Concept de langages de programmation : devoir 1

Stackulator : la calculatrice à pile du futur

par Nicolas Hurtubise et Guillaume Riou

> *Vous êtes complètement tarés*
>
> – Alexandre St-Louis-Fortier, étudiant à la maîtrise au DIRO, membre du laboratoire de parallélisme

Explications générales

Pour ce devoir, nous avons implanté une calculatrice à pile comprenant les instructions de base demandées telles que l'addition, la multiplication et la soustraction ainsi que le storage de variables. Nous avons également implanté quelques fonctionnalités supplémentaires utiles, telles que des comparaisons booléennes (`!` , `<` , `>` , `==`), des shifts arithmétiques en base 10 (`}` et `{`) et une opération pour renverser la pile (`'`).

Nous avons aussi implanté un système de boucles (délimitées par `[` et `]`) ainsi que des procédures nommées avec les lettres majuscules allant de A à Z

Lancé sans arguments, `stackulator` se comporte de la manière décrite dans l'énoncé du devoir, c'est à dire qu'il affiche '>' en attendant une entrée et qu'il affiche le dessus de la pile après avoir reçu et exécuté les instructions d'une ligne.

Pour rendre l'utilisation plus conviviale, nous avons ajouté quelques opérations :

```
-s, --silence : mode silencieux, n'affiche pas de caractère de "prompt"
    Le mode silencieux est fait pour être utilisé en conjonction avec un fichier
    de code `stackulator` (généralement des fichiers `.skii`) en argument.
-v, --verbose : mode verbeux qui affiche `x:(y)>` en attendant une entrée
    (où x est la hauteur de la pile et y est le nombre sur le dessus).
-h, --help : permet d'avoir un résumé des fonctionnalités possibles.
```

Détails d'implantation

Durant le développement de stackulator nous avons dû faire plusieurs choix de développement pour nous assurer que le programme utilise bien la mémoire et se comporte correctement.

Représentation des variables et nombres

Dans notre implantation de la machine à pile, les nombres sont représentés sous la forme de listes chainées de nombres en base 10 associées à un char (0 ou 1) représentant le signe. Cette représentation naïve des "bignums" nous a permis d'implanter les algorithmes requis sans trop nous casser la tête.

Nos variables sont stockées dans un tableau de 26 pointeurs de bignums, un par lettre de l'alphabet. Cette représentation est simple mais efficace.

Analyse et calcul de la réponse

Puisque le Stackulator est un langage simple, il est possible d'en analyser la syntaxe en lisant un charactère à la fois (en utilisant une fonction telle que `getc`).

La procédure d'analyse est simple :

i. lire les prochains caractères en entrée
ii. effectuer un traitement selon le caractère lu

Le traitement est composé de la manière suivante :

- Dans le cas d'un opérateur ou d'une commande à un caractère, exécuter l'opération correspondante
- Dans le cas de commandes pouvant être composées de plusieurs caractères, lire d'autres caractères jusqu'à avoir l'expression complète (ex.: entrée d'un nombre : `1...` , assignation de variables : `=x` , …)

En lisant le code, on peut remarquer que la fonction pour lire la prochaine entrée n'est pas getchar, ou getc, mais plutôt get_next.
Cette fonction nous permet :
- De récupérer les caractères lus par des sous-procédures qui n'en avaient pas besoin (ex.: la procédure d'entrée de nombres pourrait lire `1234*`)
- De permettre de lire des commandes depuis un fichier plutôt que depuis l'entrée standard
- De gérer les `contextes` pour les procédures et pour les boucles

En supplément des fonctionnalités demandées nous avons implanté des boucles et des procédures. Les boucles sont déclarées par `[` et `]` . Une boucle se termine quand le dessus de
la pile est égal à zéro ou si la pile est vide. Les procédures sont déclarées comme ceci: `:X "mettre des opérations ici"` et sont appelés en écrivant simplement X ou X est une lettre majuscule allant de A à Z.

Pour implanter facilement ces fonctionnalités, nous avons implanté ce que nous appelons des contextes. Un contexte est un genre de liste chainée sans index puisque le "début" est relié avec la "fin". Lorsqu'on rencontre un début de boucle ou de procédure, on construit un contexte contenant tous les symboles lus jusqu'à la fin de la boucle/procédure. Ensuite, dans le cas d'une boucle, on empile le contexte sur la pile de contexte et par la suite toutes les instructions suivantes seront prises dans le contexte sur le dessus de la pile. Dans le cas d'une procédure, on store le contexte dans un des emplacement de procédures et quand la procédure est appelée il suffit de mettre le contexte sur le dessus de la pile de contexte.

