Université de Strasbourg
UFR de Mathématique et d'Informatique

Calculatrice polonaise

Date limite de rendu : lundi 1er mai 2017

1 Notation polonaise

Pour représenter des expressions algébriques, il est possible d'utiliser trois types de notations : *préfixe*, *infixe*, et *postfixe*. La notation *préfixe* est aussi appelée notation *polonaise* ou notation de *Łukasiewicz* tandis que la notation *postfixe* est aussi appelée notation *polonaise inversée*. La notation la plus communément utilisée est la notation *infixe*. Chacune de ces notations présente des avantages et des inconvénients. La différence fondamentale entre ces notations est la position d'un opérateur par rapport à ses opérandes.

Notation. *En notation préfixe, l'opérateur précède ses opérandes.*

Notation. *En notation infixe, l'opérateur se situe entre ses opérandes.*

Notation. *En notation postfixe, l'opérateur se trouve après ses opérandes.*

Contrairement à la notation *infixe*, les notations *préfixe* et *postfixe* ne nécessitent pas de parenthèses dans les cas où les arités (le nombre d'opérandes requis) des opérateurs considérés sont connues et fixes. Par exemple, voici plusieurs notations possibles pour une même expression :

— en notation *préfixe* : « $-\ \times\ +\ 1\ 2\ 3\ 4$ »
— en notation *infixe* : « $(1\ +\ 2)\ \times\ 3\ -\ 4$ »
— en notation *postfixe* : « $3\ 2\ 1\ +\ \times\ 4\ -$ »

2 Devoir

Votre tâche est d'implémenter une calculatrice en notation polonaise. Le nom de l'éxécutable devra être `pc` (Polish Calculator). Le comportement de votre programme devra être similaire à celui des antiques, mais vénérables, `bc(1)` ou `dc(1)`. Par exemple :

```
$ echo '(1 + 2) * 3 - 4' | bc
5
$ echo '3 2 1 + * 4 - p' | dc
5
$ echo '- * + 1 2 3 4' | ./pc
5.0000
```

2.1 Structure de données

Vous devrez représenter les expressions sous forme d'arbres. Les noeuds non terminaux sont des opérateurs tandis que les feuilles sont des nombres. Les nombres devront être stockés dans des `double`.

Par exemple, pour l'expression « $-\ \times\ +\ 1\ 2\ 3\ 4$ », la racine est l'opérateur $-$. Ses noeuds fils sont l'opérateur $\times$ et le nombre 4. L'opérateur $\times$ a pour noeuds fils l'opérateur $+$ et le nombre 3. Enfin, l'opérateur $+$ a pour feuilles les nombres 1 et 2. Voir Figure 1.

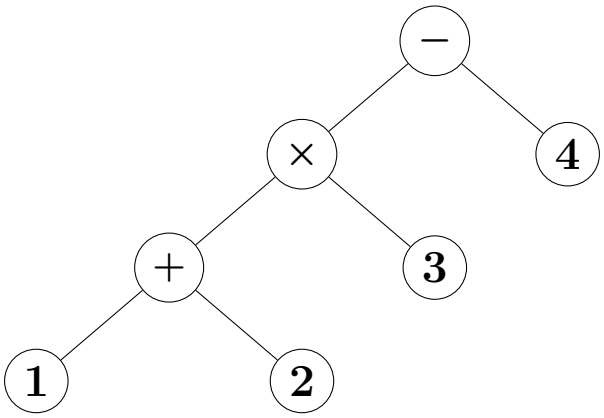

FIGURE 1 – Arbre représentant l'expression « − × + 1 2 3 4 ».

