Concevoir son propre langage de programmation

Programmer ou être programmé

Autonomie : plutôt qu'utiliser passivement un programme (traitement de texte, application de réservation de billets de train...) concevoir soi-même un programme

Mais quand nous écrivons un programme, nous sommes utilisateurs passifs d'un langage de programmation (Scratch, Python, Java...) au lieu de le concevoir nous-mêmes

Autonomie^2

Microsoft, Google, Facebook, Amazon... ne dépendent pas de langages qu'ils utilisent passivement, mais ils conçoivent eux-mêmes leurs propres langages

Quid de nos élèves ?

Ils peuvent aussi concevoir de petits langages de programmation

Non, dans le but de les utiliser, mais de comprendre ce qu'est un langage de programmation

Programmer : lien entre algorithme et langage
Concevoir un langage : lien entre langage et machine

Éclaire deux mystères :
(1) le bouton `compile`, pourquoi compiler avant d'exécuter ?
(2) Comment une machine électronique peut-elle se débrouiller avec un programme (incarnation)

Fascination

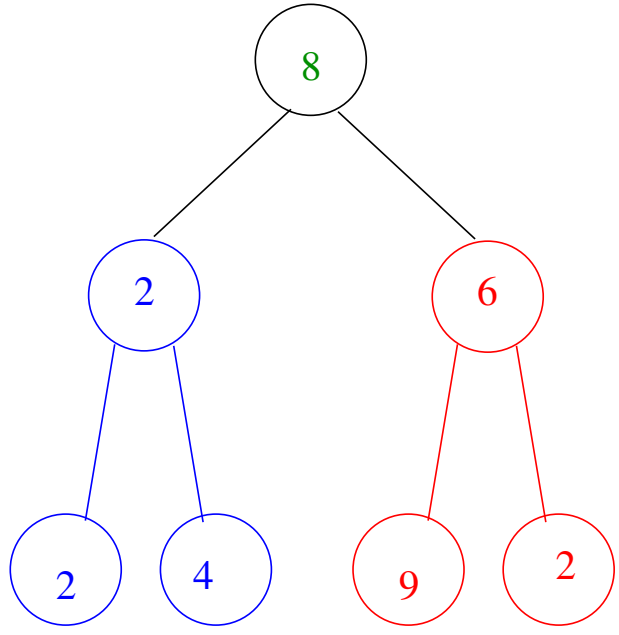
Mais aussi

- Pas d'effet Waouh (42, 42, 42...)
- Peu de lignes de code mais beaucoup de temps pour les écrire (densité)
- Il faut aimer les mots et les symboles
- Parfois difficile à débugger

I. Les programmes sont des arbres

Les arbres

Un arbre est ou bien l'arbre vide ou bien un triplet formé d'une valeur et de deux autres arbres

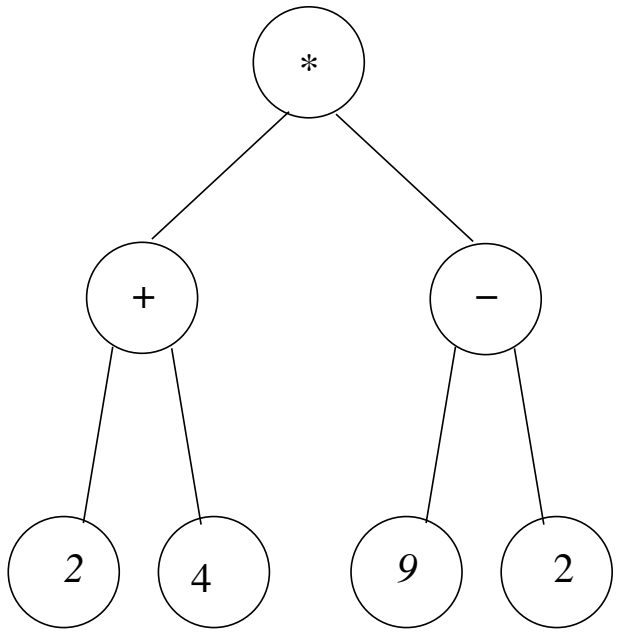



```
class node(object):
    def __init__(self, value, l, r):
        self.value = value
        self.l = l
        self.r = r
```

