École Doctorale Sciences, Technologies et Ingénierie

THÈSE

pour obtenir le grade de docteur délivré par

Université Assane Seck de Ziguinchor

Mention : Informatique

Spécialité : Données et Connaissances

présentée et soutenue publiquement par

Gorgoumack SAMBE

le 13 Août 2021

Contribution au développement des compétences de résolution de problèmes durant l'initiation à la programmation

Directeur de thèse : **Marie Salomon SAMBOU**
Co-encadrant de thèse : **Adrien BASSE**

Jury

M. Moussa LO, Professeur, Université Virtuelle du Sénégal	Président/Rapporteur
M. Ibrahima FALL, Maître de Conférence, École Supérieure Polytechnique de Dakar	Rapporteur
M. Amadou Dahirou GUEYE, Maître de Conférence, Université Alioune Diop de Bambey	Rapporteur
M. Youssou FAYE, Maître de Conférence, Université Assane Seck de Ziguinchor	Examinateur
M. Marie Salomon SAMBOU, Professeur, Université Assane Seck de Ziguinchor	Directeur
M. Adrien BASSE, Maître-assistant, Université Alioune Diop de Bambey	Encadrant

THESE

École Doctorale Sciences, Technologies et Ingénierie

THÈSE

pour obtenir le grade de docteur délivré par

Université Assane Seck de Ziguinchor

Mention : Informatique

Spécialité : Données et Connaissances

présentée et soutenue publiquement par

Gorgoumack SAMBE

le 13 Août 2021

Contribution au développement des compétences de résolution de problèmes durant l'initiation à la programmation

Directeur de thèse : **Marie Salomon SAMBOU**
Co-encadrant de thèse : **Adrien BASSE**

Jury

M. Moussa LO, Professeur, Université Virtuelle du Sénégal	Président/Rapporteur
M. Ibrahima FALL, Maître de Conférence, École Supérieure Polytechnique de Dakar	Rapporteur
M. Amadou Dahirou GUEYE, Maître de Conférence, Université Alioune Diop de Bambey	Rapporteur
M. Youssou FAYE, Maître de Conférence, Université Assane Seck de Ziguinchor	Examinateur
M. Marie Salomon SAMBOU, Professeur, Université Assane Seck de Ziguinchor	Directeur
M. Adrien BASSE, Maître-assistant, Université Alioune Diop de Bambey	Encadrant

THESE

Résumé

L'initiation à l'algorithmique et à la programmation est fondamentale dans les formations à vocation scientifique informatique. L'apprenant doit être capable, à la fin de son cours d'initiation, d'analyser un problème algorithmique simple et de concevoir et d'évaluer une solution à ce problème.

Les taux d'abandon et d'échec sont relativement élevés et la faiblesse des compétences de résolution de problèmes est vue comme l'une des causes principales. Ces compétences ne sont pas explicitement intégrées dans beaucoup de curricula et les formateurs se focalisent assez souvent sur la syntaxe des langages.

Cette thèse s'inscrit dans le cadre des Environnements Informatiques pour l'Apprentissage Humain (EIAH). Nous proposons un système pour le soutien au développement des compétences de résolution de problèmes durant l'initiation à l'algorithmique et à la programmation. Notre approche s'appuie sur deux stratégies :

1. le guidage de l'apprenant et l'explicitation des concepts durant le processus de résolution de problèmes.
2. l'analyse et la comparaison sémantique du code source de l'apprenant avec un code expert afin de lui proposer un feedback sur la sémantique de son code source.

Ces deux stratégies ont montré un impact positif sur les compétences de résolution de problèmes durant l'apprentissage de la programmation.

Le processus de guidage proposé s'est voulu simple et est basé sur une méthode de conception de programmes guidée par les données et prenant en compte l'aspect compréhension du problème qui est essentiel.

