Langage et compilateur

Projet – Documentation  
Module : programmation avancee  
Auteurs : Matthieu chavaillaz et Anthony geiser  
classe : inf3n  
Date : 24.01.2012  
mandataire : hatem ghorbel

Table des matières

[2 Introduction 3](#_Toc315178315)

[3 But du projet 3](#_Toc315178316)

[4 Fonctionnalitées 3](#_Toc315178317)

[4.1 Mot clés 3](#_Toc315178318)

[4.1.1 Structure d’une instruction 4](#_Toc315178319)

[4.1.2 Assignation d’une variable 4](#_Toc315178320)

[4.1.3 Opérations et calculs 4](#_Toc315178321)

[4.1.4 Comparaisons 5](#_Toc315178322)

[4.1.5 Affichage 6](#_Toc315178323)

[4.1.6 Interaction utilisateur 6](#_Toc315178324)

[4.1.7 Bloc de condition 7](#_Toc315178325)

[4.1.8 Bloc de boucle 7](#_Toc315178326)

[4.2 Gestion des erreurs 8](#_Toc315178327)

[5 Codes d’exemple 9](#_Toc315178328)

[5.1 Boucles imbriquées 9](#_Toc315178329)

[5.2 Interactions utilisateurs 10](#_Toc315178330)

[5.3 Conditions imbriquées 11](#_Toc315178331)

[5.4 Calculatrice 12](#_Toc315178332)

[6 structure du projet 13](#_Toc315178333)

[6.1 ASTCustom.py 13](#_Toc315178334)

[6.1.1 Inputnode 13](#_Toc315178335)

[6.1.2 IFELSENODE 13](#_Toc315178336)

[6.1.3 COMPARENODE 14](#_Toc315178337)

[6.2 lex.py 14](#_Toc315178338)

[6.2.1 Reserved words 14](#_Toc315178339)

[6.2.2 Tokens 14](#_Toc315178340)

[6.2.3 literalsNo 15](#_Toc315178341)

[6.3 parserv2.py 15](#_Toc315178342)

[6.4 recinterpreter.py 15](#_Toc315178343)

[6.4.1 AssignNode 15](#_Toc315178344)

[6.4.2 Comparenode 17](#_Toc315178345)

[7 Problèmes rencontrés 17](#_Toc315178346)

[7.1 Shift / reduce 17](#_Toc315178347)

[7.2 Opérations sur des variables saisies par l’utilisateur 17](#_Toc315178348)

[8 Conclusion 18](#_Toc315178349)

[9 Annexes 18](#_Toc315178350)

# Introduction

Dans le cadre du module « 3244-ProgrammationAvancee », nous avons dû réaliser notre propre compilateur en langage Python et en utilisant PLY.

Ce projet s’est déroulé par groupe de 2 personnes et le choix dans la création du type de langage et de sa syntaxe était libre.

# But du projet

Ce projet a pour but de mettre en pratique toutes les nouvelles notions acquises durant la 1ère partie de ce module à savoir le langage Python et les Compilateurs.

Nous avons choisi de définir notre propre langage « exotique » de programmation ayant des fonctionnalités et une syntaxe particulière. Nous avons également développé un interpréteur récursif pour ce langage.

Les détails de nos objectifs principaux et secondaires se trouvent dans le cahier des charges disponible en annexe.

# Fonctionnalitées

On retrouve dans notre langage quelques fonctionnalités dites « classiques » en programmation à savoir :

* Variable
* Test
* Boucle
* Affichage en mode console
* Saisie utilisateur

Le tableau ci-dessous décrit les correspondances entre notre langage et les instructions typiques en langage C.

