

Examen Python – M1

Stéphane Vialette

14 janvier 2011

1 Introduction

Nous allons formaliser une notion de machine théorique très simple, les *machines à registres*. Ces machines ont une mémoire constituée d'un nombre fini de registres $R_0, R_1, \dots, R_k$, chaque registre pouvant contenir un entier arbitraire. Une machine à registres dispose d'autre part d'un *programme* qui est une suite finie constituée de 4 types d'*instruction*, et d'un *index de lecture* qui indique la prochaine instruction du programme à exécuter. C'est donc un entier inférieur à la longueur du programme, les instructions étant numérotées de 1 en 1 à partir de 0. Les instructions sont les suivantes :

1. *incrémenter* de 1 le registre numéro i , passer à l'instruction suivante :

$$R_i := R_i + 1$$

2. *décrémenter* de 1 le registre numéro i , passer à l'instruction suivante :

$$R_i := R_i - 1$$

3. *exécuter un goto conditionnel*, si le registre numéro i est nul, aller à l'instruction numéro p , sinon passer à l'instruction suivante :

if $R_i = 0$ goto p

4. une *instruction d'arrêt* qui apparaît une et une seule fois en fin du programme

halt

On considère que la numérotation des instructions est implicite : le numéro désigne la place de l'instruction dans le programme, même si dans les exemples, on l'explicitera pour la clarté. Une telle suite d'instructions sera appelé *programme goto* dans la suite. Le calcul d'une telle machine peut ne pas terminer, et l'instruction `goto` est clairement la seule instruction susceptible de conduire le calcul à ne pas terminer.

Un exemple devrait clarifier les choses. On calcule l'addition avec la machine à 4 registres (R_0, R_1, R_2 , et R_3) dont le programme est le suivant :

```
0  if  $R_1 = 0$  goto 4
1   $R_1 := R_1 - 1$ 
2   $R_0 := R_0 + 1$ 
3  if  $R_3 = 0$  goto 0
4  if  $R_2 = 0$  goto 8
5   $R_2 := R_2 - 1$ 
6   $R_0 := R_0 + 1$ 
7  if  $R_3 = 0$  goto 4
8  halt
```

Vous pouvez suivre pas à pas ce programme et remarquez qu'il calcule (dans R_0) l'addition du contenu des registres R_1 et R_2 . Remarquez que le registre R_3 ne sert qu'à pouvoir écrire un `goto` incondi-
tionnel (il est donc nécessaire que le test $R_3 = 0$ soit toujours vrai).

2 Notre but

Notre but est d'écrire un module python `registermachine` pour simuler des machines à registres en python. Voici le code que nous pourrions par exemple utiliser pour (1) créer une machine à 4 registres, (2) créer le programme donné précédemment en exemple (addition), (3) imprimer ce programme sur la sortie standard, (4) donner des valeurs initiales à nos registres, et (5) suivre pas à pas l'exécution de ce programme (valeur des registres après chaque itération et représentation de l'index de lecture dans le programme)

```
# import everything from the registermachine module
from registermachine import *
# create a register machine with 4 registers: R0, R1, R2, and R3
machine = RegisterMachine(4)
# the program (list of instructions): addition of R1 and R2 into R0
program = Program([make_if_goto_instruction(1, 4),
                  make_dec_instruction(1),
                  make_inc_instruction(0),
                  make_if_goto_instruction(3, 0),
                  make_if_goto_instruction(2, 8),
                  make_dec_instruction(2),
                  make_inc_instruction(0),
                  make_if_goto_instruction(3, 4),
                  make_halt_instruction())])

# and print it
print 'Program:\n', program
# initialize the registers: R0 := 0, R1 := 1, R2:= 1, and R3 := 0
machine.init_registers((0, 1, 1, 0))
# execute the program on the 4-registers machine
machine.execute(program, trace=True)
```

L'exécution de ce programme produit la sortie présentée en annexe, le symbole > indique l'index de lecture dans le programme. Vous vérifierez sur la sortie qu'à la fin de son exécution, le registre R_0 contient le résultat de l'addition des registres R_1 et R_2 ... à savoir $1 + 1 = 2$.