Pour l'analyse et la comparaison de la sémantique de codes sources, nous avons proposé une méthode statique automatique basée sur le calcul formel. Cette méthode a l'avantage de demander moins d'efforts de la part du formateur comparée aux méthodes dynamiques qui se basent sur l'exécution du code et aux méthodes statiques manuelles basées sur l'analyse du code par un expert humain.

Nous avons mis en place un prototype du système, IDE4SCAPSS, avec le langage python pour le guidage, l'analyse et la comparaison sémantique de programmes codés en Pascal. Le système a été évalué sur des solutions expertes de problèmes utilisant des structures séquentielles et conditionnelles.

Mots-clés : initiation à la programmation, résolution de problèmes, comparaison sémantique de codes sources, guidage de l'apprenant.

Abstract

The introduction to algorithms and programming is fundamental in computer science training. At the end of any introductory course, the learner should be able to analyse a simple algorithmic problem and to design and evaluate a solution to any given problem.

Accordingly, drop-out and failure rates are relatively high and weakness of problem-solving skills is one of the causes of dropout and failure. Most curricula ignore these problem solving skills while teachers focus more on the syntax of the language and scarcely on problem-solving skills.

This thesis is in the context of Computer Based Learning Environnement (CBLE). Along this study, we propose a system to scaffold problem-solving skills during learning programming. Our educational approach is based on two strategies :

1. The guidance of the learner through the problem-solving process in an incremental learning scenario and explanation.
2. The semantic analysis and comparison of the learner's source code with expert code in order to provide feedback on the semantic of the source code.

Those two key strategies showed and revealed a positive impact on problem solving skills during learning programming.

Consequently, the proposed guidance process was intended to be simple and is based on a data-driven method that takes into account the understanding of the problem which is a key component of the problem solving process.

Thus, for the analysis and comparison of source code semantics, we have proposed an automatic static method based on formal computation. This method has the advantage of requiring less effort from the trainer compared to dynamic methods based on code execution and manual static methods based on code analysis by a human expert.

Finally, we implemented a system prototype, IDE4SCAPSS, with the python language for guidance, analysis and comparison of programs coded in Pascal language. The system was evaluated on expert solutions of problems using sequential and/or conditional structures.

Keywords : introduction to programming, problem solving skills, semantic comparison of source codes, guidance of the learner.

Remerciements

Je tiens à adresser ma gratitude à toutes les personnes qui m'ont accompagné et soutenu durant ces années de thèse.

Je ne pense pas que quelques mots de remerciements puissent exprimer ma gratitude à l'endroit de mes encadrants, Marie Salomon SAMBOU et Adrien BASSE, pour m'avoir encadré avec une disponibilité totale et beaucoup de clairvoyance.

Je remercie le Professeur Moussa LO de l'Université Virtuelle du Sénégal pour l'honneur qu'il me fait de rapporter cette thèse et de participer au jury.

Je remercie Monsieur Ibrahima FALL, Maître de Conférence à L'École Polytechnique Supérieure de Dakar, d'avoir accepté d'être rapporteur de cette thèse et membre du jury.

Je remercie également Monsieur Amadou Dahirou GUEYE, Maître de Conférence à l'Université Alioune Diop de Bambey, d'avoir accepté d'être rapporteur de cette thèse et membre du jury.

Je remercie Monsieur Youssou FAYE, Maître de Conférence à l'Université Assane Seck de Ziguinchor, d'avoir accepté de participer à ce jury de thèse.

Je remercie tous les collègues de l'Université Assane Seck de Ziguinchor qui m'ont constamment soutenu durant cette thèse mais plus particulièrement l'équipe du département d'informatique. Je remercie spécialement le collègue Khadim DRAME pour sa collaboration à ce travail.

Je remercie mes parents, mes frères et sœurs, mon épouse, mes enfants et toute la famille pour leur soutien constant et permanent.

Je remercie toutes celles et tous ceux qui de près ou de loin ont participé à l'accomplissement de ce travail.

