Modèles de mémoire pour l'enseignement de la programmation

Léo Exibard[1 [0000−0003−0318−1217]], Nadime Francis[1 [0009−0009−4531−7435]], Antoine Meyer[1 [0000−0003−4513−4347]], and Marie Van Den Bogaard[1 [0009−0007−2070−1196]]

LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-Vallée, France
{leo.exibard,nadime.francis,antoine.meyer,
marie.van-den-bogaard}@univ-eiffel.fr

Résumé Dans cette communication, nous proposons une réflexion sur la construction de représentations et de modèles mentaux du fonctionnement de la mémoire dans le cadre d'un enseignement d'initiation à la programmation, et de ses liens avec l'acquisition des concepts de variable et d'affectation. Après avoir donné plusieurs exemples de modèles de mémoire envisageables dans le discours enseignant, nous choisissons comme cas d'étude l'apprentissage du langage Python, prescrit par les programmes officiels du lycée en France et très utilisé en premier cycle d'enseignement supérieur. Nous présentons les principales caractéristiques du modèle de mémoire de Python, en particulier dans son implémentation de référence (CPython). Ces observations nous amènent à formuler des hypothèses quant aux conséquences possibles du choix d'un modèle de mémoire sur l'apprentissage de la programmation.

Keywords: Didactique de l'informatique · Sémantique des langages de programmation · Machine notionnelle · Diagramme mémoire

1 Introduction

L'apprentissage de la programmation, qu'il concerne la découverte d'un premier langage ou qu'il s'appuie sur des connaissances antérieures, suppose l'appropriation conjointe de règles de syntaxe et de sémantique (comme souligné par [6]). Dans sa thèse, Nguyen [9, partie B] livre une réflexion sur l'impact possible de la présence (explicite ou non) ou de l'absence dans le propos didactique d'une *machine de référence*, en prenant appui sur plusieurs traités et sur un corpus de manuels scolaires. Il insiste aussi sur l'émergence progressive dans l'histoire de l'informatique de la notion de *mémoire effaçable*, qui sous-tend les conceptions actuelles les plus courantes sur les notions de variable et d'affectation. Dans un article proposant, dès 1985, un panorama de quelques enjeux de recherche en didactique de l'informatique, Rogalski [10] souligne l'importance de la prise en compte des représentations et modèles « spontanés » des apprenantes [1] quant aux concepts fondamentaux liés à l'algorithmique et à la programmation.

1. Nous choisissons arbitrairement d'employer dans ce texte le genre grammatical féminin partout où il sera question de personnes génériques.

Parallèlement au propos de la communauté didactique, les langages de programmation employés dans l'enseignement ont connu une évolution certaine. Logo et Turbo Pascal, largement utilisés dans les années 1980 – 1990 (respectivement à l'école élémentaire et dans les classes préparatoires), ont aujourd'hui essentiellement disparu de l'enseignement public français. Dans les collèges, à la faveur de l'introduction de l'informatique dans les programmes de mathématiques et de technologie en 2016, le langage visuel Scratch, héritier de Logo, s'impose progressivement. Entre 2009 et 2019 le lycée a vu apparaître des contenus informatiques, d'abord dans les programmes de mathématiques puis sous la forme de matières spécifiques (comme « Informatique et Sciences du Numérique » (ISN) en terminale en 2012). Dans cette période ont été créés plusieurs langages « pédagogiques » tentant de s'adapter aux recommandations officielles (voir [7] pour une discussion de l'influence possible d'un tel langage). En 2019 le langage Python, déjà populaire dans le monde professionnel et à l'université (où il tend à remplacer le C, le Java simplifié ou encore le Scheme comme premier langage) devient le langage de référence dès la classe de seconde en mathématiques, et pour la nouvelle spécialité Numérique et Sciences Informatiques (NSI).

Face à l'évolution des langages, la question du choix d'un modèle d'exécution ainsi que de son degré d'explicitation nous semble donc importante. Dans ce travail exploratoire, nous nous intéressons spécifiquement au modèle de mémoire qui sous-tend la sémantique du langage Python. Nous portons une attention particulière sur les concepts de variable, valeur et type, sur ceux de référence mémoire, d'*aliasing* et de mutabilité, sur les différentes zones de mémoire (pile, tas) et leur fonction et sur la sémantique des différentes formes d'affectation (y compris le passage de paramètres). Sans volonté normative, nous souhaitons contribuer à décrire les enjeux du choix d'un modèle de mémoire « de travail » par l'enseignante et formulons des hypothèses quant à ses conséquences possibles sur les apprentissages. Nous nous posons en particulier la question du niveau de détail des modèles, de leur efficacité pour la résolution des tâches proposées, de leur validité vis-à-vis de ce qu'on pourrait qualifier de « modèle de référence », et de leur impact possible sur les apprentissages ultérieurs.

