Compilation
Fiche de Révisions:
Mini-Py-Machine

5 avril 2019

Attention : La machine décrite ici est compatible avec Python au sens où lorsque qu'elle rend un résultat et que python rend un résultat (pas d'erreur), les deux rendront la même chose.

En particulier, on ne demande pas de faire de phase de détermination de scope des variables dans les fonctions, ce qui est une phase statique de l'analyse sémantique. [1]

1 La Mini-Py-machine pour le Package 1

1.1 La sémantique

1.1.1 Types de données.

1.1.2 Le "typage" des cases mémoires.

Dans une case mémoire de la pile on peut trouver plusieurs types de structures, mais on ne peut pas vérifier qu'il s'agit du type attendu, on fait entièrement confiance au programmeur.

Voici les types :

Une instruction. Noté `(i)`. Il s'agit d'une des instructions décrites au chapitre idoine et qui déclenche une action spécifique de la machine.

Une string. Noté `(s)`. Il s'agit d'une chaîne de caractères utilisée pour stocker les noms des fonctions, variables, classes...

Une valeur. Noté `(v)`. Une valeur `x:(e)` contient un nom `x.nom : s`.

Un entier. Noté `(e)`. Un entier `x:(e)` est une valeur pour laquelle `x.nom = "Entier"`, et qui contient un Int `x.int`.

1. Si vous voulez l'ajouter elle peut rapporter un bonus.

Un réel. Noté `(r)`. Idem pour les flottants (bonus).

Un tuple. Noté `(u)`. Un tuple `x:(u)` est une valeur pour laquelle `x.nom = Tuple`, et qui contient une taille `x.taille : (i)` et un vecteur `x.tab : vec`.

Un vecteur. Noté `(vec)`. Un vecteur `x:(u)` est un tableau en mémoire, avec `x.acces(5)` on accède à la 5ème case si elle existe.

Un contexte. Noté `(c)`. Un contexte est un dictionnaire (aussi appelé Map, ou table d'association selon les implémentations) qui associe des valeurs à des noms (les clés du dictionnaire).

1.1.3 Les zones mémoires.

La bande de code. Le `CODE` est une bande de calcul en lecture seule. Il s'agit du programme entré traduit en une suite d'instructions (voire la section suivante pour le détails des instructions). Ne contient que des instructions `(i)` et des objets `(o)`.

La pile. La `STACK` est une bande de calcul de taille virtuellement infinie et manipulée comme une pile/fifo/liste. Chaque case contient une valeur c'est à dire un entier `(e)`, un flottant `(r)` ou un tuple `(u)`.

Les contextes. Les contextes sont stockés dans une zone appelée `CONT`. On les abstrait comme des dictionnaires, mais dans l'implémentation de référence, ce sont des piles de couples (nom,valeur).

Le reste de la mémoire. Cette machine est abstraite, on peut stocker des structures à des adresses diverses (correspond à un appel système). On appelle `RAM` le reste de la mémoire. [2]

Les pointeurs. Il y a plusieurs pointeurs (adresses) spéciaux que l'on manipule en permanance :

- `SP:a` pour *Stack Pointer* : pointe sur une adresse de `STACK` correspondant au sommet de pile.
- `PC:*i` pour *Code Pointer* (? ?) : pointe sur une instruction `(i)` dans `CODE` correspondant a l'instruction réalisée.
- `CC:*c` pour *Curent Context* : pointe sur le contexte courant `(c)`. dans `CONT`.

2. même si techniquement tous les autres sont aussi dans la ram...

1.1.4 Exécution.

La machine lit une instruction sur `CODE[PC]` et l'exécute, ce qui peut changer `SP`, `PC`, `CC`, `STACK`, `CONT` et `RAM`, mais pas `CODE`. Une fois l'instruction effectuée, elle continue jusquà ce que `PC` arrive à la dernière instruction de `CODE` (instruction `HALT`).