La sortie d'une boucle se fait lorsque le dessus de la pile est évalué à 0 à la lecure d'un `]` . Dans ce cas, il suffit de poper le contexte de la pile de contexte pour continuer le programme directement après la fin de la boucle. Dans le cas d'une procédure, dès que le symbole ';' est lu, le contexte est poppé du dessus de la pile de contexte et l'exécution du programme continue.

Gestion de la mémoire

Pour gérer la mémoire, nous avons implanté un système de comptage de référence simple pour les bignums : chaque bignum enregistre le nombre de références qui lui sont associées. Nous incrémentons le nombre de références vers un bignum quand celui-ci est assigné à une varible à laquelle il n'est pas déjà assigné ou lorsqu'il est ajouté quelque part sur la pile.

Lorsqu'un bignum n'est plus référencé par le stack ou par une variable, un appel à la fonction `bignum_destoroyah()` (notre fonction destructrice de bignum) décrémente le nombre de références et détruit le bignum et tous ses bigdigits si le compte de références tombe à 0.

Après avoir conçu ce système, nous avons testé notre programme avec Valgrind.

Ce fut une révélation : notre programme fuyait de partout ! Nous avons donc débuté une redoutable chasse à la fuite de mémoire dans tous nos algorithmes pour tenter de détecter les espaces mémoires alloués et non libérés.

Quand Valgrind a arrêté de nous alerter au sujet de fuites de mémoire, nous avons déterminé que notre programme gérait assez bien la mémoire.

Algorithmes d'addition, de soustraction et de multiplication

Algo d'addition et de soustraction

L'addition se fait de façon similaire à la méthode papier enseignée à la petite école : du chiffre le moins significatif au chiffre le plus significatif en conservant une retenue d'au plus 1 après chaque opération.

La soustraction est réduite à l'addition, `a - b = a + (-b)` . L'addition doit donc se soucier des signes au moment de faire l'addition chiffre à chiffre :

```
Si a et b sont de même signe :
```

```
    Pour chaque chiffre à la position i (en partant du moins significatif) :
        nouveau chiffre = carry (initialisé à zéro) + a[i] + b[i]
        carry = nouveau chiffre >= 10
        nouveau chiffre = nouveau chiffre mod 10

Si a et b sont de signes contraires :

    Pour chaque chiffre à la position i (en partant du moins significatif) :
        nouveau chiffre = carry (initialisé à zéro) + a[i] - b[i]
        carry = -(nouveau chiffre < 0)
        nouveau chiffre = (nouveau chiffre + 10) mod 10

Le signe de la somme est déterminé comme suit :
    - Du signe de a si a et b sont de même signes
    - Du signe du plus grand nombre en valeur absolue entre a et b sinon
```

Algo de multiplication

Dans nos premiers prototypes de la calculatrice, nos multiplications étaient effectuées par additions répétée. Par example, si nous voulions multiplier
25 par 5, nous additionnions 25 fois 5 avec lui même. Il va sans dire que ce procédé était peu efficace et extrêmement lent. Cette opération nous a par
contre permis d'avancer plus rapidement dans le développement d'autres aspects de notre machine à pile.

Un peu plus tard, nous nous sommes penchés sur le problème de l'algorithme de multiplications qui serait approprié pour les grands nombres.
Après un peu de recherche (merci wikipédia !), nous avons trouvés l'algorithme de Karatsuba qui nous semblait être une manière relativement simple et très efficace de multiplier. Seul problème, pour implanter cet algorithme, il faut une manière de multiplier deux nombre dans le cas de base ou un de ces nombres est composé d'un seul chiffre. Hereusement pour nous, nous avons réalisé qu'en utilisant notre multiplication par addition répétée et en "optimisant" en prenant le plus petit nombre pour le nombre d'additions nous avions déjà une multiplication en temps linéaire par rapport au nombre (Un maximum de 9 quand c'est utilisé par Karatsuba!). Pour implanter cet algortihme, nous avons du créé une nouvelle fonctionnalité : le shift arithmétique en base 10. La multiplication par Karatsuba est en théorie dans l'ordre de $O(n^{log_2 3})$.

Traitement des erreurs

Tous les messages d'erreurs sont envoyés sur stderr.