Table des matières

Table des figures **vii**

Liste des tableaux **ix**

1 Introduction **1**
1.1 Contexte 1
1.2 Problématique 2
1.3 Plan de la thèse 4

2 État de l'art **5**
2.1 Introduction 6
2.2 L'initiation à la programmation 7
2.2.1 Introduction 7
2.2.2 Les curricula 7
2.2.3 Le langage d'initiation 9
2.2.4 L'abandon et les causes d'abandon 12
2.2.5 La programmation et la résolution de problèmes 14
2.2.6 L'initiation à la pensée informatique 16
2.2.7 Conclusion 18
2.3 La résolution de problèmes 20
2.3.1 Introduction 20
2.3.2 Les compétences de résolution de problèmes 20
2.3.3 Des méthodes à l'ingénierie logicielle 22
2.3.4 Les méthodes de conception de programmes 24
2.3.4.1 La Logique de Construction de Programmes . 25
2.3.4.2 La méthode basée objectifs/plans 30
2.3.4.3 Le commun aux méthodes 34
2.3.5 La faiblesse des compétences en initiation 35
2.3.6 Conclusion 38
2.4 Le développement des compétences 39
2.4.1 Introduction 39

2.4.2 Le guidage de l'apprenant et l'explicitation 39
2.4.3 Le travail collaboratif 45
2.4.4 Le feedback sur la sémantique du code 46
2.4.5 Conclusion 50
2.5 Conclusion 52

3 Contributions **53**
3.1 Introduction 54
3.2 Guidage de l'apprenant et explicitations 55
3.2.1 Introduction 55
3.2.2 Cadre théorique 55
3.2.3 Processus de guidage de l'apprenant 57
3.2.4 Conclusion 61
3.3 Analyse et comparaison sémantique de codes 63
3.3.1 Introduction 63
3.3.2 Concepts de bases 63
3.3.3 Cas de la séquence d'instructions 64
3.3.3.1 Définitions 64
3.3.3.2 Processus de calcul 65
3.3.4 Généralisation à la condition 69
3.3.4.1 Définitions 69
3.3.4.2 Processus de calcul et algorithmes 72
3.3.5 Conclusion 79
3.4 Implémentation 80
3.4.1 Introduction 80
3.4.2 Architecture du système 80
3.4.3 Expérimentation et simulation 82
3.4.4 Conclusion 85
3.5 Conclusion 86

4 Conclusion et perspectives **87**
4.1 Résumé des contributions 88
4.2 Limites et Perspectives 89

Bibliographie **91**

A Base de données des problèmes **99**
A.1 Problèmes à solution séquentielle 100
A.2 Problèmes à solution conditionnelle 102
A.3 Problèmes à solution itérative 105

Table des figures

2.1 Taux d'échecs par langages 13
2.2 Taux d'échecs par pays 13
2.3 Fichier Logique de Sorties 26
2.4 Fichier Logique d'Entrées 27
2.5 Unité de traitement 27
2.6 Unité de traitement : calcul de Δ 28
2.7 Structure du programme obtenu 28
2.8 Ordinogramme obtenu par la LCP 29
2.9 Ordinogramme détaillé obtenu par la LCP 30
2.10 Calcul de la moyenne par la méthode basée objectifs/plans . . 32
2.11 Programme obtenu par la méthode basée objectifs/plans . . . 33
2.12 Exemple de transformation de script en AST 49
2.13 Transformation d'un AST en chaîne de token 49

3.1 Processus suivi par l'apprenant sur IDE4SCAPSS 58
3.2 Code source proposé par IDE4SCAPSS 60
3.3 Architecture du système 80
3.4 Processus d'évaluation d'une proposition sur IDE4SCAPSS . . 81

Liste des tableaux

2.1 Langages d'initiation par régions/états 11
2.2 Motifs de choix du langage d'initiation à la programmation . . 11
2.3 Taxonomie de Solo 37
2.4 Activités pour le développement des compétences de résolution de problèmes 44