Si à n'importe quel moment `PC` ou `SP` sort de sa zone allouée, le comportement n'est plus défini ; mais ca ne devrait pas pouvoir arriver.

1.2 Les instructions

1.2.1 Syntaxe pseudo-code

Manipulation de la pile On utilise les fonctions suivantes pour exprimer des manipulations de la pile :

- `Push(x):= (STACK[++SP]:=x)` pour pousser une valeur sur la pile.
- `Pop := return STACK[SP--]` pour récupérer la tête de la pile et décrémenter cette dernière.
- `Pull(x) := return STACK[SP]` pour récupérer la tête de la pile sans décrémenter cette dernière.

Manipulation du contexte Le contexte est un dictionnaire (aussi dit Map, ou table d'association), avec les opérations suivantes :

- `Exist(n)` rend vrai si le nom `n` est associé dans le contexte.
- `Modif(n,v)` modifie la valeur associée à `n` dans le contexte. Le comportement n'est pas défini si le nom n'est pas trouvé.
- `Get(n)` récupère la valeur associée à `n` dans le contexte. Le comportement n'est pas défini si le nom n'est pas trouvé.
- `Insert(n,v)` associe au nom `n` la valeur `v` dans le contexte, si le nom est déjà référencé, la nouvelle valeur cache l'ancienne, mais ne modifie pas l'ancienne valeur, au sens où tout pointeur sur cette ancienne valeur ne verra aucune modification.

Opérations On suppose toutes les opérations habituelles d'arithmétique ou de comparaison sur les entiers et les flottants, ainsi que l'arithmétique booléenne et le cast des entiers vers les flottants.

Sens d'évaluation Le psedo-code se lit de la droite vers la gauche, par exemple, si j'écrit `Push (Pop -_e Pop)`, celà correspond au Pop du membre droit, puis celui de membre gauche puis la négation puis le Push. Ne vous inquietez pas, il s'agit de l'ordre naturel puisque l'on travail sur une pile, dans l'exemple ci-dessus, on a bien empilé le membre de gauche avant le membre de droit.

1.2.2 Arithmétique

Instruction	sémantique	pile avant	pile après
AddE	Push(Pop $+_e$ Pop);	e:e:pile	e:pile
SubE	Push(Pop $-_e$ Pop);	e:e:pile	e:pile
MulE	Push(Pop $*_e$ Pop);	e:e:pile	e:pile
DivE	Push(Pop$/_e$Pop);	e:e:pile	e:pile
NegE	Push(Pop $*_e$ (-1));	e:pile	e:pile
ModE	Push(Pop$\%_e$Pop);	e:e:pile	e:pile
CastER	Push(IntToFloat(Pop));	e:pile	r:pile
Floor	Push(Floor(Pop));	r:pile	e:pile
AddR	Push(Pop $+_r$ Pop);	r:r:pile	r:pile
SubR	Push(Pop $-_r$ Pop);	r:r:pile	r:pile
MulR	Push(Pop $*_r$ Pop);	r:r:pile	r:pile
DivR	Push(Pop$/_r$Pop);	r:r:pile	r:pile
NegR	Push(Pop $*_r$ (-1));	r:pile	r:pile
And	Push(Pop && Pop);	e:e:pile	e:pile
Or	Push(Pop $\|\|$ Pop);	e:e:pile	e:pile
Not	Push(not(Pop));	e:pile	e:pile

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :

```
PC := PC+1
```

Notez que `Floor` projette un flottant sur le plus grand entier inférieur.

