau indiqué par l'indice d'exécution. Les sections ci-après décriront les opérateurs que l'on peut récupérer. Avant que cet opérateur ne soit déchargé, l'indice d'exécution devra être avancé jusqu'au plateau suivant (s'il existe).

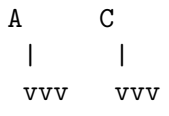
1.4 Opérateurs

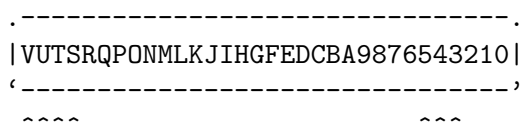
La machine universelle dispose de 14 opérateurs. Le numéro des opérateurs est décrit par les 4 bit de poids fort du plateau d'instructions.



1.4.1 Opérateurs standards

Chaque opérateur standard effectue un calcul en utilisant trois registres nommés : A,B et C. Chaque registre est décrit par un segment de 3 bits du plateau d'instruction. Le registre C est décrit par les 3 bits de poids faibles du plateau, le registre B par les 3 suivant et le registre A se trouve à la suite de B.





Les opérateurs standards sont les suivants :

Opérateur 0 Mouvement conditionnel
: Le registre A reçoit la valeur du registre B sauf si le registre C contient 0.

Opérateur 1 Indice de tableau
: Le registre A reçoit la valeur à l'offset contenu dans le registre C dans le tableau identifié par registre B.

Opérateur 2 Modification de tableau
: Le tableau identifié par le registre A est modifié à l'offset B par la valeur contenu dans le registre C.

Opérateur 3 Addition
: Le registre A reçoit la valeur contenu dans le registre B plus la valeur contenu dans le registre C modulo 2^{32}.

Opérateur 4 Multiplication
: Le registre A reçoit la valeur du registre B multipliée part la valeur du registre C modulo 2^{32}.

Opérateur 5 Division
: Le registre A reçoit la valeur du registre B divisée par la valeur du registre C.

Opérateur 6 Not-And
: Le registre A reçoit le `NAND` des registres B et C.

1.4.2 Autres opérateurs

D'autres opérateurs sont disponibles :

Opérateur 7 Stop
: Arrête la machine universelle.

Opérateur 8 Allocation
: Un nouveau tableau est créé avec une capacité égale à la valeur du registre C. Ce nouveau tableau est initialisé entièrement avec des plateaux à 0. Le registre B reçoit l'identificateur du nouveau tableau.

Opérateur 9 Abandon
: Le tableau identifié par le registre C est abandonné et une future allocation peut réutiliser son identificateur.

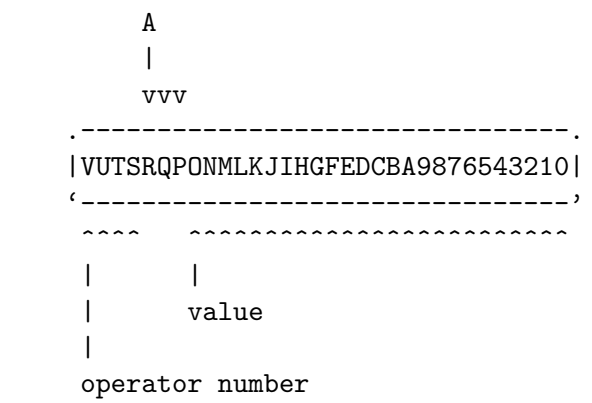
Opérateur 10 Sortie
: Écrit sur la console la valeur du registre C, seules des valeurs comprises entre 0 et 255 sont tolérées.

Opérateur 11 Entrée
: La machine universelle attend une entrée sur la console. Quand une donnée arrive, le registre C est chargé avec :
  — Tous ses bits positionnés à 1, si la fin de l'entrée est signalée.
  — La valeur de cette entrée sinon.

Opérateur 12 Chargement de programme
: Le tableau dont l'identificateur est B est dupliqué et sa copie remplace le tableau '0', quelle que soit sa taille. L'indice d'exécution doit être placé sur le plateau dont l'indice est contenu par le registre C.

1.4.3 Opérateurs spéciaux

Pour les opérateurs spéciaux, les registres utilisés ne sont pas décrits de la même façon. Les trois bits immédiatement après ceux décrivant l'opérateur donnent le numéro du registre à utiliser. Les 25 bits restant donnent une valeur qui devra être chargée dans le registre A.



Opérateur 13 Orthographe
La valeur indiquée doit être chargée dans le registre A.

1.5 Mesures de réduction de coûts

Tout comportement non décrit par le schéma précédent pourra mettre la machine en panne.

1.6 Tests

Afin de vous assurez du bon fonctionnement de votre intepréteur, vous téléchargerez le fichier `sandmark.umz` depuis le lien `http://www.boundvariable.org/task.shtml` et vous vous assurerez que la sortie produite par votre interpréteur sur ce fichier correspond à la sortie attendue (également fournie sur le lien ci-dessus).