3.1 Variantes du problème du maximum de trois entier par niveau 57
3.2 Données du problème max de trois entiers dans la base de données 62
3.3 Impact de la condition sur les instructions 73
3.4 Calcul de la sémantique de l'échange de variables - expert . . . 83
3.5 Calcul de la sémantique de l'échange de variables - apprenant 83
3.6 Calcul de la sémantique du signe du produit - expert 84

A.1 Problèmes à solution séquentielle 100
A.1 Problèmes à solution séquentielle 101
A.2 Problèmes à solution conditionnelle 102
A.2 Problèmes à solution conditionnelle 103
A.2 Problèmes à solution conditionnelle 104
A.3 Problèmes à solution itérative 105

Liste des abréviations et acronymes

ACM . . . . . Association for Computing Machinery

ALGOL . . . . . algorithmic language

ANTLR . . . . . ANother Tool for Language Recognition

BASIC . . . . . Beginner's All-purpose Symbolic Instruction Code

COBOL . . . . . COmmon Business Oriented Langage

CORIG . . . . . COnception et Réalisation en Informatique de Gestion

CS1 . . . . . Computer Science 1

EBNF . . . . . Extended Backus-Naur Form

EDI . . . . . Environnement de Développement Intégré

EIAH . . . . . Environnement Informatique pour l'Apprentissage Humain

FOAD . . . . . Formation Ouverte à Distance

FORTRAN . . . . . mathematical FORmula TRANslating system

GCC . . . . . GNU Compiler Collection

IDE . . . . . Integrated Development Environment

IDE4SCAPSS . . . . . IDE 4 scaffolding problem solving skills

IEEE . . . . . Institute of Electrical and Electronics Engineers

JAVAC . . . . . Java Compiler

LCP . . . . . Logique de Construction de Programmes

LISP . . . . . LISt Processing

MOOC . . . . . Massive Open Online Course

PISA . . . . . Programme international pour le suivi des acquis des élèves

PC . . . . . Personal Computer

PL/1 . . . . . Programming Language number 1

POGIL . . . . . Process Oriented Guided Inquiry Learning

PROLOG . . . . . PROgrammation en LOGique

SNOBOL . . . . . StriNg Oriented symBOlic Language

UP . . . . . Unified Process

XP . . . . . eXtreme Programming

Chapitre 1

Introduction

1.1 Contexte

La montée de la demande de formation en informatique et l'orientation dans ce domaine connaît un grand engouement [1]. De nos jours, les emplois et métiers nécessitant des compétences en informatique et particulièrement en programmation sont nombreux et toujours en hausse. Dans [1], les auteurs notent une augmentation significative des inscriptions dans les cours et les programmes de premier cycle en informatique et que la poussée actuelle des inscriptions a dépassé les précédents booms du domaine.

Savoir comment fonctionne le numérique est aujourd'hui un impératif, l'outil technologique est présent dans notre quotidien. Il y'a une augmentation exponentielle des masses de données qu'il est nécessaire d'exploiter pour s'offrir plus d'opportunités. Cela implique de connaître les bases de la conception des systèmes, donc de la programmation informatique.

La programmation informatique est aujourd'hui le socle des formations en informatique toutes spécialités confondues. Elle est également fondamentale dans beaucoup de spécialités en mathématiques, en physique, en électronique et dans beaucoup d'autres domaines. Elle s'est d'ailleurs généralisée dans les filières non scientifiques et au collège parce qu'elle permet de développer les compétences de résolution de problèmes et de se confronter ainsi aux problèmes de la vie [2].

On constate toutefois que les taux d'échec et d'abandon sont élevés et que les apprenants rencontrent des difficultés dans l'apprentissage de l'algorithmique et de la programmation. Il devient ainsi impératif de réfléchir sur des stratégies pour réduire les taux d'abandon et améliorer l'apprentissage.

