UNIVERSITÉ DE CORSE PASCAL PAOLI

L3 Science pour l'ingénieur Informatique

Rapport du projet

Compilateur

Projet préparé sous la direction de Yannick Stara, enseignant

Présenté par :

Jean-François GIAMMARI

Baptiste VARAMO

Session 2020 - 2021

Dans ce projet, le but est de réaliser un compilateur d'un langage de type PASCAL, ce rapport expose la méthodologie utilisé pour réaliser étape par étape ce projet. Pour le réaliser nous avons décider d'utiliser le langage Python (3.7) dans lequel nous implémenteront les fonctions utiles à notre programme.

ETAPE 1 : Interpréteur de code exécutable

Dans notre système, il existe un jeu d'instruction qui permette d'effectuer une action spécifique, c'est l'interpréteur qui est chargé de lire les instructions dans un fichier, puis de les enregistrer. L'interpréteur exécute les unes après les autres les instructions présent dans celui-ci en sauvegardant ça position.

Notre système comporte donc 2 structures de données, représenter sous forme de liste :
PCODE : Liste de toutes les instructions à effectuer
MEM : Liste de stockages des variables de la pile remplis et vider par les instructions

Puis il existe une variable d'entier *PC* qui est le pointeur d'instructions, ce qui nous permet de savoir la position de l'interpréteur dans le PCODE tout au long de l'interprétation. Dans un premier temps, c'est la fonction *charge(file)* qui s'occupe de charger et de formater dans le PCODE la liste des instructions présentes dans un fichier externe, puis il lance l'interpréteur.

```
def charge(file):
    """
    Charge le fichier contenant les instructions dans le PCODE
    """
    with open('INST.txt') as f:
        for line in f:
            line = line.rstrip()
            if " " in line:
                line = line.split(" ")
                PCODE.append(line)
            else:
                PCODE.append([line])
            if 'str' in line:
                break
    interprete()
```

Le fichier externe que nous utiliserons tout au long de ce rapport est une suite d'instructions qui consiste à faire la somme des nombres entier saisis jusqu'à une saisie null (0).

Le jeu d'instruction que nous utilisons (MACH) est représenté comme ceci :

ADD	Additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	Laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	Imprime le sommet et dépile
INN	Lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile
LDI v	Empile v
LDA a	Empile a
LDV	Remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet
STO	Stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	Branchement inconditionnel à l'
BZE i	Branchement à l'instruction i si le sommet = 0, dépile
HLT	Fin du programme (Halte)

Une fois les instructions sauvegarder correctement dans le PCODE, l'interpréteur vas parcourir le PCODE, guider la variable PC, et il va exécuter la fonction associer à chaque instruction, avec ou sans paramètre de fonctions, bien sûr si il n'existe pas la fonction associer a l'instruction parcourue, le programme lève une erreur et indique la ligne défaillante :

```
def interprete():
    """ Lance les instructions à partir de leurs nom """
    global MEM
    global PC
    global PCODE
    while PC < len(PCODE):
        inst = PCODE[PC]
        try:
            if len(inst) == 1:eval(inst[0]+"()")
            else:eval(inst[0]+"("+inst[1]+")")

        except NameError:
            print("ERREUR : "+str(inst[0])+" n'existe pas (Ligne : "+str(PC)+")")
            PC = 0
        PC+=1
```

Voici notre PCODE durant l'exécution de l'interpréteur :

```
[['INT', '2'], ['LDA', '0'], ['INN'], ['LDA', '1'], ['LDA', '0'], ['LDV'],
['LDA', '1'], ['LDV'], ['ADD'], ['STO'], ['LDA', '0'], ['LDV'], ['LDI', '0'],
['EQL'], ['BZE', '1'], ['LDA', '1'], ['LDV'], ['PRN'], ['HLT']]
```