3 Premiers éléments python

Pour implémenter notre simulateur de machine à registres, nous utiliserons dans la suite les 4 classes suivantes :

- `Program` pour représenter un programme. La classe ne propose pas de méthodes pour interpréter le programme mais uniquement des méthodes pour stocker les instructions, accéder aux instructions et afficher le programme.
- `RegisterMachine` pour représenter une machine à registre. Elle devra être initialisée pour un nombre donné de registres et permettra *autant de fois que nécessaire* d'initialiser les registres, de consulter le contenu des registres, et d'exécuter un programme.
- `RegisterMachineException` et `ProgramException` pour les exceptions relatives à la machine à registre et au programme, respectivement.

Qu'en est-il des instructions ? Pour simplifier, nous les représenterons simplement par des tuples dont le premier élément sera *toujours* un mot clef désignant l'instruction. Pour faciliter la création et la manipulation des instructions, nous supposons déjà disposer du fragment suivant (notez la présence de constantes, par exemple, `IF_GOTO_KW`, `IF_GOTO_KW_IDX`, `IF_GOTO_REG_IDX`, et `IF_GOTO_LINE_IDX` pour faciliter l'accès aux différents constituants de l'instruction `if ... goto`):

```
##
## if ... goto instruction
##
IF_GOTO_KW   = 'IF_GOTO'
```

```

IF_GOTO_KW_IDX    = 0
IF_GOTO_REG_IDX   = 1
IF_GOTO_LINE_IDX  = 2

def make_if_goto_instruction(register_idx, line_number):
    """construct a 'if goto' instruction

    format: (if_goto instruction keyword, register index, line number)"""
    return (IF_GOTO_KW, register_idx, line_number)

def is_if_goto_instruction(instruction):
    """return True iff the argument is a 'if goto' instruction"""
    return len(instruction) == 3 && instruction[IF_GOTO_KW_IDX] == IF_GOTO_KW

##
## inc instruction
##
INC_KW            = 'INC'
INC_KW_IDX        = 0
INC_REG_IDX       = 1

def make_inc_instruction(register_idx):
    """construct an 'inc' instruction

    format: (increment instruction keyword, register index)"""
    return (INC_KW, register_idx)

def is_inc_instruction(instruction):
    """return True iff the argument is an 'inc' instruction"""
    return len(instruction) == 2 && instruction[INC_KW_IDX] == INC_KW

##
## dec instruction
##
DEC_KW            = 'DEC'
DEC_KW_IDX        = 0
DEC_REG_IDX       = 1

def make_dec_instruction(register_idx):
    """construct a 'dec' instruction

    format: (decrement instruction keyword, register index)"""
    return (DEC_KW, register_idx)

def is_dec_instruction(instruction):
    """return True iff the argument is a 'dec' instruction"""
    return len(instruction) == 2 && instruction[DEC_KW_IDX] == DEC_KW

##
## halt instruction
##
HALT_KW           = 'HALT'
HALT_KW_IDX       = 0

def make_halt_instruction():
    """construct a 'halt' instruction

```

```

    format: (halt instruction keyword, )"""
    return (HALT_KW,)

def is_halt_instruction(instruction):
    """return True iff the argument is a 'halt' instruction"""
    return len(instruction) == 1 && instruction[HALT_KW_IDX] == HALT_KW

```

Question 1.

Écrire la fonction `is_instruction(instruction)` qui retourne `True` si l'argument `instruction` est une instruction valide et `False` sinon. En d'autres termes cette fonction retourne `True` si `instruction` est un tuple respectant le format des fonctions `make_if_goto_instruction`, `make_inc_instruction`, `make_dec_instruction`, et `make_halt_instruction`.