2.2 Opérateurs

Votre programme devra supporter des opérateurs unaires ainsi que des opérateurs binaires. La Figure 2 contient la liste des opérateurs attendus ainsi que le « symbole » correspondant.

opération	**symbole**
addition	+
soustraction	−
multiplication	*
division	/
minimum	`min`
maximum	`max`

(a) Opérateurs binaires attendus.

opération	**symbole**
racine carrée	`sqrt`
exponentielle	`exp`
logarithme népérien	`ln`
valeur absolue	`abs`
plancher (ou partie entière)	`floor`
plafond (ou partie entière par excès)	`ceil`

(b) Opérateurs unaires attendus.

FIGURE 2 – Opérateurs attendus.

2.3 Entrées/Sorties

2.3.1 Entrées

Les expressions fournies en entrée de votre programme seront représentées sous forme de chaîne de caractères. Les éléments (un opérateur ou un nombre) de l'expression sont séparés par un ou plusieurs espaces. Chaque expression est terminée par un saut de ligne ('`\n`'). Une ligne commençant par le caractère '`#`' est considérée comme un commentaire et doit être ignorée.

Votre programme devra lire ses entrées sur l'entrée standard `stdin`. En procédant ainsi, il sera alors possible, sans effort supplémentaire de votre part, d'envoyer des données à votre programme par l'intermédiaire d'un tube directement depuis le `shell` ou bien de l'utiliser de manière « interactive ».

```
$ printf '+ 1 2\n- * 3 4 5\n' | ./pc
3.0000
7.0000
```

```
$ ./pc
>>> + 1 2
3.0000
>>> - * 3 4 5
7.0000
>>>
```

Si vous écrivez correctement le code qui permet de lire les entrées, il suffira d'envoyer `EOF` (raccourci clavier `<Ctrl + D>` dans la plupart des `shell`) pour quitter le programme.

2.3.2 Sorties

La sortie doit se faire sur la sortie standard `stdout`. Chaque réponse doit se trouver sur une ligne séparée (ne pas oublier le caractère `'\n'`, même s'il n'y a qu'une seule sortie). Pour chaque expression valide, seul le résultat doit être écrit sur `stdout`. Ce résultat doit être affiché avec exactement 4 chiffres après la virgule. En cas d'erreur, il faudra afficher `ERROR`.

Vous êtes libres d'afficher (ou de ne rien afficher) tout ce que vous jugez utile sur la sortie d'erreur `stderr`.

```
$ echo '+ 1.5 min 3 2.5' | ./pc
4.0000
$ echo '+ 1 min 3' | ./pc
Error: missing operand
+ 1 min 3
    ^~~
ERROR
```

Dans l'exemple d'erreur ci-dessus, `ERROR` est affiché sur `stdout` tandis que les trois premières lignes facultatives sont affichées sur `stderr`.

2.4 Conseils

Pour évaluer le résultat d'une expression, vous pouvez utiliser un algorithme de parcours en profondeur. Voici une liste *non exhaustive* de headers/fonctions de la bibliothèque standard qui vous seront *peut-être* utiles :

- `stdlib.h` : **`strtod(3)`**
- `stdio.h` : **`fopen(3)`**, **`fgets(3)`**, **`sscanf(3)`**
- `string.h` : **`strtok(3)`**, **`strstr(3)`**
- `tgmath.h` : **`fmin(3)`**, **`fmax(3)`**, **`sqrt(3)`**, **`log(3)`**, **`ceil(3)`**

Si vous testez votre programme en mode interactif, pensez à utiliser un *wrapper* tel que **`rlwrap(1)`** ou **`ledit(1)`**.

3 Bonus

Vos propositions de fonctionnalités supplémentaires sont les bienvenues **si et seulement si** ce qui vous est demandé ci-dessus fonctionne **parfaitement**.

