Partie optionnelle des TPs: les calculatrices programmables

Le langage des calculatrices est étendu de manière à pouvoir y programmer n'importe quelle *fonction calculable* de $\mathbb{Z}$ dans $\mathbb{Z}$: c'est donc une introduction à la notion de *fonction calculable* – définie dans la seconde partie de TL2. Ces extensions correspondent principalement à ajouter quelques opérateurs dans le langage **input** des calculatrices. Chacun de ces opérateurs a la même sémantique dans les deux langages, il n'y a que leurs syntaxes qui diffèrent. Aussi, si vous n'avez pas beaucoup de temps, il est préférable de réaliser l'extension du TP1, car la syntaxe de celle du TP2 est légèrement plus compliquée à traiter. Vous pouvez attaquer l'extension du TP1 dès que vous avez fini le TP1. Par contre, à la séance encadrée du TP2, vous devrez finir le TP2 **avant** de finir cette extension du TP1. En effet, finir le TP1 et le TP2 (sans les extensions) est un pré-requis de l'examen de TL2. Par contre, avoir traité les extensions n'est pas requis pour l'examen de TL2. Réaliser ces extensions n'est pas très long (1 heure au total ?), même si l'énoncé semble assez long (il y a beaucoup de d'explications sur l'utilisation des langages à implémenter).

1 Calculatrice programmable en syntaxe préfixe (TP1)

L'extension est définie en deux sous-extensions : la première sous-extension permet de définir des fonctions par cas, la seconde sous-extension permet de faire des boucles.

1.1 Un langage pour définir des fonctions par cas

On étend la sémantique de l'opération "$\ell[i]$" pour $-|\ell|+1 \leq i \leq 0$ avec la définition suivante :

$$\ell[i] = \begin{cases} |\ell| \text{ si } i = 0 \\ \ell[|\ell|+1+i] \text{ si } i < 0 \end{cases}$$

Cela permet d'utiliser la liste de calculs comme une pile. En particulier, "$\ell[-1]$" – qui est équivalent à "$\ell[\ell[0]]$" – retourne le dernier élément calculé. Ainsi, une fonction à un argument lira son argument dans "$\ell[-1]$" et écrira le résultat au même endroit. Pour cela, il faut permettre aux fonctions de modifier les calculs déjà écrits dans la pile. On ajoute donc une opération de modification de la pile notée "store(ℓ, i, n)" qui, lorsque $i \in [-|\ell|+1, |\ell|]$ et $n \in \mathbb{Z}$ et $i = 0 \Rightarrow n \in [0, |\ell|]$, correspond à une liste de $\mathbb{L}$ vérifiant :

— store$(\ell, i, n)[i] = n$

— et, pour tout $j \geq 0$, tel que $i \geq 0 \Rightarrow j \neq i$ et $i < 0 \Rightarrow j \neq |\ell|+1+i$,
store$(\ell, i, n)[j] = l[j]$

Remarquons, que "store$(\ell, 0, n)$" correspond à une opération de dépilement : on supprime tous les éléments dont l'indice est plus grand que n. Pour simplifier la syntaxe, on définit aussi

store$(\ell, 0, n)$ pour $n < 0$, comme équivalent à store$(\ell, 0, |\ell|+1+n)$. Cette opération peut être implémentée en PYTHON via une modification en place de la liste avec l'instruction "**del** `l[n:]`".

Comme expliqué ci-dessus, on va représenter une fonction f de $\mathbb{Z} \to \mathbb{Z}$ par un calcul qui lit son argument n en sommet de pile, et écrit son résultat $f(n)$ au même endroit. Par exemple, la fonction définie mathématiquement par "$x \mapsto 2 \times x$" sera représentée dans notre sémantique par "$\ell \mapsto$ store$(\ell, -1, 2 \times \ell[-1])$" : une fonction, notée `DOUBLE`, qui prend une pile ℓ et retourne cette même pile en ayant doublé l'entier à son sommet.