Nos réflexions sont liées au concept de *machine notionnelle*, défini par du Boulay et al. [4] comme « le modèle idéalisé de l'ordinateur induit par les structures d'un langage de programmation » et exploré depuis par de nombreux travaux. Ces machines posent la question d'un nécessaire compromis entre complétude (permettre d'expliquer les comportements des programmes), cohérence (permettre d'établir des prédictions conformes à la sémantique du langage) et simplicité (niveau de détail et d'abstraction les rendant accessibles aux apprenants). Dans [2], après un intéressant survol bibliographique, les auteurs insistent sur la distinction entre une machine notionnelle proprement dite et les outils de visualisation associés (même si ces deux aspects sont liés). Des travaux proches sur les diagrammes mémoire [3] mettent en lumière les enjeux du choix et de l'adaptation d'une représentation de l'état mémoire selon les objectifs d'enseignement poursuivis. Dans cet article, nous nous appuyons sur les représentations produites par l'outil Python Tutor [5], non pour les étudier pour elles-mêmes

mais comme simple illustration des machines notionnelles sous-jacentes. Nous nous appuyons aussi (pour l'instant de manière informelle) sur l'idée de *conception* proposée par Vergnaud [11] puis intégrée par Balacheff au modèle cK¢[1], repris par Modeste [8] dans son étude du concept d'algorithme, qui pourra servir à une analyse plus complète dans la poursuite de ce travail.

Le reste de cet article est organisé comme suit. La section 2 présente plusieurs variantes de modèles de mémoire « de travail » possibles, indépendamment d'un langage particulier. La section 3 discute les conséquences possibles du choix de chacun d'eux sur l'enseignement de Python. Enfin, la section 4 propose une synthèse de nos observations et quelques pistes de recherche. L'annexe A définit brièvement quelques termes liés à notre questionnement.

2 Modèles de mémoire pour l'enseignement

Nous distinguons ici trois catégories de « modèles de travail » couramment utilisés dans l'apprentissage de la programmation, chacune admettant un certain nombre de variantes ou d'hybridations. Ces modèles reflètent avec un degré de fidélité plus ou moins grand la *sémantique* du langage étudié, c'est-à-dire le comportement (documenté ou observé) des programmes écrits dans ce langage. Ils emploient des métaphores supposées faciliter l'apprentissage d'un langage, en particulier des concepts de variable et d'affectation. Nous illustrons ces modèles à l'aide de visualisations générées par l'outil Online Python Tutor [2] [5].

2.1 Modèles à boîtes

Cette première catégorie de modèles s'appuie sur l'idée que la mémoire est structurée sous forme de « boîtes » ou « tiroirs » que l'on peut désigner, ouvrir ou fermer afin d'accéder à leur contenu. Selon les descriptions, on pourra insister sur le caractère consécutif des boîtes, leur numérotation, leur capacité, etc. Un tel exemple apparaît dans l'un des manuels analysés par Nguyen [3][9, p. 36] :

> Dans notre modèle de l'ordinateur, chaque variable est associée à une boîte de stockage. Une étiquette du nom de variable est collée sur la boîte. Dans la boîte se trouve un papier sur lequel est écrit la présente valeur de la variable. (...) Chaque boîte possède un couvercle que l'on peut ouvrir pour affecter une nouvelle valeur à la variable. De plus, il y une fenêtre grâce à [laquelle] on peut lire cette valeur sans la modifier.

On notera qu'il n'est pas fait mention dans cet extrait de la notion de *type*, bien que cela soit susceptible d'apparaître dans la suite du manuel. L'adjonction de cette notion imposerait de statuer sur l'existence de plusieurs sortes de boîtes de capacités variées (au sens de la quantité d'information qu'elles peuvent contenir), et sur le genre d'objets qu'il est permis d'y stocker. Il est possible d'interpréter cette description de deux manières différentes.