Question 2.

En prenant exemple sur l'affichage des différentes instructions donné dans l'annexe, écrire la fonction `instruction_str(instruction)` qui retourne une version textuelle de l'instruction passée en argument. Cette fonction lève l'exception `ProgramException` si l'argument n'est pas une instruction valide, c'est-à-dire si l'argument n'est pas un tuple respectant le format des fonctions `make_if_goto_instruction`, `make_inc_instruction`, `make_dec_instruction`, et `make_halt_instruction`. Par exemple,

```

>>> from registermachine import instruction_str
>>> from registermachine import make_if_goto_instruction, make_inc_instruction
>>> from registermachine import make_dec_instruction, make_halt_instruction
>>> instruction_str(make_if_goto_instruction(4, 3))
'if R[4] = 0 goto 3'
>>> instruction_str(make_inc_instruction(1))
'inc R[1]'
>>> instruction_str(make_dec_instruction(0))
'dec R[0]'
>>> instruction_str(make_halt_instruction())
'halt'
>>> instruction_str(('GOTO', 10)) # oups !!!!
ProgramException Traceback (most recent call last)
ProgramException: unknown instruction
>>> instruction_str((IF_GOTO_KW, 1, 2)) # hard to remember, but it works !
>>> 'if R[1] = 0 goto 2'

```

4 Le programme

Nous voilà maintenant en mesure de nous attaquer à la classe `Program` dont voici un extrait :

```

class Program(object):
    def __init__(self, instructions):
        """init a program with some instructions"""
        self._instructions = []
        for instruction in instructions:
            self.add_instruction(instruction)

    def __len__(self):
        """return the number of instructions in this program"""
        ...

    def to_str(self, program_index):
        """return a string representation of this program"""

```

```

...

def __getitem__(self, idx):
    """return the instruction at line idx in this program"""
    return self._instructions[idx]

def __setitem__(self, idx, instruction):
    """set an instruction in this program"""
    self._instructions[idx] = instruction

def add_instruction(self, instruction):
    """add an instruction at the end of this program"""
    ...

```

Un mot d'explication sur la méthode `to_str`. Son but est de retourner une représentation textuelle du programme (reférez vous à l'annexe pour des exemples) et vous remarquerez qu'elle prend en argument un entier `program_index` qui représente l'index de lecture courant dans le programme. Par conséquent, pour indiquer cet index de lecture dans la représentation textuelle, l'instruction en position `program_index` dans le programme, *et seulement celle-ci*, sera précédée d'un symbole distinctif >. Encore une fois reférez vous à l'annexe pour des exemples.

Question 3.

Compléter les méthodes `__len__`, `to_str`, et `add_instruction` de la classe `Program`.

5 La machine à registres ... enfin !

Nous arrivons au cœur de notre module, la classe `RegisterMachine`, dont voici un extrait :

```

class RegisterMachine(object):
    def __init__(self, nb_registers):
        """initialize the register machine for nb_register registers"""
        self._registers = [0] * nb_registers

    def _execute_if_goto_instruction(self, instruction):
        """processing a if ... goto instruction"""
        ...

    def _execute_inc_instruction(self, instruction):
        """processing an inc instruction"""
        ...

    def _execute_dec_instruction(self, instruction):
        """processing a dec instruction"""
        ...

    def execute(self, program):
        """execute a program on this register machine"""
        self._program_idx = 0
        i = 1
        while True:
            print "Register machine snapshot. Step: %d" % (i,)
            print self.registers_str()
            print program.to_str(self._program_idx)
            print
            i = i+1

```

```

        # exit infinite loop on halt instruction
        instruction = program[self._program_idx]
        if is_if_goto_instruction(instruction):
            self._execute_if_goto_instruction(instruction)
        elif is_inc_instruction(instruction):
            self._execute_inc_instruction(instruction)
        elif is_dec_instruction(instruction):
            self._execute_dec_instruction(instruction)
        elif is_halt_instruction(instruction):
            break
        else:
            RegisterMachineException("unknown instruction")

    def registers_str(self):
        """return a string representation of the registers"""
        ...

    def init_registers(self, values):
        """initialize the registers"""
        ...