La macro `please_dont_segfault`

Puisque notre programme se doit d'être robuste et de résister aux conditions extrêmes de manque de mémoire, chaque `malloc` de variable est suivi de la macro `please_dont_segfault`, qui se charge essentiellement de tuer le programme avec un message d'erreur assez explicite : "FIN DE LA MÉMOIRE. FUYEZ !! FUYEZ PAUVRES FOUS !! (fonction X, ligne Y)"

Commandes non définies ignorées

Si une expression ne correspond à rien, on affiche le message d'erreur suivant :

```
Expression inconnue ignorée : αβε)(...
```

Et on ignore tous les caractères jusqu'au prochain espace/nouvelle ligne.

Manque d'arguments sur la pile

Lorsqu'un appel à un opérateur est effectué, on valide que la taille de la pile est suffisante pour effectuer l'opération. En cas de manque d'arguments, on affiche un message d'erreur du type :

```
+ nécessite deux opérandes, taille du stack insuffisante
```

Variables/procédures non définies

Lorsqu'on tente d'appeler une procédure inexistante ou d'ajouter sur la pile une variable nulle, on reçoit le message d'erreur correspondant :

```
La procédure `P` n'est pas définie.
```

ou

```
La variable `p` n'a pas été définie
```

Définition de procédure dans une boucle/procédure

Si une procédure est définie dans un contexte (une boucle ou une autre procédure) ce message d'erreur est lancé: Impossible de définir une procédure depuis une boucle ou une procédure

Sortie d'une procédure hors d'une procédure/ sortie d'une boucle hors d'une boucle

Il est bien entendu illégal de tenter de sortir d'une boucle ou d'une procédure si on y est pas entré. Ces messages d'erreurs s'affichent si l'utilisateur tente de faire ce genre de choses:

```
Instruction de sortie de boucle hors d'une boucle.
Instruction de sortie de procédure hors d'une définition de procédure.
```

Extras : opérations ajoutées à l'énoncé original

Turing complétude du langage `Stackulator`

Outre le fait qu'à peu près tous les caractères ASCII sont associés à une commande (voir l'annexe 1), certaines fonctionnalités sont particulièrement utiles :

- Les boucles : `[` et `]`
- L'inversion de pile : `'`

Judicieusement combinées, ces deux opérations permettent d'accéder à des éléments arbitraires de la pile, et donc de potentiellement pouvoir simuler une machine de Turing :

```
:A =a . ' a ';
:B [ 1 - =b . A b ] .;
:C =c . ' c B ';

# Copie le Nième élément sous le stack sur le top
:D =d B =e d C e;
# Remplace le Nième élément du stack par M
:E =d . =e . d B . e d C;
```

Pour avoir une preuve un peu plus formelle de la Turing-complétude de notre langage, nous avons implanté un interpréteur du langage ésotérique brainfuck (https://fr.wikipedia.org/wiki/Brainfuck), lui même prouvé Turing-complet (http://www.hevanet.com/cristofd/brainfuck/utm.b). Le code de notre interpréteur est en annexe de ce rapport.

Metakulator

Ayant un langage de programmation plutôt complet à notre disposition, nous avons décidé d'implémenter l'énoncé original dans ce dit langage.

La version implantée est en fait une relaxation du problème, aucun message d'erreur n'étant affiché et les variables étant toutes instanciées à 0.

ANNEXES :

1. Documentation du language :

Opérations disponibles

```
    | opérandes | résultat
```

symbole	dépilées	empilé
nombre	-	le nombre naturel entré
,	-	un nombre naturel entré sur l'entrée standard
`	-	un code ASCII lu depuis un char sur l'entrée
@	-	une copie du dessus de la pile
+	2	la somme des deux opérandes
-	2	la différence des deux opérandes
*	2	le produit des deux opérandes (méthode de Karatsuba)
==	2	1 si les opérandes sont égales, 0 sinon
>	2	1 si op1 > op2
<	2	1 si op1 < op2
&	2	1 si op1 != 0 *et* op2 != 0
|	2	1 si op1 != 0 *ou* op2 != 0
!	1	la négation binaire du dessus de la pile
?	3	op1 si op3 != 0, op2 sinon
{	1	op1 << 1 (les shifts sont en base 10)
}	1	op1 >> 1

Autres commandes

symbole	effet
.	pop le dessus de la pile
"…"	copie le texte entre " sur la sortie standard
	(" affiche ", \ affiche)
~	vide la pile
$	dump le contenu de la pile
%	dump les adresses contenues dans les variables
^	affiche le dessus de la pile
'	renverse la pile
\	affiche le charactère ASCII ayant le code du dessus de la pile

Variables

=a : assigne le dessus de la pile dans la variable a

_a : assigne NULL dans la variable a
a : push le contenu de la variable a sur la pile

Les variables disponibles sont les lettres a à z

Procédures

:A expressions; : assigne les expressions de A à ; exclusivement à la procédure A
A : appelle la procédure A.