On étend maintenant la lexicographie de la calculette avec les 4 nouveaux terminaux suivants

`COLON` := {`':'`} `SHARP` := {`'#'`} `OPAR` := {`'('`} `CPAR` := {`')'`}

Rappelons que le prochain lexème reconnu par l'analyseur lexical est le plus long possible, de sorte que la suite de caractères `#1` `#0` `#-1`
correspond à la suite de terminaux `VAR↑1` `VAR↑0` `SHARP MINUS INT↑1`

La syntaxe des expressions est étendue avec :

$$\begin{aligned} \textbf{exp}{\downarrow}\ell{\uparrow}n ::= \;& \ldots \\ \mid\; & \texttt{SHARP}\ \textbf{exp}{\uparrow}i \quad n := \ell[i] \end{aligned}$$

Ainsi, les expressions `'#2'` et `'#+ 1 1'` ont la même sémantique "$\ell[2]$", bien qu'ayant des structures grammaticales très différentes.

Finalement, on étend le profil de **input** en **input**$\downarrow\mathbb{L}\downarrow\mathbb{Z}\uparrow\mathbb{L}$, de sorte que dans une dérivation de "**input**$\downarrow\ell\downarrow e\uparrow\ell'$", l'attribut e (pour *execute*) lorsqu'il est négatif ou nul, indique qu'il faut ignorer les calculs. La calculette est initialisée avec $e = 1$. Pour définir la sémantique de "**input**", on introduit maintenant l'opérateur noté "$(e?v_1 : v_2)$" tel que

$$(e?v_1 : v_2) = \begin{cases} v_1 \text{ si } e > 0 \\ v_2 \text{ sinon} \end{cases}$$

Formellement, le langage de **input** est maintenant défini par :

$$\begin{aligned} \textbf{input}{\downarrow}\ell{\downarrow}e{\uparrow}\ell' ::= \;& \texttt{QUEST}\ \textbf{exp}{\downarrow}\ell{\uparrow}n\ \textbf{input}{\downarrow}(e?\ell \oplus n : \ell){\downarrow}e{\uparrow}\ell' \\ \mid\; & \texttt{COLON}\ \textbf{exp}{\downarrow}\ell{\uparrow}i\ \textbf{exp}{\downarrow}\ell{\uparrow}n\ \textbf{input}{\downarrow}(e?\text{store}(\ell, i, n) : \ell){\downarrow}e{\uparrow}\ell' \\ \mid\; & \texttt{OPAR}\ \textbf{exp}{\downarrow}\ell{\uparrow}c\ \textbf{input}{\downarrow}\ell{\downarrow}(e?c : e){\uparrow}\ell_0\ \textbf{input}{\downarrow}\ell_0{\downarrow}e{\uparrow}\ell' \\ \mid\; & \texttt{CPAR} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ell' := \ell \\ \mid\; & \texttt{END} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ell' := \ell \end{aligned}$$

Informellement, l'instruction `QUEST` (quand elle n'est pas ignorée) ne change pas de sémantique par rapport au TP1. L'instruction `COLON` (quand elle n'est pas ignorée) correspond à l'opération "store" sur l'état courant de la pile. L'instruction `CPAR` a la même sémantique que `END` : elle termine la liste de calculs en cours. Enfin, l'instruction `OPAR` peut être interprétée (quand elle n'est pas ignorée) comme un traitement conditionnel : si l'expression en argument est strictement positive, alors on commence par exécuter la première liste de calculs (de **input**), puis, dans tous les cas, on exécute la seconde liste de calculs.

Par exemple, la fonction `DOUBLE` est codée par `: -1 * 2 #-1` qui se comprend ainsi :

`:`	écrire dans la pile
`-1`	à la dernière position
`* 2 #-1`	deux fois la valeur en dernière position de la pile.

On va maintenant tester cette fonction en Python (ou plus exactement, on va tester l'exécution de la calculette sur cette fonction). Pour cela, la figure 1 commence par définir une fonction Python `test` qui prend une fonction Python `f`, une chaîne `code` qui est censée représenter cette fonction Python dans le langage de la calculette, et un entier `arg` et vérifie que le résultat de la calculette sur `code` dans une pile initialisée à `[arg]` vaut bien `[f(arg)]`. La figure 2 l'utilise pour tester `DOUBLE` sur 7 et 15.