## Mot clés

|  |  |
| --- | --- |
| **Notre langage** | **Langage C** |
| **Lexèmes** |  |
| Ipaf{condition}[programme] | If(condition){programme} |
| Epaf[programme] | Else{programme} |
| Xptdr{condition}[programme] | While(condition){programme} |
| Chocho{« texte »} ! | Printf(« texte») ; |
| Chocho{var1} ! | Printf(« %d», var1) ; |
|  |  |
| **Opérateurs** |  |
| Pequenoque | Plus petit que < |
| Majorque | Plus grand que > |
| ? | Affectation = |
| Plouche | Addition + |
| Pasplouche | Soustraction - |
| Moul | Multiplication \* |
| Modoulche | Modulo % |
| Divizione | Division / |
| = | Comparaison == |
| # | Différent de != |

### Structure d’une instruction

Chaque instruction doit finir par un « ! ». Voici quelques exemples d’instructions.

instruction **!**chocho{var1} **!**var2 ? var1 plouche 5 **!**

### Assignation d’une variable

#### Numérique

La valeur de la variable « a » est 2.

a **?** 2!

#### Variable

La valeur de la variable « b » est la valeur de la variable « a » (dans ce cas 2).

b **?** a!

### Opérations et calculs

Nous décrivons ici la syntaxe pour les calculs.

#### Addition

a = 1 + 2

a ? 1 **plouche** 2 !

#### Soustraction

a = 7 – 2

a ? 7 **paplouche** 2 !

#### Multiplication

a = 6 \* 2

a ? 6 **moul** 2 !

#### Division

a = 10 / 5 = 2

a ? 10 **divizione** 5!

#### Modulo

a = 10 % 6 = 4

a ? 10 **modoulche** 6!

### Comparaisons

Nous décrivons ici la syntaxe pour les opérations de comparaisons qui permettent de réaliser des conditions pour les fonctions « ipaf{condition} » et « xptdr{condition} ».

Toutes ces comparaisons retournent un « booléen ».

#### Egalité

a == b

a **=** b

#### Différence

a != b

a **#** b

#### Plus petit que

a < b

a **pequenoque** b

#### Plus grand que

a > b

a **mayorque** b

### Affichage

Dans ce chapitre, nous décrivons la syntaxe notre fonction d’affichage « chocho{} ! ». Cette fonction permet d’affiche le contenu d’un variable (texte ou nombre) ou un texte donné directement.

#### Variable

a ? 2!  
**chocho**{a}!

Sortie console :

2

#### Texte

**chocho**{"Bonjour"}!  
monTexte ? "Au revoir"!  
**chocho**{"monTexte"}!

Sortie console :

Bonjour  
Au revoir

### Interaction utilisateur

L’utilisateur à la possibilité de saisir une variable et en même temps d’afficher un texte avec l’instruction « maquestoudi{} ».

#### Saisie utilisateur

age ? **maquestoudi**{« Saisir votre age »} !  
prenom ? **maquestoudi**{« Saisir votre prenom »} !  
chocho{age} !  
chocho{nom} !

Sortie console :

Saisir votre age : 25  
Saisir votre prenom : Matthieu  
25  
Matthieu

### Bloc de condition

Correspond à un « if »

ipaf { **condition** }  
[  
 **instructions exécutées si condition vrai**  
]

Correspond à un « if – else »

ipaf { **condition** }  
[  
 **instructions exécutées si condition vrai**  
]  
epaf  
[  
 **instructions exécutées si condition faux**  
]

Exemple de code :

var1 ? maquestoudi{« Saisir un nombre »} !  
ipaf { **var1 # 12** }  
[  
 **chocho{« Différent de 12 »} !**  
]  
epaf  
[  
 **chocho{« Egal à 12 »} !**  
]

### Bloc de boucle

Correspond à un « while »

xptdr { **condition** }  
[  
 **instructions exécutées tant que la condition est fausse**  
]

Exemple de code :

var1 ? maquestoudi{« Saisir un nombre »} !  
xptdr { **var1 pequenoque 10** }  
[  
 **chocho{var1} !  
 var1 ? var1 plouche 1 !**  
]

## Gestion des erreurs

C’est au moment de créer les codes d’exemples que nous avons remarqué qu’il y avait des problèmes dans nos calculs et nos opérations de comparaisons.

Le 1er problème que nous avons découvert était au niveau des opérations (addition, soustraction, multiplication, division et modulo). Notre interpréteur permettait des opérations entre un « float » et un « string » ce qui ne doit pas être possible

Le 2ème problème était au niveau des comparaisons. Notre n’empêchions pas comparaison entre un « float » et un « string ».