1.2 Problématique

Nous nous sommes ainsi intéressés aux motifs d'abandons et d'échecs dans le cadre de l'initiation à la programmation. Plusieurs facteurs ressortent de la littérature mais l'un des éléments les plus mis en cause est la faiblesse des compétences de résolution de problèmes.

Les compétences de résolution de problèmes et les compétences métacognitives sont indexées à plusieurs titres :

1. la faiblesse de ces compétences apparaît comme une des causes principales d'échec et d'abandon ;
2. ces compétences sont faibles chez les apprenants ayant échoué mais aussi chez une partie des apprenants ayant validé leur cours d'initiation ;
3. ces compétences ne sont pas explicitement incluses dans beaucoup de curricula ;
4. beaucoup de formateurs se focalisent sur la syntaxe des langages enseignés et abordent assez faiblement les démarches de résolution de problèmes.

Développer les compétences de résolution de problèmes durant l'initiation à la programmation est ainsi un vrai défi, particulièrement dans un cours d'introduction avec un temps limité et des effectifs élevés comme c'est le cas aujourd'hui dans beaucoup d'institutions. Les Environnements Informatiques pour l'Apprentissage Humain (EIAH) restent une bonne alternative pour développer cette compétence en accompagnement de l'apprenant entre outils autonomes sur ordinateur ou tablette, la Formation à Distance (FAD) et les Cours en ligne ouverts et massifs (MOOC).

Nous avons rencontré dans la littérature plusieurs stratégies qui ont été évalués et qui montrent un impact positif sur ces compétences.

Le **guidage de l'apprenant** durant le processus et l'explicitation des concepts est une des méthodes qui montrent un impact très positif sur ces compétences et qui a été évalué en enseignement présentiel.

Les **feedbacks sur la sémantique** du code source ont également montré un impact très positif sur ces compétences aussi bien en enseignement présentiel qu'avec un outil. Nous avons rencontrés peu d'outils qui automatisent le processus d'analyse et de comparaison sémantique de codes sources dans ce cadre. Ces outils peuvent être décomposés en ceux basés sur une approche dynamique dans laquelle le code source est exécuté et ceux basés sur une approche statique sans exécution du code. Dans cette dernière catégorie, nous

avons des outils qui adoptent une approche statique manuelle dans laquelle le code est analysé par un humain et ceux qui adoptent une approche statique automatique dans laquelle le code est analysé par une routine pour générer un indicateur.

Les approches dynamiques s'appuient sur des tests unitaires qui nécessitent beaucoup d'efforts de la part des formateurs. Il en est de même pour les méthodes statiques manuelles qui impliquent une analyse à la main du code. Les méthodes statiques automatiques ont l'avantage de nécessiter moins d'efforts de la part du formateur et d'avoir l'immédiateté du feedback, mais nous en avons rencontrés très peu. Celles que nous avons rencontrés s'appuient sur l' apprentissage automatique (machine learning) avec des taux de précision relativement faibles.

Nous pouvons également citer le **travail collaboratif** comme approche ayant eu un impact très positif sur les compétences de résolution de problèmes durant l'initiation à la programmation. Des méthodes telles que l'apprentissage par les pairs (peer instruction) et l'apprentissage par découverte guidée (Process Oriented Guided Inquiry Learning - POGIL) ont été évalué dans ce cadre.

Nous nous sommes ainsi intéressé à l'intégration du guidage de l'apprenant dans un outil et à l'automatisation du processus d'analyse et de comparaison sémantique de codes sources.

Nous proposons ainsi, dans cette thèse, une approche pour venir en appui à l'apprenant dans le développement des compétences de résolution de problèmes durant l'apprentissage de la programmation informatique avec un outil numérique. Cet outil, IDE4SCAPSS (IDE for SCAffolding Problem Solving Skills), peut être utilisé de manière autonome ou en accompagnement de l'apprenant dans un enseignement classique en présentiel ou dans une formation à distance.

Notre approche est basée sur deux stratégies :

1. le guidage de l'apprenant dans le processus de résolution de problèmes et l'explicitation : l'apprenant est ainsi amené à suivre une démarche simple de résolution de problèmes à travers ses différentes étapes pour développer ces compétences.