2. Disponible sur https://pythontutor.com, captures effectuées le 13/10/2023.

3. Le manuel M2, qui est le seul du corpus étudié à fournir une description de machine destinée à exécuter les programmes.

Boîtes à étiquettes fixes. Premièrement, on peut supposer que chaque variable possède sa boîte propre, créée lorsque la variable est déclarée ou initialisée pour la première fois, et qu'une étiquette reste associée à la même boîte tout au long de sa durée de vie. Dans ce cas, la sémantique probable de l'affectation (disons `x = expr` avec `expr` une expression quelconque) consiste premièrement à construire (ou à localiser s'il existe déjà) l'objet `obj` résultant de l'évaluation de `expr`, puis à recopier `obj` dans la boîte associée à `x`. Le mode de passage de paramètre y correspondant naturellement serait un passage par valeur, chaque appel de fonction créant sur la pile des boîtes correspondant à ses paramètres et variables locales. La figure 1 montre une visualisation possible pour un tel modèle.

Boîtes à étiquettes déplaçables. La seconde interprétation suppose que l'étiquette représentant une variable puisse être repositionnée sur une boîte différente. Lors d'une affectation, l'objet `obj` est créé en mémoire (ou localisé), et l'étiquette `x` est apposée à la boîte le contenant. La boîte précédemment désignée par `x` devient alors inaccessible, à moins qu'elle ne soit encore étiquetée par une autre variable. Dans ce type de modèle, il n'est pas évident de déterminer un mode de passage de paramètre cohérent : sans recopie des objets, comment par exemple distinguer les paramètres de fonctions des variables déjà existantes ?

2.2 Modèles à fils et épingles

Dans cette catégorie de modèles, on postule que l'accès aux objets est assuré par l'intermédiaire de références (Cf. annexe A), représentées par des flèches ou « fils » reliant un nom (par exemple une variable ou un attribut d'objet) à l'objet qu'il désigne. Les variables sont recensées dans une zone ne contenant aucun objet, typiquement, un espace de noms stocké dans la pile d'appels, et les objets sont tous alloués dans le tas. Selon les cas, un type peut être associé à chaque variable ou attribut (typage statique), ou seulement à l'objet désigné (typage dynamique). Une variable désignant un objet est reliée par l'intermédiaire d'un fil (« épinglée ») à l'emplacement mémoire contenant les données de l'objet. Les objets composites ou objets de type conteneur (par exemple les tableaux) sont également reliés aux objets qu'ils contiennent par le même procédé.

Dans ce modèle comme dans le modèle à étiquettes déplaçables, l'affectation `x = expr` crée ou localise en premier lieu dans le tas l'objet `obj` issu de l'évaluation de `expr`. Ensuite, l'étiquette `x` (après avoir été créée le cas échéant) est reliée par un fil à l'emplacement mémoire le contenant. Le mode de passage de paramètres correspondant est le passage par référence, et l'affectation d'un attribut à un objet ou d'un élément à un tableau suit le même principe. Il est en outre aisé de regrouper clairement l'ensemble des étiquettes (variables) appartenant à un même contexte d'exécution ou à une même portée grâce à la pile : variables globales, variables locales à un appel de fonction, etc. La figure 2 montre l'exécution du programme précédent dans un modèle de ce type.

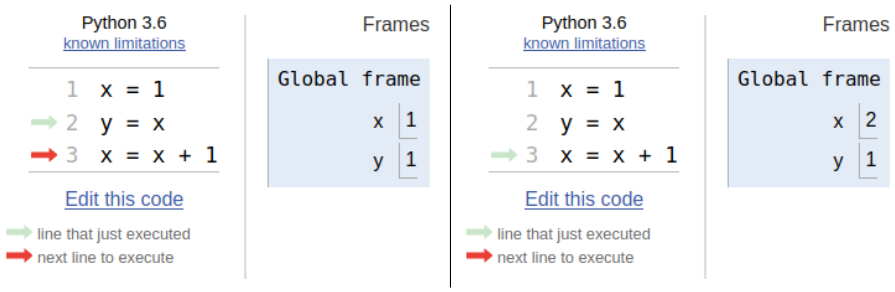

Figure 1. Représentation de type « boîtes à étiquettes fixes » dans Python Tutor. Ici, `x` et `y` désignent deux boîtes indépendantes, même après l'instruction `y = x`. La zone *Frames* représente l'état de la pile, le cadre bleu l'espace de noms global.