Structure de contrôle

La seule structure de contrôle est la boucle :

```
20 =a .
10 [
    a 10 + =a .
    1 -
]
```

Le contenu entre crochets est exécuté tant que le top de la pile existe et est différent de zéro.

Pour obtenir un bloc conditionnel, il suffit de boucler exactement une fois :

```
...
a 100 < [
    "Vous avez entré un nombre inférieur à 100 !" . 0
] .
```

2. celsius2fahrenheit.skii

Ceci est un convertisseur de celcius à fahrenheit (avec des nombre à virgule !!!)

```
#!./stackulator -s
#C est abs(top du stack).
:C =z . 0 z - ;
#B est un modulo 10
:B @ 10 > [ . 10 - @ 10 > ] . @ 10 == [ . 10 - @ ] . ;
#A calcule la partie décimale et la met dans a et met la partie entière dans b.
:A @ 18 * 320 + @ 0 < [ . C 0 ] .  B =a . 18 * } 32 + =b ~;
"Entrez la température en Celcius : " ,
A a b
 "
Équivalent en Fahrenheit : " ^. "," ^
"
```

3. beer.skii

Ceci est un programme classique démontrant les boucles.

```
#!./stackulator -s
99 [ ^ " bottles of beer on the wall, " ^
" bottles of beer. Take one down and pass it around, " 1 -  ^
" bottles of beer on the wall." "
"]
```

4. metackulator.skii

Ceci est le programme metackulator dont il est question ci-dessus.

#!./stackulator -s

```
#----------- Fonctions pour lire un élément arbitraire sur le stack -----------#

# Envoie l'élément du top tout en bas
:A =a . ' a ' ;
# Répète A [stack_top] fois
:B [ 1 - =b . A b ] . ;
# Remet en ordre après un appel à B
:C =c . ' c B ' ;
# Copie le -nième élément du stack sur le top
:D =d B =e d C e ;
# Écrit sur le -nième élément du stack la valeur du top (le top est consommé)
:E =d . =e . d B . e d C ;
#------------------------------------------------------------------------------#


#------------------- Variables ------------------------------------------------#
0 =p # variable en cours d'input
0 =s # taille du stack
~
#------------------------------------------------------------------------------#

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 # Variables du metakulator

"-- Metakulator v1.0 --
Entrez Ctrl+A suivi de [enter] pour quitter
> "
1 [ . ` =i .

    # Si un nombre est entré
    i 47 > i 58 < & [ .
        # Si on est en train de construire un nombre
        p !! [ .
            { i 48 - +
        0 ] .
        # Si on est pas en train d'en construire un, on en crée un nouveau
        p ! [ .
            0 i 48 - +
            s 1 + =s .
            1 =p .
        0 ] .
    0 ] .

    # Si une addition est entrée et qu'il y a assez d'éléments sur le stack
    i 43 == s 1 > & [ .
        +
        s 1 - =s .
    0 ] .

    # Si une soustraction est entrée et qu'il y a assez d'éléments sur le stack
    i 45 == s 1 > & [ .
        -
        s 1 - =s .
    0 ] .

    # Si une multiplication est entrée et qu'il y a assez d'éléments sur le stack
    i 42 == s 1 > & [ .
        *
```

```
        s 1 - =s .
    0 ] .

    # Si une variable est entrée, on push sa valeur sur le stack
    i 96 > i 123 < & [ .
        i 97 - s + D
        s 1 + =s .
    0 ] .

    # Si on lit '=', on lit le nom de variable qui suit et on stock le résultat
    i 61 == [ .
        ` 97 - =i .
        i 0 1 - > i 27 < & [ .
            @ i s + E
        0 ] .
        61 =i . # On ne veut pas sortir de la loop principale
    0 ] .

    # Si on lit '\n', on affiche le top du stack puis le symbole du prompt
    i 10 == [ .
        ^
        "
"
        "> "
    0 ] .

    # Fin de la saisie d'un nombre
    i 48 < i 57 > | [ .
        0 =p .
    0 ] .
i 1 == ! ] .

^ "
"
```