2. l'usage d'une approche automatique d'analyse et de comparaison sémantique de codes sources. L'approche que nous proposons est basée sur les mathématiques symboliques et le calcul formel. Elle peut servir de base à un système automatique de feedbacks sémantiques.

1.3 Plan de la thèse

Le mémoire est divisé en quatre chapitres ; le **premier chapitre** étant l'introduction.

Le **deuxième chapitre**, composé de trois sections, est un état de l'art sur le développement des compétences de résolution de problèmes durant l'initiation à l'algorithmique et à la programmation informatique. Nous reviendrons dans ce chapitre sur l'initiation à l'algorithmique et à la programmation informatique dans différents aspects. Nous aborderons ensuite les compétences de résolution de problèmes particulièrement dans le cadre de la programmation avec les méthodes d'analyse et de conception de programmes. Nous aborderons enfin les stratégies utilisées pour développer les compétences de résolution de problèmes durant l'initiation à l'algorithmique et à la programmation et les éléments ayant eu un impact positif sur cette compétence, aussi bien en enseignement présentiel qu'avec un outil numérique.

Le **troisième chapitre** est consacré aux contributions de cette thèse. Nous présenterons d'abord le cadre théorique et l'approche de guidage de l'apprenant proposée ; nous introduirons ici les différentes étapes du processus de résolution de problèmes et les aspects relatifs à l'explicitation. Nous présenterons ensuite l'approche automatique d'analyse et de comparaison sémantique de codes sources proposée. La comparaison sémantique de codes sources est la partie sur laquelle nous avons le plus travaillé. Nous introduirons en premier lieu le formalisme proposé avant d'aborder les processus de calcul et les algorithmes. Nous terminerons par une section sur l'implémentation et l'expérimentation de l'outil.

Le **quatrième chapitre** est la conclusion dans laquelle nous ferons une synthèse de nos différentes contributions avant de dégager les limites et les perspectives de recherche de ce travail.

Chapitre 2

État de l'art

2.1 Introduction

L'initiation à l'algorithmique et à la programmation informatique est un impératif pour tout apprenant dans un cursus universitaire scientifique / informatique. Elle s'est généralisée dans beaucoup de formations universitaires non scientifiques et s'est bien développée dans l'enseignement primaire, moyen et secondaire. Elle peut être introduite comme matière seule ou dans le cadre d'un cours d'apprentissage de la pensée informatique. Cela s'explique, d'une part, par le fait qu'elle permet de développer des compétences de résolution de problèmes qu'il est possible aujourd'hui de transférer à tous les domaines, et d'autre part, par le fait que l'outil informatique est présent partout, impliquant son usage dans toutes les situations, de manière à ce que beaucoup de problèmes sont, de nos jours, d'ordre computationnel.

Toutefois, la faiblesse des compétences de résolution de problèmes des apprenants à la fin du cours d'initiation et l'absence de sa prise en compte dans les enseignements sont considérées comme une des causes d'échec et d'abandon. Ces compétences ne sont pas intégrées explicitement dans beaucoup de curricula et les formateurs se focalisent assez souvent sur la syntaxe des langages enseignés.

Le développement des compétences de résolution de problèmes dans le cadre de l'initiation à l'algorithmique et à la programmation informatique est un sujet qui intéresse aujourd'hui de nombreux chercheurs. Cela s'explique par le fait que ces compétences sont faibles chez beaucoup d'apprenants et, comme dit plus haut, sont causes d'abandon et d'échec. Elles peuvent toutefois être développées durant le processus d'apprentissage de la programmation. Nous avons ainsi rencontré plusieurs initiatives en enseignement présentiel mais aussi dans le cadre des outils numériques qui essaient de venir en appui à l'apprenant dans le développement de ces compétences durant l'initiation.