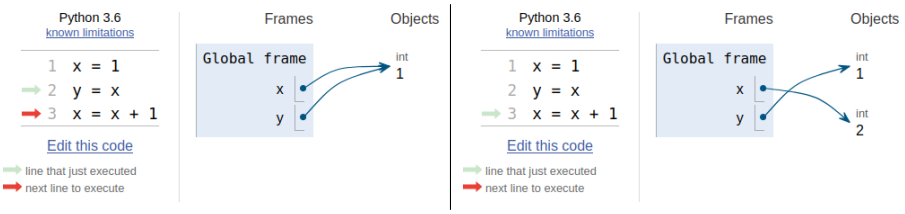

Figure 2. Représentation de type « fils » dans Python Tutor. La zone *Objects* représente le tas. Chaque objet y est représenté par sa valeur surmontée de son type. Ici, `x` et `y` désignent la même boîte après l'instruction `y = x` (aliasing), et `x = x + 1` provoque la création d'un nouvel objet en mémoire, laissant `y` inchangée.

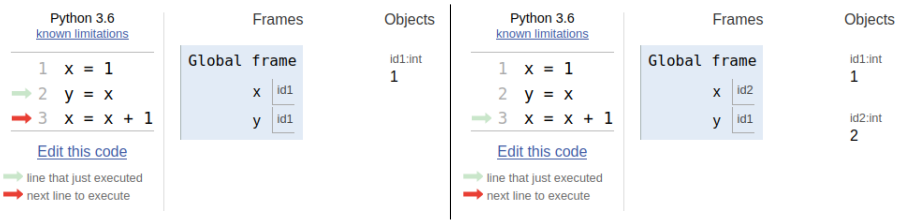

Figure 3. Représentation de type « adresses » dans Python Tutor. La sémantique est semblable au déroulement représenté en Figure 2.

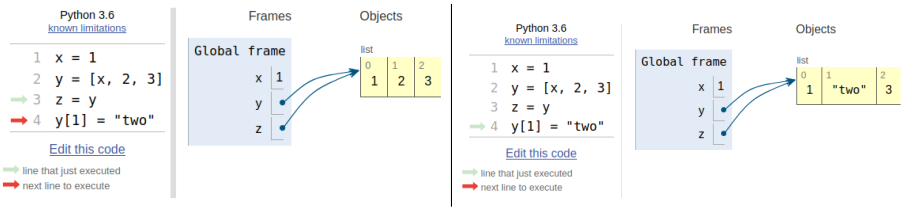

Figure 4. Représentation mixte dans Python Tutor. Les entiers et chaînes de caractères sont stockées par valeur, les listes par référence. Cet exemple illustre également l'effet de la modification d'un objet mutable en présence d'aliasing (une modification de la liste désignée par `y` est visible depuis la variable `z`).

2.3 Modèles à adresses

Dans cette catégorie de modèles, on distingue deux types de valeurs : les valeurs d'objets (ou données) et les adresses (ou emplacements mémoire). Le principe d'indirection présent dans les modèles à fils est maintenu, mais il est rendu plus explicite : lors d'une affectation `x = expr`, l'objet `obj` issu de l'évaluation de `expr` est créé ou localisé en mémoire, et son *adresse* est déterminée. Celle-ci est alors stockée dans une zone ad-hoc (par exemple sur la pile), qu'on peut voir comme un registre associant à chaque variable une adresse, ou une collection de boîtes contenant l'adresse de l'objet désigné par chaque variable. La figure 3 montre l'exécution du programme précédent selon un tel modèle.

2.4 Modèles mixtes

Il est possible d'envisager des modèles mêlant certaines des caractéristiques des modèles à boîtes, à fils ou à adresses. Au risque d'une plus grande complexité, ceci peut permettre de rendre plus fidèlement compte de la sémantique de référence du langage étudié. L'une de ces variantes consiste à représenter les objets de types « primitifs » (comme les nombres entiers ou les caractères) comme stockés dans des boîtes à étiquettes fixes, et de désigner les objets composites (y compris conteneurs) par l'intermédiaire de fils ou d'adresses (c'est-à-dire par référence). Une visualisation de ce type est représentée dans la figure 4.