`None` identifié avec l'arbre vide

```
t = node(8,
         node(2,
              node(2, None, None),
              node(4, None, None)),
         node(6,
              node(9,None,None),
              node(2,None,None)))
```

Les valeurs sont des nombres ou des chaînes de caractères



Les valeurs sont des chaînes de caractères ou des nombres

```
t = node("*",
         node("+",
              node(2, None, None),
              node(4, None, None)),
         node("-",
              node(9,None,None),
              node(2,None,None)))
```

Imprimer une expression algébrique

```
def pr(a):
    if a == None:
        print("None", end = "")
    elif (a.l == None) & (a.r == None):
          print (a.value, end = "")
    else:
        print (a.value, end = "")
        print ("(",end="")
        pr(a.l)
        print (",", end = "")
        pr(a.r)
        print (")",end="")

*(+(2,4),-(9,2))
```

Et en notation infixe

```
def infix(a):
    if a == None:
        print("None", end = "")
    elif (a.l == None) & (a.r == None):
          print (a.value, end = "")
    else:
        print ("(",end="")
        infix(a.l)
        print (a.value, end = "")
        infix(a.r)
        print (")",end="")
```

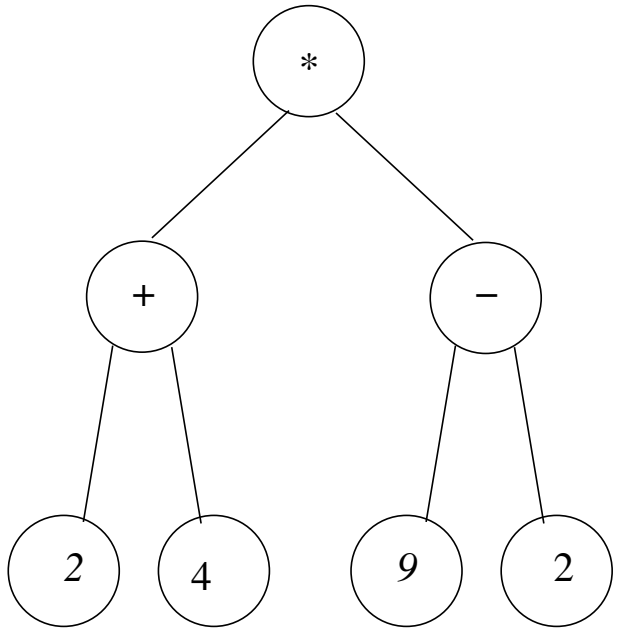
```
((2+4)*(9-2))
```

Pas très joli, mais correct et améliorable

Le point d'où partent tous les chemins

Évaluer une expression arithmétique

```
def ev(a):
    if a.value == "+":
        return ev(a.l) + ev(a.r)
    elif a.value == "-":
        return ev(a.l) - ev(a.r)
    elif a.value == "*":
        return ev(a.l) * ev(a.r)
    elif a.value == "/":
        return ev(a.l) / ev(a.r)
    else:
        return a.value
```


*
+
−
2
4
9
2

II. Une première direction : enrichir le langage

- Évaluer une expression arithmétique avec des variables

$$(x + 4) \times (y - 2)$$

  dans l'environnement $x = 2, y = 9$
- Ajouter le test
- Ajouter des fonctions

```
f2 = node("fact",
          node("x",
               node("if",
                    node("x",None,None),
                    node("",
                         node(1,None,None),
                         node("*",
                              node("x",None,None),
                              node("app",
                                   node("fact",None,None),
                                   node("-",
                                        node("x",None,None),
                                        node(1,None,None))))))),
               None),
          None)

print(
    eval4(
        node("app", node("fact",None,None), node(6,None,None)),
        None,
        f2))
```