Dans cet état de l'art, nous aborderons l'initiation à l'algorithmique et à la programmation dans la section 2.2 ; nous présenterons ensuite la résolution de problèmes algorithmiques dans la section 2.3 ; nous finirons par les stratégies de développement des compétences de résolution de problèmes durant l'apprentissage de l'algorithmique et de la programmation aussi bien en présentiel que sur des outils numériques en section 2.4 avant de conclure.

2.2 L'initiation à l'algorithmique et à la programmation

2.2.1 Introduction

Nous nous intéressons dans nos travaux à l'initiation à l'algorithmique et à la programmation dans les formations à vocation informatique dans lesquelles cela reste une matière fondamentale mais avec des taux d'abandon et d'échec élevés. L'algorithmique et la programmation peuvent être abordées en parallèle dans une seule matière ou comme deux matières indépendantes ou l'une à la suite de l'autre. Vu l'impact de l'enseignement de la programmation sur les compétences de résolution de problèmes, elle est aujourd'hui enseignée à plusieurs niveaux du cycle scolaire et dans toutes les spécialités à travers les cours d'introduction à la pensée informatique (Computational thinking). Selon le niveau et la spécialité, les apprenants sont initiés à la programmation informatique par des méthodes et des outils différents.

Nous aborderons dans la section 2.2.2 l'historique de l'initiation à l'algorithmique et à la programmation et les curricula. Nous présenterons ensuite dans la section 2.2.3 le choix du premier langage de programmation. Nous reviendrons dans la section 2.2.4 sur l'abandon et les difficultés rencontrées par les apprenants durant l'initiation. Nous présenterons, dans la section 2.2.5, le lien entre la programmation et la résolution de problèmes.

Nous ne pouvons terminer cette partie sans parler de l'initiation à la pensée informatique dans la section 2.2.6. Dans cette section, nous ferons un focus sur l'apprentissage de la programmation dans les formations non scientifiques et dans l'enseignement primaire, moyen et secondaire.

2.2.2 Les curricula

Au tout début de l'informatique, les programmes étaient conçus par des spécialistes du matériel et l'enseignement de la programmation n'était pas encore vraiment né. Avec l'apparition des langages de haut niveau dans les années 60 (FORTRAN, BASIC, ...), le logiciel se détachant du matériel, le métier de programmeur apparaît avec toutefois des pratiques de programmation empiriques et désordonnées. L'enseignement avait pour objectif de faire apprendre un langage et cela était souvent une question d'expérience et d'art. Les programmeurs étaient alors fortement marqués par la machine avec des variations individuelles fortes dans la pratique. Le coût du logiciel ne cessait d'augmenter et apparaissait alors la nécessité de communiquer et de travailler en équipe mais aussi de formaliser les choses [3].

La programmation structurée se formalise et il est conçu que tout programme peut être exprimé de façon hiérarchique et modulaire avec des primitives standardisées.

L'informatique est devenue un domaine académique en 1962 avec la création du premier département d'informatique à l'université de Purdue aux États-Unis. L'enseignement de l'informatique a connu depuis lors un développement fulgurant, une évolution particulière et une certaine complexité.

Dès l'acceptation de l'informatique comme domaine académique, la question du curriculum s'est posée. C'est ainsi que l'"Association for Computing Machinery" (ACM) propose dès 1968 une première recommandation de curriculum pour la formation universitaire en informatique et pour le cycle d'ingénieur en informatique [4]. Dans ce curriculum, figurent les thèmes de l'informatique, 22 **matières** avec leurs prérequis, leur description et une bibliographie et la proposition du langage SNOBOL pour l'initiation.

Ce curriculum est mis à jour en 1978 [5] puis en 1984 [6]. Il apparaît une évolution sur la proposition des langages d'initiation vers Pascal, PL/1 et Ada et alternativement FORTRAN et BASIC et la mise en exergue de l'introduction des méthodes et de la résolution de problèmes [7, 8].