Ce type de modèle mixte est par exemple assez proche de la sémantique réelle de langages comme Java, voire comme le C où les deux mécanismes existent. Dans ce dernier cas, une adaptation supplémentaire est requise pour tenir compte du fait que les adresses mémoires peuvent être manipulées directement par la programmeuse au même titre que tout autre objet.

3 Discussion : le cas de Python

Python est un langage interprété, dont l'implémentation de référence (écrite en C) est CPython. Dans ce langage, toutes les données d'un programme (y compris fonctions, classes, modules, etc.) sont des objets [4]alloués sur le tas. Chaque objet possède un type et un identifiant assimilable à une adresse mémoire (tous deux invariants), et une valeur potentiellement modifiable (objets mutables). Les adresses peuvent être consultées mais ne sont jamais manipulées directement par la programmeuse. L'allocation et la libération de mémoire sont prises en charge par l'interpréteur. Ce modèle de mémoire est *homogène* au sens où il ne repose que sur un seul mécanisme, celui de référence, contrairement à des langages comme C ou Java. Variables et attributs d'objets sont des références vers d'autres objets, et le passage de paramètres à une fonction s'effectue par référence. Ainsi, Python repose fortement sur le phénomène d'aliasing, dont la documentation indique qu'« [i]l n'apparaît pas au premier coup d'œil en Python

4. Documentation Python – 3. Modèle de données. https://docs.python.org/fr/3/reference/datamodel.html, consulté le 16/10/2023.

et [qu']il peut être ignoré tant qu'on travaille avec des types de base immu[t]ables (nombres, chaînes, n-uplets) », mais prévient que « les alias peuvent produire des effets surprenants » dans le cas contraire [5].

En raison de ces caractéristiques, il semble que les modèles à boîte (à étiquettes fixes ou déplaçables) présentent des domaines de validité trop restreints pour représenter fidèlement l'ensemble des concepts enseignés dans la plupart des cours d'initiation à Python. En effet, les progressions habituelles sont susceptibles d'aborder relativement tôt les questions relatives à la structuration des espaces de noms (variables globales *vs.* variables locales, phénomène de masquage), au passage de paramètres mutables (typiquement des listes ou dictionnaires) et à leur éventuelle modification par le corps d'une fonction, ou encore aux phénomènes d'aliasing fréquemment rencontrés dans la construction de listes imbriquées. De même, il peut être malaisé dans ces modèles d'expliquer par des métaphores satisfaisantes le fonctionnement des objets de taille arbitraire (comme les entiers de Python ou les conteneurs imbriqués), sauf à admettre qu'il est possible de « faire tenir » une quantité illimitée d'information dans une zone mémoire finie, ce qui peut entrer en contradiction avec l'acquisition d'autres concepts de l'informatique comme le codage binaire des données.

A contrario, la plupart des caractéristiques de Python évoquées ci-dessus peuvent être prises en charge par les modèles à base de fils ou d'adresses. On remarque que ces modèles mettent nettement en évidence la notion d'aliasing dans l'explication et la visualisation de la sémantique de programmes même très simples, et induit une séparation claire entre pile et tas, noms (sources de flèches) et objets (désignés par des flèches).

Cependant, les modèles à fils peuvent sembler moins concrets (les objets semblant « flotter » dans une mémoire amorphe), et ne permettent pas aisément de porter un discours sur l'aspect contigu des emplacements mémoire. Les modèles à adresses introduisent quant à eux une notion d'indirection relativement complexe, et présentent un potentiel risque de confusion entre valeurs d'adresses et valeurs d'objets. En outre, ces deux types de modèles sont susceptibles d'imposer à l'enseignante ainsi qu'aux apprenantes déjà exposées à un autre modèle de mémoire une remise en question plus profonde de conceptions antérieures. La question générale de déterminer si de tels modèles peuvent être globalement perçus par les apprenantes comme plus complexes qu'un modèle à boîtes reste ouverte, et peut justifier la mise en place d'un travail expérimental.