Pour illustrer une fonction avec une définition par cas, on considère la fonction Python prédéfinie "`abs`" (valeur absolue). Son code dans la calculette préfixe est donné à la figure 3. L'algorithme de cette fonction est le suivant : ligne 1, étant donné n en sommet de pile, empiler $-n$; ensuite, lignes 2 et 3, si $-n$ est positif, remplacer la valeur n de la pile (en position `-2` donc avant-dernière position) par $-n$; enfin, ligne 4, dépiler un élément.

```
import test_parser
from pcalc import parse
test_parser.PARSER_UNDER_TEST=parse
from test_parser import test_result

def test(f, code, arg):
    test_result("?" + str(arg) +
                " " + code,
                [f(arg)])
```

Figure 1 – Test avec la syntaxe préfixe

```
def double(x):
    return 2*x

DOUBLE = """
: -1 * 2 #-1
"""

test(double, DOUBLE, 7)
test(double, DOUBLE, 15)
```

Figure 2 – Test de `DOUBLE`

```
? - #-1
(#-1
   : -2 #-1)
: 0 -1
```

Figure 3 – `ABS` en syntaxe préfixe

```
- #-1 ?
#-1 (
   -2: #-1 ?)
0: -1 ?
```

Figure 4 – `ABS` en syntaxe infixe

1.2 Ajout des boucles

On aimerait maintenant pouvoir programmer des boucles. La sémantique formelle d'un langage de programmation avec des boucles est assez complexe : la sémantique doit elle-même manipuler le programme exécuté pour effectuer des dépliages. Comme indiqué dans le chapitre 1 du cours, l'idéal est de décomposer la définition en deux parties. D'abord définir un analyseur syntaxique qui représente chaque programme sous forme d'un arbre, puis définir la sémantique par réécriture sur les arbres. Cette architecture donne en particulier une implémentation qui ne recommence pas plusieurs fois l'analyse syntaxique de la même boucle.

Dans le cadre de ce TP, on va suivre une approche plus simple : on utilise la possibilité de déplacer le curseur de l'analyseur lexical n'importe où dans le flot d'entrée. [1] Dans l'analyseur

1. Cette possiblité n'existe que sur certains types de flot, comme ceux créés à partir de chaînes de caractères. Elle n'existe pas quand le flot est l'entrée standard. Le mécanisme utilisé ici pour coder les boucles ne fonctionnera donc pas sur n'importe quel type de flot. En particulier, on ne pourra pas le tester de manière interactive depuis l'entrée standard.

lexical `lexer` fourni, le flot d'entrée correspond à la variable globale `in_stream`. La fonction `in_stream.tell()` retourne la position du prochain caractère à lire dans `in_stream` (sous la forme d'un entier naturel). On peut alors faire poursuivre l'analyse syntaxique de la suite de terminaux à partir d'une telle position `pos`, via la fonction `goto` suivante. L'argument `exe` indique juste si on doit ignorer ou pas cette action.

```
def goto(exe, pos):
    if exe:
        lexer.in_stream.seek(pos)
        init_parser(lexer.in_stream)
```

On étend donc la lexicographie de la calculette avec les 2 nouveaux terminaux suivants

$$\texttt{POS} := \{\texttt{'p'}\} \qquad \texttt{GOTO} := \{\texttt{'g'}\}$$

Le terminal `POS` a le profil $\texttt{POS}\uparrow\mathbb{N}$: l'attribut attaché correspond à la position – obtenue via `in_stream.tell()` – du caractère qui suit le `'p'` reconnu.

Le langage **input** est ensuite étendu avec les règles suivantes :

$$\begin{aligned}\textbf{input}\downarrow\ell\downarrow e\uparrow\ell' ::= \;& \ldots \\ \mid\; & \texttt{POS}\uparrow n\ \textbf{input}\downarrow(e?\ell \oplus n : \ell)\downarrow e\uparrow\ell' \\ \mid\; & \texttt{GOTO}\ \textbf{exp}\downarrow\ell\uparrow n\ \{\ \texttt{goto}(e, n)\ \} \qquad \ell' := \ell\end{aligned}$$

L'instruction `'p'` a donc pour effet (si elle n'est pas ignorée) d'empiler la position courante du curseur. L'instruction `'g` n`'` termine la liste de calculs en cours et (si elle n'est pas ignorée) positionne l'analyseur syntaxique sur le premier terminal qui suit la position n, ce qui a pour effet de modifier l'arbre d'analyse en construction (l'analyse syntaxique se poursuit dans le plus "proche" non-terminal de l'arbre pas encore analysé). À titre d'exemple, la figure 5(b) donne la définition de la factorielle pour cette calculatrice. Lorsque la valeur initialement en sommet de pile est un nombre positif ou nul n, l'analyse de ce code va produire un arbre d'analyse de $n+1$ traitements conditionnels (de l'opérateur `OPAR`) imbriqués via le fils droit.

```
def fact(n):
    if n > 1:
        return n*fact(n-1)
    return n
```