1.2.3 Comparaison

Instruction	sémantique	pile avant	pile après
EqE	Push(Pop $==_e$ Pop);	e:e:pile	e:pile
NeE	Push(Pop $\neq_e$ Pop);	e:e:pile	e:pile
LeE	Push(Pop $\leq_e$ Pop);	e:e:pile	e:pile
GeE	Push(Pop $\geq_e$ Pop);	e:e:pile	e:pile
LsE	Push(Pop $<_e$ Pop);	e:e:pile	e:pile
GsE	Push(Pop $>_e$ Pop);	e:e:pile	e:pile
EqR	Push(Pop $==_r$ Pop);	r:r:pile	e:pile
NeR	Push(Pop $\neq_r$ Pop);	r:r:pile	e:pile
LeR	Push(Pop $\leq_r$ Pop);	r:r:pile	e:pile
GeR	Push(Pop $\geq_r$ Pop);	r:r:pile	e:pile
LsR	Push(Pop $<_r$ Pop);	r:r:pile	e:pile
GsR	Push(Pop $>_r$ Pop);	r:r:pile	e:pile

Toutes ces opérations sont suivies d'une incrémentation du compteur de code :
`PC := PC+1`

1.2.4 Tuples

Instruction	sémantique	pile avant	pile après
NewTuple	`Push(NewTuple{taille = Pop,` `tab = newTab(taille)});`	`e:pile`	`u:pile`
Taille	`Push(Pop.taille);`	`u:pile`	`e:pile`
GetU	`Push(Pop.tab[Pop]);`	`e:u:pile`	`X:pile`
SetU	`Pull.tab[Pop] := Pop;`	`X:e:u:pile`	`u:pile`

AddU	`tuple1 = Pop`
	`tuple2 = Pop`
	`n = tuple1.taille + tuple2.taille`
	`res = NewTuple{taille = n,`
	`tab = newTab(taille)};`
	`CopyTab(tuple1.tab, res.tab)`
	`CopyTab(tuple2.tab, res.tab + tuple1.taille)`
	`Push(res)`
		`u:u:pile`	`u:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code : `PC := PC+1`

Remarquez que l'on ne précise pas où est stocké le tableau ou même le tuple : tout est dans la mémoire, on ne maîtrise rien.

1.2.5 Jumps

Instruction	sémantique	pile avant	pile après
Jump offset	`PC := PC + off + 1;`	`pile`	`pile`
Cjmp offset	if `Pop` then `PC := PC + 1;` else `PC := PC + off + 1;`	`e:pile`	`pile`
Case gap	`PC := PC + 1 + gap * Pop;`	`e:pile`	`pile`

Ici, offset et gap sont des entiers écrits directement dans le code assembleur. Le premier correspond à un offset positif ou négatif indiquant une position dans le code relative à PC, le second correspond à la taille du saut entre deux instructions (entre deux cas du case).

Le case n'est pas nécessaire pour le projet, c'est un exemple de commande qui est pratique pour des optimisations.

1.2.6 Gestion de la pile

Instruction	sémantique	pile avant	pile après
CstE x	`Push(x);`	`pile`	`e:pile`
Copy	`x := Pull;` `Push(x);`	`X:pile`	`X:X:pile`

Swap	`x := Pop;`
	`y := Pop;`
	`Push(x);`
	`Push(y);`
		`X:Y:pile`	`Y:X:pile`

Drop	`Pop;`	`X:pile`	`pile`
Noop	ne fait rien	`pile`	`bpile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code : `PC := PC+1`

Ici, `(X)` et `(Y)` désignent des types quelconques (polymorphisme).

Le `x` de **CstE x** est un entier donné directement dans le code assembleur. On ne met pas de restriction sur sa taille.

1.2.7 Gestion des variables

Instruction	sémantique	pile avant	pile après
Modifier n	`Modif(n, Pop);`	`X:pile`	`pile`
Ajouter n	`Inserer(n, Pop);`	`X:pile`	`pile`
Modi n	if `Existe(n)` then `Modif(n, Pop);` else `Inserer(n, Pop);`	`X:pile`	`pile`
Get n	`Push(Get(n))`	`pile`	`X:pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code : `PC := PC+1`

Les `n` sont des chaînes de caractères données directement dans le code assembleur et faisant référence à des noms de variables, méthodes, ou classe. On ne met pas de restriction sur leurs tailles mais elles ne peuvent pas contenir d'espace.