Dans le reste du programme, se trouve donc l'ensemble du jeu d'instruction présent sous forme de fonctions effectuant des actions sur PC et/ou MEM, voici un exemple des fonctions les plus représentatives des interactions :

```
def ADD():
    """Additionne le sous-sommet de pile et le sommet, laisse
    le résultat au sommet."""
    MEM.append(MEM[-1]+MEM[-2])
    MEM.pop(-2)
    MEM.pop(-2)

def EQL():
    """Laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon"""
    if MEM[-1] == MEM[-2]:
        MEM.append(1)
    else:
        MEM.append(0)
    MEM.pop(-2)
    MEM.pop(-2)

def INN():
    """Lit un entier, le stocke à l'adresse trouvée au sommet de
    pile, dépile """
    MEM[MEM[-1]] = int(input("Saisi d'un entier :"))
    MEM.pop(-1)

def INT(p):
    """Incrémente de la constante p le pointeur de pile"""
    global PC
    for p in range(p):
        MEM.append(0)
        pass

def LDI(p):
    """Empile la valeur p"""
    MEM.append(p)


def STO():
    """Stocke la valeur au sommet à l'adresse indiquée par le
    sous-sommet, dépile 2 fois """
    MEM[MEM[-2]] = MEM[-1]
    MEM.pop(-1)
    MEM.pop(-1)

def BRN(p):
    """Branchement inconditionnel à l'instruction p"""
    global PC
    PC = p

def BZE(p):
    global PC
    """Branchement à l'instruction p si le sommet = 0, dépile"""
    if MEM[-1] == 0:
        PC = p
```

Avec ce système, nous obtenons un interpréteur qui fonctionne, mais qui ne supporte pas les erreurs, on peut constater la bonne exécution du programme en lisant la trace de MEM tout au long de l'exécution des instructions PCODE :

```
PC : 0 - PCODE : ['INT', '2'] - MEM : [0, 0]
PC : 1 - PCODE : ['LDA', '0'] - MEM : [0, 0, 0]
Saisi d'un entier :5
PC : 2 - PCODE : ['INN'] - MEM : [5, 0]
PC : 3 - PCODE : ['LDA', '1'] - MEM : [5, 0, 1]
PC : 4 - PCODE : ['LDA', '0'] - MEM : [5, 0, 1, 0]
PC : 5 - PCODE : ['LDV'] - MEM : [5, 0, 1, 5]
PC : 6 - PCODE : ['LDA', '1'] - MEM : [5, 0, 1, 5, 1]
PC : 7 - PCODE : ['LDV'] - MEM : [5, 0, 1, 5, 0]
PC : 8 - PCODE : ['ADD'] - MEM : [5, 0, 1, 5]
PC : 9 - PCODE : ['STO'] - MEM : [5, 5]
PC : 10 - PCODE : ['LDA', '0'] - MEM : [5, 5, 0]
PC : 11 - PCODE : ['LDV'] - MEM : [5, 5, 5]
PC : 12 - PCODE : ['LDI', '0'] - MEM : [5, 5, 5, 0]
PC : 13 - PCODE : ['EQL'] - MEM : [5, 5, 0]
PC : 1 - PCODE : ['LDA', '0'] - MEM : [5, 5, 0]
Saisi d'un entier :0
PC : 2 - PCODE : ['INN'] - MEM : [0, 5]
PC : 3 - PCODE : ['LDA', '1'] - MEM : [0, 5, 1]
PC : 4 - PCODE : ['LDA', '0'] - MEM : [0, 5, 1, 0]
PC : 5 - PCODE : ['LDV'] - MEM : [0, 5, 1, 0]
PC : 6 - PCODE : ['LDA', '1'] - MEM : [0, 5, 1, 0, 1]
PC : 7 - PCODE : ['LDV'] - MEM : [0, 5, 1, 0, 5]
PC : 8 - PCODE : ['ADD'] - MEM : [0, 5, 1, 5]
PC : 9 - PCODE : ['STO'] - MEM : [0, 5]
PC : 10 - PCODE : ['LDA', '0'] - MEM : [0, 5, 0]
PC : 11 - PCODE : ['LDV'] - MEM : [0, 5, 0]
PC : 12 - PCODE : ['LDI', '0'] - MEM : [0, 5, 0, 0]
PC : 13 - PCODE : ['EQL'] - MEM : [0, 5, 1]
PC : 14 - PCODE : ['BZE', '1'] - MEM : [0, 5, 1]
PC : 15 - PCODE : ['LDA', '1'] - MEM : [0, 5, 1, 1]
PC : 16 - PCODE : ['LDV'] - MEM : [0, 5, 1, 5]
5
PC : 17 - PCODE : ['PRN'] - MEM : [0, 5, 1]
PC : 18 - PCODE : ['HLT'] - MEM : [0, 5, 1]
```

ETAPE 2 : Réalisation de l'interface d'entrée et de l'analyseur lexicale associé

(Suite a un manque de temps, cette partie n'a pas pu être implémenté)

Pour accéder à l'ensemble du code, vous pouvez vous rendre sur l'adresse internet suivante :

https://github.com/JF-GIAMMARI/Compilation