(a) Python

```
?#-1
p
: -2 + #-2 -1
(#-2
   : -3 * #-2 #-3
   g #-1)
: 0 -2
```

(b) Syntaxe préfixe

```
#-1 ?
p
-2: #-2 - 1 ?
#-2 (
   -3: #-2 * #-3 ?
   #-1 )
0:-2 ?
```

(c) Syntaxe infixe

FIGURE 5 – Trois définitions de la factorielle

2 Calculatrice programmable en syntaxe infixe (TP2)

La calculette du TP2 utilise la même lexicographie que celle du TP1 (hormis le terminal `GOTO` devenu superflu). L'opérateur `SHARP` est ajouté au niveau du langage **exp**, avec la

priorité 0, de la même manière qu'en section 1.1 :

$$\begin{array}{rcll} \textbf{exp}{\downarrow}\ell{\uparrow}n & ::= & \ldots & \\ & | & \texttt{SHARP}\ \textbf{exp}{\uparrow}i & n := \ell[i] \end{array}$$

C'est donc un opérateur dont la syntaxe est très similaire au moins unaire.

Comme au TP1, on considère ici que le terminal spécial `END` peut faire partie des terminaux qui apparaissent en partie droite des règles des BNF. C'est la convention ANTLR, mais ça ne correspond pas à la convention usuelle des grammaires (qui elle était utilisée pour le TP2). Comme la convention classique, cette convention ANTLR considère que le terminal `END` ne correspond à aucun lexème de l'entrée. Par contre, elle considère qu'on ajoute pour l'analyse syntaxique une suite *infinie* de terminaux `END` après les terminaux de l'entrée (c'est le point de vue donné par `parse_token`). De plus, cette convention considère que l'analyseur LL traite le plus long préfixe de l'entrée accepté par **input** en ignorant le suffixe résiduel. [2]

Le langage **input** du TP2 est généralisé de la manière suivante (avec la convention ANTLR). Voir la section 1 pour les explications sur la sémantique.

$$\begin{array}{rcll} \textbf{input}{\uparrow}\ell & ::= & \textbf{block}{\downarrow}[]{\downarrow}1{\uparrow}\ell' & \\ \textbf{block}{\downarrow}\ell{\downarrow}e{\uparrow}\ell' & ::= & \texttt{END} & \ell' := \ell \\ & | & \texttt{CPAR} & \ell' := \ell \\ & | & \textbf{exp}{\downarrow}\ell{\uparrow}n\ \texttt{CPAR}\ \{\ \texttt{goto}(e,n)\ \} & \ell' := \ell \\ & | & \texttt{POS}{\uparrow}n\ \textbf{block}{\downarrow}(e?\ell \oplus n : \ell){\downarrow}e{\uparrow}\ell' & \\ & | & \textbf{exp}{\downarrow}\ell{\uparrow}n\ \texttt{QUEST}\ \textbf{block}{\downarrow}(e?\ell \oplus n : \ell){\downarrow}e{\uparrow}\ell' & \\ & | & \textbf{exp}{\downarrow}\ell{\uparrow}i\ \texttt{COLON}\ \textbf{exp}{\downarrow}\ell{\uparrow}n\ \texttt{QUEST}\ \textbf{block}{\downarrow}(e?\text{store}(\ell,i,n) : \ell){\downarrow}e{\uparrow}\ell' & \\ & | & \textbf{exp}{\downarrow}\ell{\uparrow}c\ \texttt{OPAR}\ \textbf{block}{\downarrow}\ell{\downarrow}(e?c : e){\uparrow}\ell_0\ \textbf{block}{\downarrow}\ell_0{\downarrow}e{\uparrow}\ell' & \end{array}$$

Les figures 4 et 5(c) illustrent la correspondance entre la syntaxe préfixe et la syntaxe infixe (noter que l'opérateur `GOTO` préfixe correspond à un opérateur unaire suffixe noté `CPAR`).