Ici, **Modi n** est une combinaison de **Modifier n** et **Ajouter n**, faisant l'un ou l'autre selon que la variable soit connue ou non à ce point du programme, ce qui est le comportement par défaut de Python. On garde les deux autres pour pouvoir faire des optimisations.

1.2.8 Securité et appels système

Instruction	sémantique	pile avant	pile après
Halt	arrête la machine	`pile`	
TypeVerif n	`Push(Pop.nom ==`$_s$ `n)`	`o:pile`	`e:pile`
TypeCase	Pop une valeur, rend 0 sur un entier, 1 sur un flottant, 2 sur un tuple, 3 sur un tableau, 4 sur une cloture, 5 sur un objet	`o:pile`	`e:pile`
Error s	renvoit l'erreur s	`pile`	
Assert s	renvoit l'erreur s si Pop est faux	`e:pile`	`pile`
Print	affiche `Pop` sur le terminal	`X:pile`	`pile`

Toutes ces opérations sont suivies d'une incrémentation du compteur de code (lorsque la machine n'est pas arrêtée) :
`PC := PC+1`

Seuls **TypeVerif n** et **Halt** (fin du programme) sont nécessaires pour le projet. Les autres sont importants pour le débugage.

1.3 Détails d'implémentation

Implémentation possible du contexte Il y a plusieurs implémentations pour cette sémantique de contexte. Celà ne change pas la machine, mais peut influencer les choix d'optimisation. Dans notre cas on utilise une liste chaînée dont les noeuds contiennent des couples (nom,valeur), les fonctions définies précédemment sont alors :

- `Exist(n) = Exists_rel(n,CC)` avec
  `Exist_rel(n,c) = (c != Null) && (c.nom == n || Exist_rel(n,c.cont))`
- `Modif(n,v) = Modif_rel(n,v,CC)` avec
  `Modif_rel(n,v,c) = if (c.nom == n) then (c.val := v) else Modif_rel(n,v,c.cont)`
  On crash si le nom n'est pas trouvé.
- `Get(n,v) = Get_rel(n,v,CC)` avec
  `Get_rel(n,c) = if (c.nom == n) then (return c.val) else Get_rel(n,c.cont)`
  On crash si le nom n'est pas trouvé.
- `Insert(n,v)` est défini comme la procédure ;
  `CC := NewContext{nom = n; val = v; cont = CC}`

1.4 Exemple de compilation

Vous êtes libre de compiler un programme comme vous voulez tant que sa sémantique (c'est à dire les couples entrées/sorties) est le même. Néanmoins, pour plus de clarté, nous fournissons ici des exemples de compilation de petits programmes.

`Si x : y := 3 Sinon toto := z` peut se compiler en :

```
Get x
CJmp 3
Cst 3
Modi y
Jump 2
Get z
Modi Toto
```

Par contre, dès que l'opération dépend du type dynamique, il faut tester les types. Pour ça, il y a deux moyens : soit utiliser `TypeVerif` et faire des ifs imbriqués suivis d'un assert pour tous les cas impossibles, soit faire un case avec `TypeCase`. À vous de voir ce que vous pensez être le mieux pour chaque opération.