Voici quelques exemples d'améliorations :

- Arités variables pour certains opérateurs (il faut alors supporter la présence de parenthèses dans une expression) :

  ```
  >>> * (+ 1 2 3 4) 3
  30.0000
  ```

- Variables au sein d'une expression. L'opérateur '`=`' doit :
  - stocker sa deuxième opérande dans une variable dont le nom est la première opérande (vous pourrez utiliser une table de hachage)
  - retourner la deuxième opérande

  ```
  >>> + = a 3 a
  6.0000
  ```

4 Modalités de rendu

Ce devoir est **à réaliser seul** en langage `C`, révision `C11`. Le travail rendu devra être déposé sur la plateforme Moodle au plus tard le **lundi 1er mai 2017** sous la forme d'une archive `tar` (éventuellement compressée). Le nom de cette archive doit impérativement suivre la convention `nom-prenom.tar[.gz]`. Le désarchivage doit produire un dossier `nom-prenom` qui contient le travail soumis. À titre d'exemple, voici une commande `tar(1)` valide :

```
$ tar -czf "nom-prenom.tar.gz" "nom-prenom"
```

Cette archive devra contenir les sources de votre programme, un `Makefile` ainsi qu'un court rapport (choix d'implémentation, difficultés rencontrées, etc). Cette archive ne doit en aucun cas contenir de fichiers générés par le processus de compilation (fichiers objets, bibliothèques, etc) ou par votre outil de développement favori (fichiers `.swp`, `.workspace`, etc).

A Annexe

Pour vous aider lors du développement du programme, vous pouvez vous inspirer du `Makefile` en Figure 4 ainsi que du script de test en Figure 5.

A.1 `Makefile`

```
.PHONY: archive clean distclean
CFLAGS=-std=gnu11 -pedantic -O3 -march=native -Wall -Wextra -g
LDFLAGS=-lm

pc: main.o expression.o
	$(CC) $^ -o $@ $(LDFLAGS)

main.o: main.c expression.h
expression.o: expression.c expression.h

NAME=$(shell basename $(shell pwd))
archive:
	tar -czf $(NAME).tar.gz --transform="s,^,$(NAME)/," *.c *.h Makefile test
clean:
	rm -rf *.o pc
distclean: clean
	rm -rf *.tar.gz
```

FIGURE 4 – Exemple de `Makefile`.

A.2 Exemple de tests

Fichier `check.bash` :

```
#!/bin/bash

if ! test -x ./pc; then
  echo "Error: no executable \"./pc\"" 1>&2
  exit 1
fi

if diff results <(./pc <tests 2>/dev/null); then
  printf "\033[32mSUCCESS\033[0m\n"
  exit 0
else
  printf "\n\033[31mFAIL\033[0m\n"
  exit 1
fi
```

FIGURE 5 – Script de test (voir le jeu d'expressions et de résultats en Figure 6).

Fichier `tests` :

```
# 42 = 42.0000
42
# (1 + 2) * 3 - 4 = 5.0000
- * + 1 2 3 4
# 1.5 + min(3, 2.5) = 4.000
+ 1.5 min 3 2.5
# 1.5 + min(3, ???) = ERROR
+ 1.5 min 3
# 2/3 + 4 = 4.6667
+ / 2 3 4
# ln(exp(3)) = 3.000
ln exp 3
# sqrt(4) + floor(3.5) = 5.000
+ sqrt 4 floor 3.5
# sqrt(4) + 5 / 2 = 4.5000
+ sqrt 4 / 5 2
# sqrt(4) + floor(5 / 2) = 4.000
+ sqrt 4 floor / 5 2
# abs(-3) = 3.0000
abs -3
# abs(3) = 3.0000
abs 3
# 1 + floor(3.14) = 4.0000
+ 1 floor 3.14
# 1 + ceil(3.14) = 5.0000
+ 1 ceil 3.14
```

(a) Jeu d'expressions de test.

Fichier `results` :

```
42.0000
5.0000
4.0000
ERROR
4.6667
3.0000
5.0000
4.5000
4.0000
3.0000
3.0000
4.0000
5.0000
```

(b) Résultats correspondants.

FIGURE 6 – Entrées et résultats correspondants pour le script de test en Figure 5.