4 Conclusion et perspectives

Nous avons proposé une discussion sur les domaines de validité de quelques modèles de mémoire « de travail » dans le cas du langage Python. Nous considérons que ces observations restent valables pour l'enseignement d'autres langages de programmation, au sens où elles mettent l'accent sur l'importance du choix d'un modèle de mémoire adapté à la sémantique du langage choisi et aux

5. Tutoriel Python – 9. Classes. https://docs.python.org/fr/3/tutorial/classes.html, consulté le 16/10/2023.

concepts à enseigner, quels qu'ils soient. Il nous semblerait utile d'approfondir ce début d'analyse *a priori* à l'aide d'un cadre théorique rigoureux, tel que celui proposé par Balacheff [1], notamment en termes de validité, cohérence et efficacité des conceptions sous-tendues par chaque modèle. On portera une attention particulière à ce que les auteurs nomment « structures de contrôle », qui permettent d'une part d'attester de la non-contradiction de la conception par rapport à la réalité, et de l'état (résolu ou non) d'un problème. Dans le contexte de la programmation, l'interpréteur offre des moyens de contrôle potentiellement immédiats, qu'il conviendrait d'analyser avec soin.

Nous faisons l'hypothèse que le choix d'un modèle de mémoire de la part de l'enseignante est susceptible d'entraîner des effets didactiques importants sur la formation de conceptions liées à la sémantique du langage ainsi qu'aux objets fondamentaux de l'informatique. Une première piste d'expérimentation permettant de mettre à l'épreuve cette hypothèse serait de relever dans les discours d'enseignantes, dans les ressources qu'elles utilisent et dans les productions de leurs élèves ou étudiantes des indices de conceptualisations liées au modèle de mémoire. L'analyse de ce corpus pourrait permettre d'approfondir notre connaissance des modèles de travail utilisés en classe, et de tenter de mettre en évidence leur effet possible sur les conceptualisations des apprenantes et sur leur acquisition des savoirs et savoir-faire visés.

D'autres apprentissages, parallèles ou ultérieurs, par exemple sur la représentation et le codage des données, sur l'acquisition de langages de programmation de caractéristiques différentes, sur l'analyse d'algorithmes et de programmes (notamment leur complexité) ou encore sur la conception de compilateurs ou d'interpréteurs, nous semblent requérir une perception raisonnablement précise de la structure de la mémoire et de ses usages les plus courants. Nous faisons l'hypothèse que la manière dont ces notions sont initialement rencontrées et « fréquentées » peut favoriser ou au contraire retarder certains des ces apprentissages. On peut s'intéresser également à la question de la pérennité de chaque modèle au fil des apprentissages, et à celle de leur dépassement progressif.

Références

1. Balacheff, N., Margolinas, C. : cK¢. Modèle de connaissances pour le calcul de situations didactiques. In : Balises en didactique des mathématiques : Cours de la 12e école d'été de didactique des mathémathiques, pp. 1–32. Recherches en Didactique des Mathématiques, La Pensée Sauvage (2005)
2. Dickson, P.E., Brown, N.C.C., Becker, B.A. : Engage Against the Machine : Rise of the Notional Machines as Effective Pedagogical Devices. In : Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education. pp. 159–165. ACM, Trondheim Norway (Jun 2020)
3. Dickson, P.E., Dragon, T. : A Memory Diagram for All Seasons. In : Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1. pp. 150–156. ACM, Virtual Event Germany (Jun 2021)

4. du Boulay, B., O'Shea, T., Monk, J. : The black box inside the glass box : presenting computing concepts to novices. International Journal of Man-Machine Studies **14**(3), 237–249 (Apr 1981)
5. Guo, P.J. : Online Python Tutor : Embeddable Web-Based Program Visualization for CS Education. In : Proceeding of the 44th ACM technical symposium on Computer science education. pp. 579–584. SIGCSE '13, Association for Computing Machinery, New York, NY, USA (Mar 2013)
6. Hoc, J.M., Nguyen-Xuan, A. : Chapter 2.3 - Language Semantics, Mental Models and Analogy. In : Hoc, J.M., Green, T.R.G., Samurçay, R., Gilmore, D.J. (eds.) Psychology of Programming, pp. 139–156. Academic Press, London (Jan 1990)
7. Meyer, A., Modeste, S. : Rôle d'un logiciel dans la transposition didactique du concept d'algorithme : le cas du logiciel AlgoBox en France et des programmes du lycée entre 2009 et 2019. Cahiers d'histoire du Cnam **vol.15**(1), pp. 101 (2022)
8. Modeste, S. : Prendre en compte l'épistémologie de l'algorithme. Quels apports d'un modèle de conceptions ? Quelle transposition didactique ? Recherches en didactique des mathématiques **38**(1), 1–15 (2021)
9. Nguyen, C.T. : Etude didactique de l'introduction d'éléments d'algorithmique et de programmation dans l'enseignement mathématique secondaire à l'aide de la calculatrice. Ph.D. thesis, Université Joseph-Fourier - Grenoble I (Dec 2005)
10. Rogalski, J. : Alphabétisation informatique. Bulletin de l'APMEP **347**, 61–74 (1985)
11. Vergnaud, G. : La théorie des champs conceptuels. Recherches en Didactique des Mathématiques **10**(2.3), 133–170 (1990)

A Définitions

On définit ici de manière informelle quelques concepts de référence utiles. Ces définitions sont indicatives, elles admettent des variantes et des exceptions selon les contextes. Nous nous gardons volontairement de les détailler outre mesure.