```

Question 4.

Écrire les méthodes `registers_str` et `init_registers` de la classe `RegisterMachine`. Pour cela vous examinerez comment est appelée la méthode `registers_str` depuis la méthode `execute`, et comment `init_registers` a été utilisée dans notre programme d'exemple Page 2

Question 5.

Après avoir examiné attentivement le corps de la méthode `execute`, écrire les méthodes `_execute_if_goto_instruction`, `_execute_inc_instruction`, et `_execute_dec_instruction` de la classe `RegisterMachine`.

6 Annexes

```
$ python registermachine.py
```

Program:

```
> 0: if R[1] = 0 goto 4
   1: dec R[1]
   2: inc R[0]
   3: if R[3] = 0 goto 0
   4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 1

R[0] = 0, R[1] = 1, R[2] = 1, R[3] = 0

```
> 0: if R[1] = 0 goto 4
   1: dec R[1]
   2: inc R[0]
   3: if R[3] = 0 goto 0
   4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 2

R[0] = 0, R[1] = 1, R[2] = 1, R[3] = 0

```
0: if R[1] = 0 goto 4
> 1: dec R[1]
   2: inc R[0]
   3: if R[3] = 0 goto 0
   4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 3

R[0] = 0, R[1] = 0, R[2] = 1, R[3] = 0

```
0: if R[1] = 0 goto 4
1: dec R[1]
> 2: inc R[0]
   3: if R[3] = 0 goto 0
   4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 4

R[0] = 1, R[1] = 0, R[2] = 1, R[3] = 0

```
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
> 3: if R[3] = 0 goto 0
   4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
```

```
8: halt
```

Register machine snapshot. Step: 5

R[0] = 1, R[1] = 0, R[2] = 1, R[3] = 0

```
> 0: if R[1] = 0 goto 4
   1: dec R[1]
   2: inc R[0]
   3: if R[3] = 0 goto 0
   4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 6

R[0] = 1, R[1] = 0, R[2] = 1, R[3] = 0

```
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
3: if R[3] = 0 goto 0
> 4: if R[2] = 0 goto 8
   5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 7

R[0] = 1, R[1] = 0, R[2] = 1, R[3] = 0

```
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
3: if R[3] = 0 goto 0
4: if R[2] = 0 goto 8
> 5: dec R[2]
   6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 8

R[0] = 1, R[1] = 0, R[2] = 0, R[3] = 0

```
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
3: if R[3] = 0 goto 0
4: if R[2] = 0 goto 8
5: dec R[2]
> 6: inc R[0]
   7: if R[3] = 0 goto 4
   8: halt
```

Register machine snapshot. Step: 9

R[0] = 2, R[1] = 0, R[2] = 0, R[3] = 0

```
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
3: if R[3] = 0 goto 0
4: if R[2] = 0 goto 8
5: dec R[2]
```

```
6: inc R[0]
> 7: if R[3] = 0 goto 4
8: halt
```

```
Register machine snapshot. Step: 10
R[0] = 2, R[1] = 0, R[2] = 0, R[3] = 0
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
3: if R[3] = 0 goto 0
> 4: if R[2] = 0 goto 8
5: dec R[2]
6: inc R[0]
7: if R[3] = 0 goto 4
```

```
8: halt
```

```
Register machine snapshot. Step: 11
R[0] = 2, R[1] = 0, R[2] = 0, R[3] = 0
0: if R[1] = 0 goto 4
1: dec R[1]
2: inc R[0]
3: if R[3] = 0 goto 0
4: if R[2] = 0 goto 8
5: dec R[2]
6: inc R[0]
7: if R[3] = 0 goto 4
> 8: halt
```