III. Une seconde direction : compiler

Les limites de l'interprétation

```
def ev(a):
    if a.value == "+":
        return ev(a.l) + ev(a.r)
    elif a.value == "-":
        return ev(a.l) - ev(a.r)
    elif a.value == "*":
        return ev(a.l) * ev(a.r)
    elif a.value == "/":
        return ev(a.l) / ev(a.r)
    else:
        return a.value
```

```
def ev(a):
    if a.value == "+":
        return ev(a.l) + ev(a.r)
...
```

Attention le premier + est celui de votre langage (le langage interprété) alors que le second est celui du langage dans lequel vous écrivez l'interpréteur (Python)
Parfois deux symboles

Un malaise : pour évaluer les expressions de la forme `+(t,u)` on utilise le fait que Python a déjà une fonction +
Dès lors : pourquoi ne pas écrire `print ((2+4)*(9-2))` directement en Python ?

Un malaise qui ne fait que s'accroître quand on ajoute les variables, le test et les fonctions (similaires à celles de Python)

Le fond du problème

L'exercice est trop simple : un interpréteur en Python d'un sous-ensemble de Python

Deux issues :

- Ajouter de la complexité : introduire dans le langage interprété des fonctionnalités qui ne sont pas dans le langage de l'interpréteur (langage impératif dans un langage fonctionnel, langage en appel par nom, dans un langage en appel par valeur, listes infinies...)
  Finalement : trop complexe pour le lycée
- Abandonner l'interprétation pour la compilation

```
def ev2(a):
    if a.value == "+":
        n = ev2(a.l)
        p = ev2(a.r)
        return n + p
    elif a.value == "-":
        n = ev2(a.l)
        p = ev2(a.r)
        return n - p
    elif a.value == "*":
        n = ev2(a.l)
        p = ev2(a.r)
        return n * p
    elif a.value == "/":
        n = ev2(a.l)
        p = ev2(a.r)
        return n / p
    else:
        return a.value
```

Les mystères de la récursivité

```
def ev2(a):
    if a.value == "+":
        n = ev2(a.l)
        p = ev2(a.r)
        return n + p
...
```

Quand on évalue `13 + (14 + 15)`

- `n` prend la valeur 13
- lors de l'appel récursif à `ev2` sur `14 + 15`, `n` prend d'autres valeurs
- mais quand on revient de cet appel, `n` retrouve la valeur 13

Antiphysique : chaque évolution de l'univers efface son état précédent sauf si nous en gardons un modèle réduit (par exemple une photo)

Une pile

Supposons que Python ne nous garantisse pas la restauration de la valeur ancienne de `n` (comme Basic ou les langages machines), ce serait à nous d'en garder la copie

14
13
5
2

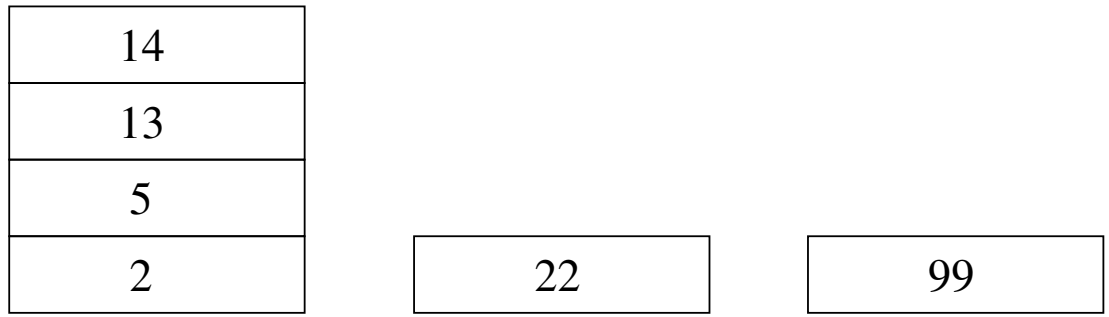
```
class cons(object):
    def __init__(self, hd, tl):
        self.hd = hd
        self.tl = tl

pile = None

def push(a):
    global pile
    pile = cons(a,pile)

def pop():
    global pile
    t = pile.hd
    pile = pile.tl
    return t
```

```
def ev3(a):
    if a.value == "+":
        n = ev3(a.l)
        push(n)
        p = ev3(a.r)
        n = pop()
        return n + p
...
```