5. brainfuck.skii

Voici le code de l'interpréteur `brainfuck` en `Stackulator`. Celui-ci est très lent et la majorité des exemples pris sur Internet ne fonctionneront pas
dans un temps raisonnable. Toutes les fonctionnalités fonctionnent et le "Hello World" de Wikipédia roule sans problème.

```
#!./stackulator -s

# v: variables, i: instructions brainfuck, p: pointeur d'instruction
# v3 v2 v1 v0 i0 i1 i2 ... il
#                ^
#                |p

# -- Définissions de fonctions ----------

    # Note: les variables a à e sont écrasées par les fonctions A à G

    # Soit N, la valeur du top du stack au moment d'appeler l'une des procédures
    # Soit M, la valeur sous N

    :A =a . ' a ';
    :B [ 1 - =b . A b ] .;
    :C =c . ' c B ';
```

```
    # Copie le Nième élément sous le stack sur le top
    :D =d B =e d C e;
    # Remplace le Nième élément du stack par M
    :E =d . =e . d B . e d C;
    # Insère M à N éléments sous le top du stack
    :F =d . =e . d B e d C;
    # Déplace l'élément N éléments sous le top vers le top du stack
    :G =d B =e . d C e;

    # Copie la prochaine instruction sur le top (sans argument)
    :I l p - 1 - D;

    # Copie la variable active sur le top
    # N : offset (nombre d'éléments par-dessus la liste d'instructions bf)
    :V l + x + v + D;

    # Écrit M dans la variable v en considérant un offset N
    :W l + x + v + E;

    # TODO Lis le num d'instruction sauvegardé dans la dernière boucle imbriquée

    # Debug
    :Z "w="w^.", v="v^.", l="l^.", p="p^.", i="i^.", x="x^. "
"
;

# -- Variables --------------------------
1 =w # Nombre de variables brainfuck instanciées (commence avec 1 variable)
0 =v # Pointeur de variable (0-based)
0 =l # Longueur du programme bf (0 = pas de programme)
0 =p # Pointeur d'instruction (0-based)
0 =i # Instruction actuelle (valeur ascii)
0 =x # Nombre de boucles imbriquées (0-based)
0 =j # Utilisé pour matcher les ouvertures/fermetures de boucles quand V = 0
~
# ---------------------------------------


# Input le code jusqu'à obtenir un \n
1 [
    ,
    ` # input un charactère
    l 1 + =l . # incrémente l

    @ 10 == 0 ==
] .. l 1 - =l .

' 0 ' # Valeur de la variable v0

1 [ . # Faire...
    I =i . # Lis la prochaine instruction BF

    # ,
    i 44 == [ ` 1 W .0 ] .

    # .
    i 46 == [ 1 V / . .0 ] .
```

```
    # +
    i 43 == [ 1 V 1 + 1 W .0 ] .

    # -
    i 45 == [ 1 V 1 - 1 W .0 ] .

    # <
    i 60 == v 0 > & [
        v 1 - =v .
    .0 ] .

    # >
    i 62 == [
        v 1 + =v .
        v w > v w == | [ # Si incrémenter le pointeur de variable crée une nouvelle
 variable...
            w 1 + =w . # On incrémente le compteur de variables
            ' 0 ' # On "push" une nouvelle variable au dessous de la pile
        .0 ] .
    .0 ] .

    # loop open : si variable actuelle est non-nulle
    i 91 == 1 V & [
        .
        p l x + F # insère l'adresse p à la bone place
        x 1 + =x . # incrémente x
    0 ] .

    # loop close
    i 93 == [
        l x + G 1 - =p .
        x 1 - =x . # décrémente x
    .0 ] . # loop close

    # loop open : si variable actuelle == 0
    i 91 == 1 V ! & [
        1 =j .
        1 [ .
            I =i .
            p 1 + =p .

            # Ouverture de boucle imbriquée
            i 93 == [
                j 1 - =j .
            .0 ] .

            # Fermeture de boucle imbriquée ou de la boucle principale
            i 91 == [
                j 1 + =j .
            .0 ] .

        j ] . # Tant que les boucles ouvertes ne matchent pas les boucles fermées
    .0 ] .

    p 1 + =p . # Incrémentation du pointeur d'instruction
    p l < ] . # ... Tant que le pointeur d'instruction est plus petit ou égal à l
"
"
```

6. Page web officielle du `Stackulator`

Visitez la page officielle du projet, générée en langage stackulator et gérée par un cgi

http://www-etud.iro.umontreal.ca/~hurtubin/cgi-bin/stackulator