Ici, on donne l'exemple de "`TantQue x <= y : x := x + 1`" qui peut se compiler comme ceci :

```
Get x
Get y
Copy
TypeVerif Entier
CJmp 13
Swap  # cas y entier
Copy
TypeVerif Entier
CJmp 2
GeE  # cas y et x entiers
Jump 21  # jump sur le corps du while
Copy
TypeVerif Reel
Assert  # échoue si y entier et x non
Swap  # y entier, x réel
Cast
LeR
Jump 14  # jump sur le corps du while
Copy  # cas y non-entier
TypeVerif Reel
Assert  # erreur si y non-réel
Swap  # cas y réel
Copy
TypeVerif Entier
CJmp 3
Cast  # cas y réel, y entier
GeR
Jump 4  # jump sur le corps du while
Copy
TypeVerif Reel
Assert  # echoue sur y réel et x non
```

```
LeR  # cas y et x réels
CJmp 60  # sortie du while si faux
Get x  # Corps du while
Cst 1
Copy
TypeCase
Case 13   # Case sur le type de 1
Swap  # cas 1 entier
Copy
TypeVerif Entier
CJmp 2
AddE   # cas 1 et x entiers
Jump 47  # jump sur la fin du +
Copy
TypeVerif Reel
Assert  # échoue si x non-réel
Swap  # cas 1 entiers et x reel
Cast
AddR
Jump 40  # jump sur la fin du +
Swap  # cas 1 réel
Copy
TypeVerif Reel
CJmp 2
AddE  # cas 1 et x réels
Jump 34  # jump sur la fin du +
Copy
TypeVerif Entier
Assert  # échoue si x non-entier
Swap  # cas 1 réel et x entiers
Cast
AddR
Jump 27  # jump sur la fin du +
Swap  # cas 1 tuple
Copy
TypeVerif Tuple
Assert  # échoue si x non tuple
Swap
AddU
Jump 20  # jump sur la fin du +
Noop  # Permet l'alignement du Case
Noop
Noop
Noop
Noop
Noop
```

```
Swap  # cas 1 tuple
Copy
TypeVerif Liste  #Packages 2 et 3
Assert
Swap
AddD
Jump 7  # jump sur la fin du +
Noop
Noop
Noop
Noop
Noop
Noop
Error
Modi x  # assignement de x (enfin)
Jump -93  # retour au test du while
```

Remarquez que l'on teste même le type de 1 qui est un entier. Ce n'est pas nécessaire, mais en absence d'optimisation de votre compilateur, c'est bien le resultat attendu, car après `Cst 1` et `Modi x`, il s'agit simplement de la compilation de l'opérateur `+`, qui ignore le type de ses argument. Si vous êtes motivés, vous pouvez essayer d'optimiser ce genre de cas.

1.5 Différences et points communs avec Python

Égalite de tuples À la différence de Python, on ne demande pas de gérer les (in)égalités de tuples.

Tests sur des non-entiers En réalité, on peut utiliser n'importe quel type comme un Booléen (càd. comme condition d'un if et d'un while). Les seuls valeurs traitées comme fausses sont l'entier 0 et le réel 0 (ou plutôt les réels 0 car il y a les flottants 0^+ et 0^-).

2 La Mini-Py-machine pour le Package 2

Tout ce qui est donné dans le package 1 reste vrai pour le package 2, sauf si précisé autrement (principalement les contextes).

2.1 La sémantique

2.1.1 Types de données.

Une addresse. Noté `(a)`. Une adresse est un pointeur vers une zonne mémoire. Pour plus de lisibilité, on notera `(*x)` pour le type d'une adresse pointant sur le type `x`. Une adresse peut aussi être nulle.

Une pile de nom. Noté `(pile)`. On va avoir à manipuler une pile contenant des strings.

Un tableau dynamique. Noté `(d)`. C'est un type de donnée `x : d`, de nom `Liste` contenant une taille `x.taille : e`, une extension `x.extension : e` et un tableau `x.tab : vec` de taille `e`.

Une méthode. Noté `(m)`. C'est un sous-type de `(*i)`, au sens où une "méthode" est l'adresse d'une instruction particulière pointant le début d'un bloque de code corespondant à une méthode, voir sections 3.1.2.

Une cloture. Noté `(l)`. C'est un type de donnée `x : l` fonctionnel, défini par un contexte `x.cont : c`, le code d'une méthode `x.methode : m` ainsi qu'une pile avec des noms d'arguments `x.narg : pile`. Remarquez que l'on a alors un contexte référencés dans une valeur, on ne travail donc plus avec un seul contexte.