2 Compilateur de microJS

2.1 Le language microJS

La grammaire du langage microJS est la suivante :

```
prog ::= <funs> <stmts>

funs ::= <fun> | <fun> <funs>

fun ::= "function" <id> "(" <params> ")" "{" <stmts> "}"

stmts ::= <stmt>
     | <stmt> "\n" <stmts>
     | <stmt> ";"  <stmts>

stmt ::=
     | var <id> "=" <expr>
     | let <id> "=" <expr>
     | <id> "=" <expr>
     | "if" "(" <expr> ")" "{" <stmts> "}"
```

```
     | "if" "(" <expr> ")" "{" <stmts> "}" "else" "{" <stmts> "}"
     | "while" "(" <expr> ")" "{" <stmts> "}"
     | "for" "(" <stmt> ";" <expr> ";" <stmt> ")" "{" <stmts> "}"
     | "return" <expr>

expr ::=
     | <int>
     | <string>
     | "(" <expr> ")"
     | <expr> <binop> <expr>
     | <unop> <expr>
     | <expr> "(" <args> ")"
     | <lambdaexpr>

lambdaexpr ::=
     | "lambda" "(" params ")" "{" <stmts> "}"

args   ::= expr | expr "," args
params ::= id | id "," params


binop ::= "+" | "-" | "*" | "/" | "%"
      | "==" | ">=" | "<=" | ">" | "<" | "&&" | "||"

unop ::= "-" | "!"

id = [a-zA-Z_] [0-9a-zA-Z_]*
int = [0-9]+
string = "[!-~]*"
```

Par exemple, la fonction factorielle peut s'écrire en MICROJS de la façon suivante :

```
function fact(n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

Certains aspects de JavaScript sont absent de MICROJS, afin de simplifier votre compilateur. En particulier, les tableaux, les objets (et donc les prototypes), les maps, les itérateurs et boucles `foreach`, la surcharge, les fonctionalités réactives (*e.g.,* `setTimeout`) et liées au browser (*e.g.,* `window.document`), les exceptions, et certains opérateurs comme `typeof` ou `===`. Le compilateur que vous allez implémenter n'est pas pour autant inutile : tout compilateur doit commencer par une base avant de s'attaquer aux fonctionalités avancées. Vous pouvez donc considérer ce projet comme un compilateur JavaScript basique, n'attendant plus qu'à être étendu à l'ensemble du langage.

2.2 Compilation manuelle

Afin de vous faire une première idée de la compilation de MICROJS vers la Machine Universelle, compilez à la main (*i.e.,* traduisez en bytecode de la Machine Universelle) les expressions MICROJS suivantes :

- `2 + 3`
- `2 + 3 + 4`

- `2000000000 + 2`
- `print "abc"`

Vous écrirez vos réponses dans votre rapport. Utilisez une notation "assembleur" pour écrire le bytecode (par opposition à "écrivez du binaire"). Par exemple :