En conclusion, en modifiant un peu les conventions sur la représentation des fonctions donnée en section 1, on peut utiliser ce langage pour définir des fonctions qui commencent par définir des sous-fonctions, puis les appellent, les passent en argument d'autres sous-fonctions, etc. Par exemple, la figure 6 montre un encodage "fidèle" d'une fonction `quad` Python qui fait deux appels sur une sous-fonction `double`. Ici, le code de la fonction principale prend son argument (et rend son résultat) dans l'indice absolue `#1` de la pile (qui doit initialement contenir uniquement cet argument). Par contre, la sous-fonction prend son argument (et rend son résultat) dans l'indice relatif `#-3` (pour une pile quelconque, mais suffisamment profonde), les indices `#-2` et `#-1` servant à encoder le mécanisme de retour dans le contexte d'appel : l'indice `#-1` contient l'adresse de retour et l'indice `#-2` contient `0` si l'appel a été fait et `1` sinon [3].

2. Autrement dit, pour reconnaître *exactement* $\{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$ avec la convention ANTLR, il faut consommer `END` explicitement dans la BNF. Par exemple, avec

$$\textbf{S} ::= \textbf{A}\ \texttt{END} \qquad \textbf{A} ::= \texttt{a}\ \textbf{A}\ \texttt{b} \mid \epsilon$$

Ce changement de convention – promu par ANTLR – permet d'étendre **notablement** l'expressivité des BNF LL. En effet, le langage $\{\texttt{a}^n\texttt{b}^p \mid n \geq p\}$ n'est reconnu par aucune grammaire LL(1) avec la convention classique. Avec la convention ANTLR, il est reconnu par la BNF LL(1) suivante :

$$\textbf{S} ::= \textbf{A}\ \texttt{END} \qquad \textbf{A} ::= \texttt{a}\ \textbf{A}\ \textbf{B} \mid \epsilon \qquad \textbf{B} ::= \texttt{b} \mid \texttt{END}$$

On avait adopté ce point de vue au TP1, sans vraiment l'expliciter.

3. Comme l'instruction `'p'` sauvegarde l'adresse actuelle, l'adresse de retour d'une fonction se trouve avant son appel. L'indice `#-2` permet donc de savoir s'il faut ou non faire l'appel.

```
def quad(x):
    def double(y):
        return 2*y
    return double(double(x))
```

```
0 ?
p
#2 (
  -3: 2 * #-3 ?
  -2: 0 ?
  #-1 )
2: 1 ?
#1 ?
1 ?
p
#-2 ( #3 )
0: -2 ?
1 ?
p
#-2 ( #3 )
0: -2 ?
1: #-1 ?
0: 1 ?
```

FIGURE 6 – Exemple d'encodage d'une sous-fonction (en syntaxe infixe)

Cette technique permet en fait d'encoder assez directement n'importe quelle fonction récursive sur les entiers (une fonction récursive étant simplement définie à l'aide une sous-fonction qui s'appelle elle-même). Le fichier fourni `bigtest_calc.py` en donne quelques exemples. Parmi ceux-ci, la figure 7 définie en PYTHON une chaîne `ACK` (colonne de gauche) dont on vérifie (en colonne de droite) qu'elle implémente la célèbre fonction d'Ackermann (définie en PYTHON comme la fonction `ack`).

```
ACK = """
0 ?
p
#3 (
  1-#-4 (
    -4: #-3 + 1 ?
    -2: 0 ?
    #-1 )
  1-#-3 (
    -4: #-4 - 1 ?
    -3: 1 ?
    #4 )
  #-4 ?
  #-4 - 1 ?
  1 ?
  p
  #-2 (#4)
  -7: #-4 ?
  0: -4 ?
  -4: #-4 - 1 ?
  #4 )
3: 1 ?
#1 ?
#2 ?
1 ?
p
#-2 (#4)
1: #-4 ?
0: 1 ?
"""
```

```
import test_parser
from calc import parse
test_parser.PARSER_NAME='calc'
test_parser.PARSER_UNDER_TEST=parse
from test_parser import test_result

def test2(f, code, arg1, arg2):
    test_result(str(arg1) + "?" + str(arg2) + "?" +
                code,
                [f(arg1, arg2)])

def ack(m,n):
    """Ackermann function"""
    if m <= 0:
        return n + 1
    elif n <= 0:
        return ack(m - 1, 1)
    else:
        return ack(m - 1, ack(m, n - 1))

test2(ack, ACK, 2, 7) # SUCCESS: [17] AS EXPECTED !
```

FIGURE 7 – Fonction d'Ackermann en syntaxe infixe