Un contexte. Un contexte comporte deux parties : le contexte local, accessible en lecture et écriture, et le contexte global, accessible en lecture seul. Atention, ici il est important de comprendre qu'un contexte ne contient pas une valeur, mais un pointeur sur une valeur (ceci car toute valeur se comporte comme un objet, et donc comme un pointeur, en Python).

2.1.2 Le zones mémoires.

La pile. La pile peut maintenant aussi contenir une adresse, une cloture ou un contexte, en plus des entiers `(e)`, un flotant `(r)` ou un tuple `(u)`.

L'arboressence des contextes. Le contexte courant n'est plus le seul contexte. Tous les contextes sont stoqués dans la zone `CONT`. Dans notre prototype, ils forment un arbre orienté vers la racine.

2.1.3 Les zones de codes.

L'entête du programme, ainsi que les entêtes de classe et de méthodes sont spécifiques : elles définissent des variables, méthodes et classes globales pour le/la programme/classe/méthode.

Entête du programme L'entête du programme total contient les déclarations de fonctions.

Entête de méthode L'entête d'une méthode/fonction contient les déclarations des arguments de la méthode.

2.2 Les instructions

2.2.1 Syntaxe psedo-code

Manipulation des contextes

- On utilise `CopyLock : c` pour déclarer une copie du contexte courant, qui pousse un lock au dessus.
- La fonction `Exist(n)` rend maintenant faux si `n` est définit dans le contexte global, mais pas le local.
- De même `Modif(n,v)` ne modifira pas la valeur d'un nom du contexte global.
- Par contre `Get(n)` va d'abord chercher dans le contexte local, puis s'il n'y a pas trouver `n`, va le chercher dans le contexte global.
- Toutes ces fonctions peuvent s'appliquer à un contexte donné. Par exemple si `cont : c` est un contexte, alors `cont.Get(n)` va donner la valeur associée à `n` dans `cont`.

Manipulation des tableaux dynamiques

- `x.Etend()` va doubler la valeur de l'extension du tableau et va recopier le tableau :
```
x.extension = 2*x.extension ;
tab = NewVect(x.extension) ;
for i < x.taille {tab[i]:= x.tab[i];}
x.tab = tab
```
- `NewTab(n)` crée un tableau vide de taille n :
```
ext = Ceil(log(n))
t = NewVect(ext) ;
return Tab{taille = n, extension = ext, tab=t}
```
- `tabdy1` $+_D$ `tabdy2` concatène les deux tableaux : [3]
```
taille = tabdy1.taille + tabdy2.taille
res = NewTab(taille)  ;
for i in [0..tabdy1.taille-1] do res.tab[i]:=tabdy1.tab[i]
for i in [0..tabdy2.taille-1] do res.tab[tabdy1.taille+i]:=tabdy2.tab[i]
return res
```

Manipulation des piles de noms Les fonctions `Pop`, `Push` et `Pull` s'appliquent aussi à des piles de noms.

3. Sa présence est une facilité pour le projet, normalement il faudrait l'encoder avec les autres opérateurs.

2.2.2 Tableaux/listes

Instruction	sémentique	pile avant	pile après
GetTD	Push(Pop.tab[Pop])	d:pile	X:pile
ModiTD	Pop.tab[Pop] := Pop)	X:d:pile	pile
AppendTD	Pop.taille + +	d:pile	pile
EtendTD	Pop.Etend	d:pile	pile
TestTD	list = Pop Push(list.taille == list.extension)	d:pile	pile
NewTD	Push(NewTab(Pop))	e:pile	d:pile
AddTD	Pop $+_D$ Pop	e:pile	d:pile