Nous avons donc ajouté des tests afin de vérifier la cohérence entre les types de données. Si une erreur se produit, le programme affiche un message d’erreur explicite et quitte l’application.

*Exemple d’une addition entre une variable de type « float » et une variable de type « string ».*

var1 ? 2.5 !  
var2 ? "texte" !  
resultat ? var1 plouche var2 !

*Sortie console :*

Opération impossible entre le type « type1 » et le type « type2 ».

*Exemple d’une comparaison entre un type « float » et une variable de type « string ».*

var1 ? 7.7 !  
var2 ? "texte" !  
ipaf{var1 # var2}  
[  
 chocho{var1} !  
]

*Sortie console :*

Comparaison impossible entre le type « type1 » et le type « type2 ».

# Codes d’exemple

## Boucles imbriquées

chocho{"Code d'exemple"}!  
chocho{"Titre: Boucles imbriquées - v 1.0"}!  
chocho{"Description: Parcours d'indices i, j d'une matrice (4x2)"}!  
chocho{"-----------------"}!

i ? 1!

j ? 1!

xptdr{i pequenoque 5}  
[

chocho{"Index ligne i:"}!

chocho{i}!

xptdr{j pequenoque 3}

[

chocho{"Index colonne j:"}!

chocho{j}!

j ? j plouche 1!

]  
  
 j ? 1!

i ? i plouche 1!

]

## Interactions utilisateurs

chocho{"Code d'exemple"}!

chocho{"Titre: Interactions utilisateurs - v 1.0"}!

chocho{"Description: demande du prénom et de l'âge de l'utilisateur"}!

chocho{"-----------------"}!

loop ? 1!

xptdr{loop = 1}

[

prenom ? maquestoudi{"Entrez votre prénom"}!

age ? maquestoudi{"Entrez votre âge"}!

chocho{""}!

chocho{prenom}!

ipaf{age mayorque 17}

[

chocho{"Vous êtes majeur!"}!

]

epaf

[

chocho{"Vous êtes mineur!"}!

]

loop ? maquestoudi{"Voulez-vous recommencer? (1=oui, 0=non)"}!

]

## Conditions imbriquées

chocho{"Code d'exemple"}!

chocho{"Titre: Conditions imbriquées - v 1.0"}!

chocho{"Description: "}!

chocho{"-----------------"}!

min ? 0!

max ? 100!

chocho{"Plage de nombres autorisés:"}!

chocho{"Max="}!

chocho{max}!

chocho{"Min="}!

chocho{min}!

in ? maquestoudi{"Entrez un nombre entre 1 et 100"}!

ipaf{in mayorque 100}

[

chocho{"Vous avez saisi un nombre trop grand!"}!

]

epaf

[

ipaf{in pequenoque 0}

[

chocho{"Vous avez saisi un nombre trop petit!"}!

]

epaf

[

chocho{"Vous avez choisi un nombre dans la plage de nombres autorisés"}!

]

]

## Calculatrice

chocho{"Code d'exemple"}!

chocho{"Titre: Calcul - v 1.0"}!

chocho{"Description: Calculatrice entre deux nombres et une opération"}!

chocho{"-----------------"}!

loop ? 1!

xptdr{loop=1}

[

nombre1 ? maquestoudi{"Entrez le premier nombre"}!

operation ? maquestoudi{"Sélectionnez une opération (addition=1,  
soustraction=2, multiplication=3, division=4, modulo=5)"}!

nombre2 ? maquestoudi{"Entrez le deuxième nombre"}!

ok ? 1!

ipaf{operation=1}

[

res ? nombre1 plouche nombre2!  
 ]

epaf

[

ipaf{operation=2}

[

res ? nombre1 paplouche nombre2!

]

epaf

[

ipaf{operation=3}

[

res ? nombre1 moul nombre2!

]

epaf

[

ipaf{operation=4}

[

ipaf{nombre2=0}

[

chocho{"Division par zéro!"}!

]

epaf

[

res ? nombre1 divizione nombre2!

]

]

epaf

[

ipaf{operation=5}

[

res ? nombre1 modoulche nombre2!