```
ORTHO r0, 0
ADD r0, r0, r0
LOAD r2, r4
```

2.3 Compilateur basique de microJS

Dans un premier temps, certaines fonctionalités de microJS peuvent être laissées de coté. En particulier :

- Les opérateurs de comparaison (`>=`, `<=`, `>`, `<`).
- Les Booléens.
- Les chaînes de caractères.
- Les lambda expressions (`lambda f(...) { ... }`).

Pour implémenter le compilateur, nous vous proposons de suivre les étapes suivantes :

- Implémentez un lexeur, parseur, AST, et générateur de code pour les expressions composées de (petites) constantes et d'additions. Techniquement, une pile n'est pas indispensable à cette étape. Cependant, vu que vous aurez besoin d'une pile par la suite, nous vous suggérons d'en utiliser une dès maintenant.
- Ajoutez la multiplication, la soustraction, la division.
- Ajoutez le modulo. Celui-ci nécessite un peu plus de travail, dans la mesure où la Machine Universelle ne dispose pas de cet opérateur.
- Ajoutez les variables globales (déclarées avec `var`). Vous aurez pour cela besoin de modifier la structure de votre AST : un programme est désormais une suite d'instructions, et non une simple expression arithmétique.
- Ajouter le `if`/`then`/`else`.
- Ajoutez les boucles `while`. La boucle `for` est très semblable, et, pas conséquent, optionelle.
- Ajoutez une primitive `print_int` qui prend en argument un entier et l'affiche. Attention, il s'agit bien de compiler la fonction microJS `print_int` et non de rajouter un opcode à la Machine Universelle. Cette fonction vous sera utile par la suite pour implémenter vos tests.
- Ajoutez les fonctions globales (déclarées avec `function`) ainsi que les appels de fonction. A ce stade, lors d'un appel de fonction, la fonction doit être un identifiant (et non une expression) correspondant au nom d'une fonction globale définie au préalable.

Vous pouvez vous inspirer librement (*i.e.,* faites autant de copier-coller que vous souhaitez) de parseurs JavaScript ou microJS existant. En particulier :

- OCaml : `https://github.com/lsylvestre/upmc-3I018/tree/master/src/MLCompiler` (fichiers `ast.ml`, `parser.mly`, `lexer.mll`).
- Java : `http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2019/ue/LU3IN018-2020fev/Cours/programmes/` (dans l'archive `microjs-jcompiler-20180320.tar.gz`, dossier `src/microjs/jcompiler/frontend`).

2.4 Jeu de tests

Afin de s'assurer que votre compilateur est correcte, et que les évolutions de la suite du sujet ne "casseront" pas des fonctionalités déjà existantes, implémentez des tests couvrant *toutes* les fonctionalités de votre compilateur. Chaque test sera fait en 3 parties : la première compilera un fichier microJS vers un fichier de bytecode de la Machine Universelle, la seconde exécutera ce fichier à l'aide de votre interpréteur pour la Machine Universelle, et la troisième vérifiera que le test est un succès. Pour effectuer ces 3 phases, vous pouvez (si vous le désirez uniquement) vous inspirer du script Perl suivant : `https://dariusmercadier.com/assets/documents/script-verif-compile.pl`.

Dans votre rapport, vous préciserez quelles fonctionalités sont testées par quels tests (ou, inversement, quels tests testent quelles fonctionalités). Vous n'êtes pas obligé de parler de tous vos tests, mais vous devez me convaincre que votre jeu de test est complet.

2.5 Améliorations du compilateur

Une fois le compilateur décrit en Section 2.3 implémenté, vous ajouterez une ou plusieurs de ces extensions (dans l'ordre que vous le souhaitez ; elles sont mutuellement indépendantes). Si vous faites le projet seul, ces extensions sont optionelles. Si vous êtes en binôme, vous devez en faire au moins une ; les autres étant optionelles. Toute extension optionelle implémentée (correctement) vous octoiera un bonus sur la note du projet.

Fonctions anonymes (lambdas). Ajoutez les fonctions anonymes (lambdas) aux constructions supportées par votre compilateur. Vous penserez à modifier les appels de fonction dans votre AST : la fonction appelée n'est plus nécessairement un identifiant mais peut être une expression arbitraire (qui doit naturellement correspondre à une valeur fonctionnelle : soit identifiant une variable globale, soit une variable contenant un fonction anonyme, soit une fonction anonyme.

Opérateurs de comparaison. Commencez par ajouter l'opérateur `==`. Ensuite, ajoutez, au choix, `<=`, `>=`, `>` ou `<`. Vous incluerez les schémas de compilation de cette section à votre rapport, accompagnés de quelques explications.

Chaînes de caractères et surcharge. Ajoutez les chaines de caractères au langage. Vous devrez trouver un méchanisme permettant de distinguer dynamiquement un entier d'une chaîne de caractères. Implémentez ensuite une fonction de concaténation de chaînes de caractères. Pour finir, faites en sorte que l'opérateur + puisse être appliqué soit à 2 chaînes de caractères, soit à 2 entiers : dans le premier cas, une concaténation sera effectuée, tandis que dans le second, une addition sera effectuée. Le choix entre l'une de ces 2 opérations sera effectué dynamiquement.

Optimisation des fonctions récursives terminales. Lorsque la dernière instruction d'une fonction est un appel à cette fonction, on dit que la fonction est *récursive terminale*. Dans ce cas, il est possible d'optimiser le code généré en réutilisant la pile de cette fonction pour l'appel récursif au lieu de créer une nouvelle pile. Implémentez cette optimisation, et montrez, à travers un ou plusieurs exemples, que le code généré est en effet plus efficace.

3 Rapport

Le rapport de rendu devra être au format pdf, markdown ou texte. Il sera écrit en français ou en anglais, selon votre préférence. Il fera au maximum 10 pages. Il devra contenir :

- Description du travail produit : qu'avez vous réalisé par rapport à l'énoncé, qu'est-ce qui marche, qu'est-ce qui ne marche pas ?
- Une vision globale de l'architecture du projet.
- La description détaillée de vos choix techniques pour la machine universelle sans répéter le sujet, par exemple : représentation des éléments constituant la machine, la représentation des données (dans le programme)...
- La description du compilateur du langage MICROJS et vos schémas de compilation : comment compilez-vous les instructions MICROJS vers UM ? Quelles difficultées rencontrées ? Vous n'êtes pas obligé de détailler tous les schéma de compilation, mais donnez au minimum une description de haut niveau.
- L'architecture de votre système de tests (cf. Section 2.4).
- Toutes autres informations pertinentes.