Objet, valeur et type. On appellera *objet* (dans un sens général, et non au sens de la programmation orientée objet) tout littéral ou plus généralement toute entité résultant de l'évaluation d'une expression. Chaque objet possède a priori une *valeur* (les données dont il est constitué) et un *type* qui peut déterminer, entre autres, les opérations qu'on peut lui appliquer, et potentiellement la manière dont ses données seront stockées en mémoire (ce dont la programmeuse n'a pas nécessairement connaissance).

Variable. Ce terme désigne généralement une entité permettant de désigner un objet par un nom à un moment donné de l'exécution d'un programme. Le détail du fonctionnement et des caractéristiques des variables dépend fortement du langage de programmation considéré.

Espace de noms. Dans la terminologie propre à certains langages, on appelle ainsi les « répertoires » de noms accessibles à un instant donné de l'exécution. Généralement, plusieurs espaces de noms co-existent (global, locaux, propres à un objet, etc.). Leur gestion relève des mécanismes de *portée* et de *résolution de noms*, qui diffèrent selon les langages.

Politique de typage. On distingue généralement (outre les langages non ou faiblement typés) les langages *statiquement typés* (comme Java ou C), dans lesquels les vérifications de type sont effectuées à la compilation, des langages *dynamiquement typés*, où cette vérification est faite à l'exécution.

Adresse, référence, pointeur. Une *adresse* est une donnée (généralement un nombre entier) permettant d'identifier de manière unique un emplacement mémoire[6]. Dans plusieurs langages, il est possible de désigner des objets de manière indirecte par le biais de leur adresse. On parle de *références* lorsque cette « indirection » est assurée sans contrôle de la programmeuse (comme en Java ou en Python), et de *pointeurs* lorsqu'un langage permet une manipulation directe de l'adresse (comme en C).

Pile d'appel. La *pile d'appel* désigne une zone de mémoire consacrée à la gestion et à la bonne exécution des appels de fonctions (potentiellement imbriqués). Elle a pour rôle le maintien du flot de contrôle correct d'un programme lors d'appels de fonctions, l'identification des paramètres reçus ainsi que la transmission du résultat d'un appel au contexte englobant.

Tas. Le *tas* est la zone de mémoire généralement consacrée au stockage des valeurs. Selon les cas, ceci peut concerner les valeurs désignées par des variables globales, les valeurs stockées dans des zones de mémoire allouées dynamiquement, voire toutes les valeurs utilisées par un programme.

6. On ne rentrera pas ici dans la distinction entre adresses réelles et virtuelles.

Politique de passage de paramètres. Lorsqu'une fonction reçoit des paramètres, on parle de *passage par valeur* quand la valeur de chaque paramètre est recopiée dans une zone de mémoire locale (par exemple la pile), et de *passage par référence* lorsque seule son adresse est transmise. Dans ce cas, différents mécanismes (par exemple en C ou C++) peuvent permettre d'autoriser ou d'interdire la modification des valeurs reçues.

Aliasing. C'est le fait pour un objet d'être désigné par plusieurs variables ou plusieurs noms (attributs d'objets, éléments de tableaux...). Ceci peut être source d'efficacité (passage par référence d'un objet volumineux à une fonction, partage de données dans la programmation orientée objet) ou source d'erreur (modification inopinée d'un objet partagé).

Mutabilité. Cet anglicisme désigne le fait, pour un objet, d'être modifiable sur place en mémoire. En présence d'aliasing, ceci peut provoquer des comportements inattendus. En programmation fonctionnelle dite *pure*, on interdit parfois totalement la manipulation de valeurs mutables.

D'autres concepts sont liés à notre questionnement et mériteraient un traitement plus approfondi. Nous ne pouvons tous les détailler faute de place.