]

epaf

[

chocho{"Opération impossible"}!

ok ? 0!

]

]

]

]

]

ipaf{ok=1}

[

chocho{"Résultat:"}!

chocho{res}!

]

loop ? maquestoudi{"Voulez-vous quittez? (oui=0, non=1)"}!

]

# structure du projet

## ASTCustom.py

Le fichier ASTCUSTOM.PY défini chaque type de nœud pour la construction de l’arbre syntaxique. Nous avons repris le fichier AST.PY du TP4 et l’avons compléter afin de répondre à nos besoins. Les types de nœuds ajoutés sont les suivants :

### Inputnode

class InputNode(Node):  
 type = 'Input'

Ce node défini le type Input qui permet de faire une saisie utilisateur.

### IFELSENODE

class IfElseNode(Node):  
 type = 'IpafEpaf'

Ce node défini les blocs de test conditionnels.

### COMPARENODE

class CompareNode(Node):  
 def \_\_init\_\_(self, op, children):  
 Node.\_\_init\_\_(self, children)  
 self.op = op  
 try:  
 self.nbargs = len(children)  
 except:  
 self.nbargs = 1  
 def \_\_repr\_\_(self):  
 return "%s (%s)" % (self.op, self.nbargs)

Nous nous sommes basé sur le code du « OpNode » qui est, dans notre cas, le même.

## lex.py

Ce fichier contient la définition des mots clés réservés (reserved words), les caractères(literals) et les symboles(tokens) de notre langage.

### Reserved words

reserved\_words = (  
 'ipaf',  
 'epaf',  
 'chocho',  
 'xptdr',  
 'pequenoque',  
 'mayorque',  
 'maquestoudi',  
 'maquestouveu'  
)

### Tokens

tokens = (  
 'NUMBER',  
 'ADD\_OP',  
 'MUL\_OP',  
 'IDENTIFIER',  
 'TEXTE'  
) + tuple(map(lambda s:s.upper(), reserved\_words))

### literalsNo

literals = '{}[]?!#='

## parserv2.py

Nous sommes reparti du fichier « parser.py » du TP4 et l’avons modifié afin qu’il réponde à nos besoins. Nous avons tout d’abord une 1ère version qui avait des conflits « Shift/Reduce ». C’est pourquoi nous avons refait toute la grammaire de façon claire et non-ambigüe.

Nous avons également complété le dictionnaire des opérations et y avons ajouté les comparaisons et le modulo.

operations = {  
 'plouche' : lambda x, y: x + y,  
 'paplouche' : lambda x, y: x - y,  
 'moul' : lambda x, y: x \* y,  
 'divizione' : lambda x, y: x / y,  
 **'modoulche' : lambda x, y: x % y,  
 'pequenoque': lambda x, y: x < y,  
 'mayorque' : lambda x, y: x > y,  
 '=' : lambda x, y: x == y,  
 '#' : lambda x, y: x != y**}

## recinterpreter.py

Nous sommes repartis du fichier « recInterpreter.py » du TP4 et l’avons modifié pour nos besoins.

Nous avons ajouté les « IfElseNode », « CompareNode » et « PrinNode » ainsi qu’améliorer les autres « Nodes » pour notamment la gestion des erreurs.

### AssignNode

Le « InputNode » est géré dans le « AssignNode » puisque la syntaxe est la même mais qu’il y a un nombre différents d’enfants. Si le nombre d’enfant est de 3, cela qui indique une saisie utilisateur sinon, c’est une assignation simple.