2.2.3 Appels de méthodes

Instruction	sémentique	pile avant	pile après
Args	v := Pop clot := Pop n := clot.args.Pop clot.cont := clot.cont.Insert(n, v) Push(clot)	X:c:pile	c:pile
Appel	clot = Pop Assert(IsEmpty(clot.args)) Push(CC) Push(PC) CC := clot.cont PC := clot.methode	l:pile	a:a:pile
Return	res := Pop PC := Pop CC := Pop Push(res)	X:a:a:pile	X:pile

2.2.4 Déclarations de méthodes et d'arguments

Instruction	sémentique	pile avant	pile après
AjoutVide n	Insert(n, Null)		
Lambda off	clot = NewCloture{cont = CopyLock, methode = PC + off + 1, args = NewPile} Push(clot)	pile	l:pile
DeclA n	Pull.args.Push(n)	l:pile	l:pile

2.3 Exemple de compilation

Voici un petit programme avec fonction et une version compilée :

2.4 Détails d'implémentation du prototype

Implémentation possible du contexte On utilise des noeuds spéciaux appelés locks pour séparer le contexte local du global. Ainsi `Exist(n)=Exists_rel(n,CC)` est définit par :

```
Exist_rel(n,c) = c != Nullc  && c != lock && (c.nom == n || Exist_rel(n,c.cont))
```

2.5 Différences et points communs avec Python

Contextes chainés ou statiques Python utilise l'autre implémentation des contextes : ce sont des contextes de taille statiques représentés par des tables de hashage que l'on copie à chaque création de clôture.

Locks En python, le lock est posé directement sur la variable dans le contexte considéré. On peut faire ca car les contxtes ne sont pas partagés entre clôtures. Celà permet d'être plus fin sur les variables modifiables (ex : mot-clef global en python).

3 La Mini-Py-machine pour le Package 3

3.1 La sémantique

3.1.1 Types de données.

Un objet. Noté `(o)`. Un objet `x : o` contient uniquement un contexte `x.cont : *c`, donnant les attributs et les fonctions. Ce contexte est un peu spécifique en ce qu'il ne contient pas d'autre variables global.

Une classe. Noté[4] `(t)`. Une classe est un objet particulier de la classe "Classe", elle contient forcément une méthode `x.__init__ : l` dans son context.

3.1.2 Les zones de codes.

L'entête du programme, ainsi que les entêtes de classe et de méthodes sont spécifiques : elles définissent des variables, méthodes et classes globales pour le/la programme/classe/méthode.

Entête du programme L'entête du programme total contient les déclarations de classes et de fonctions globales (une fonction global est une méthode n'apartenant pas a une classe).

Entête de classe L'entête de classe contient les méthodes, les classes emarquées et les arguments. Attention, les arguments doivent être donné en fin de l'entête.

3.2 Les instructions

3.2.1 Syntaxe psedo-code

On utilise `NewObjet{classe = x, cont = y}` pour déclarer un nouvel objet.

3.2.2 Objets

Instruction	sémentique	pile avant	pile après
Point n	`Push(Pop.cont.Get(n))`	`o:pile`	`X:pile`
Constr	`Push(Pop.cont.Get("__init__"))`	`t:pile`	`l:pile`

4. "t" pour type, car en Python, types et classes représentent la même chose.

3.2.3 Déclarations de classe, méthodes et d'argument

Instruction	sémentique	pile avant	pile après
NewObj n	Push(NewObjet{classe = n, cont = Nill})	pile	o:pile
NewClass n	Push(NewObjet{classe = "Classe", cont = NewClot{"__init__", Null, Vide}})	pile	t:pile
AjouterObj n	Pull.cont.Inserer(n, Pop)	X:o:pile	o:pile
AjouterConstr	Pull.cont.Inserer("__init__", Pop)	l:t:pile	t:pile
LockerObj n	Pull.cont.Lock	o.pile	o.pile
ModifierObj n	Pop.cont.Modif(n, Pop)	X:t:pile	pile
DeclStatic n	Pull.cont.Ajouter(n, Pop)	X:t:pile	t:pile