L'appel à ev3 préserve la pile

Et le résultat est...

Se débarrasser complètement des variables locales



Se débarrasser complètement des variables locales

```
pile = None
accx = 0
accy = 0

def push2():
    global pile
    global accx
    global accy
    pile = cons(accx,pile)

def pop2():
    global pile
    global accx
    global accy
    accx = pile.hd
    pile = pile.tl
```

Se débarrasser complètement des variables locales

```
def ev4(a):
    global pile
    global accx
    global accy
    if a.value == "+":
        ev4(a.l)
        push2()
        ev4(a.r)
        accy = accx
        pop2()
        accx = accx + accy
...
```

Plus de variables, mais des commandes données à une machine et qui transforment son état : `push2()`, `pop2()`, `accx = accx + accy`, `accy = accx`...

Le copilote donne des commandes

tourne à gauche
tourne à droite

Désynchroniser, enregistrer, rejouer...

`pop2()` $\longrightarrow$ `print("Pop")`

Et l'interpréteur est devenu compilateur

```
def compil(a):
    if a.value == "+":
        compil(a.l)
        print("Push", end = " ")
        compil(a.r)
        print("Movexy", end = " ")
        print("Pop", end = " ")
        print("Plus", end = " ")
...
```

Mon premier programme compilé : (2 + 4) * (9 - 2)

```
t = node("*",
         node("+",
              node(2, None, None),
              node(4, None, None)),
         node("-",
              node(9,None,None),
              node(2,None,None)))

compil(t)
```

```
Ldx 2 Push Ldx 4 Movexy Pop Plus Push Ldx 9 Push Ldx
2 Movexy Pop Minus Movexy Pop Mult
```

Impossible à calculer à la main

Et pour finir ... une machine abstraite pour exécuter le code produit

```
def execute(code):
    pile = None
    accx = 0
    accy = 0
    while code != None:
        if code.hd.hd == "Ldx":
            accx = code.hd.tl
        elif code.hd.hd == "Push":
            pile = cons(accx,pile)
        elif code.hd.hd == "Pop":
            accx = pile.hd
            pile = pile.tl
        elif code.hd.hd == "Movexy":
            accy = accx
        elif code.hd.hd == "Plus":
            accx = accx + accy
        elif code.hd.hd == "Minus":
            accx = accx - accy
        elif code.hd.hd == "Mult":
            accx = accx * accy
        elif code.hd.hd == "Div":
            accx = accx / accy
        code = code.tl
    return accx
```

Entièrement impératif : transformations d'un objet physique

Et le résultat est...

42 une fois de plus

Much ado about nothing : 37 lignes de Python pour le compilateur, 29 pour la machine abstraite

Comment continuer ?

Traduire le code compilé vers l'assembleur de votre processeur préféré ?
Possible, mais pas si utile : machines abstraites, virtualisation, calcul dans les nuages...

Enrichir le langage (par exemple avec des variables) ?
Possible, mais pas si facile

Écrire un analyseur syntaxique

Plutôt comprendre que l'on a compris le processus de compilation : le bouton `Compile`, mais surtout le lien mystérieux qui unit les aspects linguistiques et physiques de l'informatique