@addToClass(AST.AssignNode)  
def execute(self):  
 if(len(self.children) == 3):  
 #Type input  
 vars[self.children[0].tok] = input(str(self.children[1])[2:len(str(self.children[1]))-3]+": ")  
 else:  
 #Type assignation simple  
 try:  
 vars[self.children[0].tok] = self.children[1].execute()  
 except:  
 vars[self.children[0].tok] = self.children[1][1:len(str(self.children[1]))-1]

#### Arbre dans le cas d’un « InputNode »

?  
-- ‘var1’  
-- ‘ Texte’  
-- ‘Input’

? est le caractère qui indique l’opération d’assignation  
Le 1er fils ‘var1’ est le nom de la variable  
Le 2ème fils ‘Texte’ est la chaine de caractère affichée à l’utilisateur  
Le 3ème fils ‘Input’ nous permet de faire la différence avec un « AssignNode »

#### Arbre dans le cas d’une « AssignNode »

?  
-- ‘var1’  
-- ‘ Texte ou nombre’

? est le caractère qui indique l’opération d’assignation  
Le 1er fils ‘var1’ est le nom de la variable  
Le 2ème fils ‘Texte ou nombre’ est la valeur a assigner à la variable

#### Mise en forme

str(self.children[1])[2:len(str(self.children[1]))-3]+": "

Cette ligne permet de faire du « slicing » afin d’avoir un affichage sans les guillemets ajoutés automatiquement.

Sans la mise en forme :

‘"Saisir votre age :"’

Avec la mise en forme :

Saisir votre age :

Remarque :  
Nous avons utilisé cette méthode de mise en forme pour chaque affichage fait dans la console.

### Comparenode

Nous nous sommes basé sur le code de « OpNode » et l’avons adapté à nos besoins. Nous avons décidé de traiter uniquement la comparaison entre des nombres.

Nous avons également ajouté une détection d’erreur dans le cas où une comparaison autre que le type « float ».

@addToClass(AST.CompareNode)  
def execute(self):  
 args = [c.execute() for c in self.children]  
 if len(args) == 1:  
 args.insert(0, 0)  
 try:  
 argsFloat = []  
 for e in args:  
 argsFloat.append(float(e))  
 return reduce(operations[self.op], argsFloat)  
 except ValueError:  
 print("Comparaison impossible entre le type " + str(type(args[0])) + " et le type " + str(type(args[1])))  
 os.\_exit(0)

# Problèmes rencontrés

## Shift / reduce

Le problème principal que nous avons rencontré est le conflit shift/reduce. Pour résoudre ce conflit, nous avons dû refaire toute la grammaire de notre langage afin qu’il n’y ait plus d’ambiguïtés.

## Opérations sur des variables saisies par l’utilisateur

Les calculs effectués sur des variables saisies par l’utilisateur (**input**, mot clé : maquestoudi) plantaient à cause de problème de type. En effet, ces variables étaient automatiquement de type **string** et les opérations mathématiques ne pouvaient être réalisées.

Nous avons donc dû ajouter une gestion d’erreur permettant de convertir les saisies de l’utilisateur en nombre (**float**) pour pouvoir effectuer les différentes opérations mathématiques. S’il est impossible de convertir les saisies, les variables restent de type **string**.

Si l’utilisateur essaie de réaliser des opérations mathématiques sur des types string, notre interpréteur le détecte et le signale à l’utilisateur avec ce message avant de quitter le programme :

Operation impossible entre le type <class 'str'> et le type <class 'str'>

Le même principe de gestion d’erreur a été implémenté pour la **comparaison** de type (==, <, >, !=).

# Conclusion

Ce projet nous a permis de mieux comprendre comment fonctionne un compilateur/interpréteur. C’est en effet important pour un développeur de bien déchiffrer les messages que les compilateurs lui transmettent afin de produire un code optimal et sans erreurs.

Un des points très intéressant du projet était de développer notre propre langage. Comme nous étions totalement libre sur la structure et les mots-clés, nous avons pris plaisir un développer un langage « exotique » qui sort de l’ordinaire.

Nous avons réussi à développer un interpréteur récursif qui est fonctionnel et testé avec les différents codes d’exemples.

Grâce à ce projet, nous sommes maintenant conscients du travail et de la complexité de développer un langage de haut niveau qui contient beaucoup de fonctionnalités et une bonne gestion des erreurs.

# Annexes

* Cahier des charges.docx
* Répertoire code source
  + ASTCustom.py
  + LexCustom.py
  + ParserCustom.py
  + RecInterpreterCustom.py
* Codes d’exemples
  + boucles\_imbriquees.txt
  + calculatrice.txt
  + conditions\_imbriquees.txt
  + interactions\_utilisateurs.txt