En 1991, en coopération avec l'"Institute of Electrical and Electronics Engineers" (IEEE), ACM publie une nouvelle recommandation de curriculum qui n'est plus basée sur des matières mais sur des **unités de connaissances** correspondant à des thèmes qui devront être assimilés à une certaine étape du cycle d'apprentissage [9]. Cela permet aux institutions d'avoir la flexibilité de constituer leur curriculum selon leurs spécificités. Un curriculum dépend effectivement de plusieurs facteurs : les objectifs de l'institution, sa capacité, son infrastructure et ses ressources, le niveau des apprenants, les critères d'accréditation, la professionnalisation pour une entrée dans la vie active ou la préparation à des études longues, ...

Une nouvelle recommandation est proposée en 2001 avec l'introduction des approches objets [10]. En 2008, une version mettant en avant l'importance d'aborder plusieurs langages dans le cursus est sortie [11]. L'étudiant doit découvrir l'intérêt de connaître plusieurs langages car le choix du paradigme de programmation influence fortement la manière de penser un problème et sa solution. Il y'a eu une dernière révision en 2013 [12] et enfin en 2020 [13].

Concernant particulièrement le cours d'initiation à l'algorithmique et à la programmation, il est présent dans les recommandations de curricula de l'ACM depuis la première version de 1968. Il trouve le nom, Computer Science 1 (CS1), qui lui est donné aujourd'hui dans le monde anglophone dans la version de 1978. Il est proposé en 2-2-3, 2 heures de cours magistral, 2 heures de

laboratoire pour un total de 3 crédits et est considéré comme un prérequis pour le cours Computer Science 2 (CS2). S'il apparaît comme une matière avec une description totale dans les premières versions, il est introduit avec des exemples de séquencement d'unités de connaissances et de choix d'approches dans les versions plus récentes. En effet, les curricula recommandent aujourd'hui plusieurs approches pour le cours d'initiation :

- une **approche impérative** qui introduit les fondamentaux dans un paradigme de programmation impérative avec un langage de programmation procédurale ;
- une **approche objet** qui introduit en initiation l'approche objet avec un langage de programmation objet.
- une **approche fonctionnelle** qui débute l'initiation avec une approche fonctionnelle qui est vue comme étant plus proche de l'esprit mathématique avec un langage de programmation déclaratif.

Le curriculum du cours d'introduction est particulièrement important [7], mais [8] trouve dans une recherche qu'il n'y a pas d'unanimité sur la définition du cours d'initiation à la programmation CS1 ni sur le contenu. Dans [8], l'auteur précise que certains thèmes sont considérés importants quelle que soit l'approche utilisée : variables, types, expressions, structures de contrôles, structures itératives, sous-routines. Le niveau de connaissance des apprenants à la sortie du cours d'initiation est également très disparate. Beaucoup d'apprenants ont des conceptions erronées et des modèles mentaux non viables sur les concepts fondamentaux à la fin de ce cours [8].

2.2.3 Le langage d'initiation

Le choix du **langage d'initiation** a toujours été une question importante dès la prolifération des langages [14]. Cette prolifération des langages est logique et due à plusieurs facteurs :

- l'évolution historique des concepts avec
  - la révolution des langages structurés et la disparition du goto pour la boucle et la structure de contrôle ;
  - la naissance et l'évolution de l'objet ;
  - la naissance et l'évolution du web ;
- les spécificités des langages : si le langage C est connu pour la programmation système, Prolog pour le raisonnement et la logique, Pascal est plus pour la pédagogie ...
- le choix personnel du formateur qui pourrait préférer par exemple le langage C plutôt que python ou java plutôt que C++.
- ...

ACM et IEEE ne recommandent pas explicitement de langage aujourd'hui. La recommandation de curriculum encourage l'introduction d'un langage simple et pédagogique. Il revient ainsi aux responsables pédagogiques et formateurs qui s'appuient sur leurs recommandations de faire un choix.

Historiquement, FORTRAN, comme premier langage de haut niveau, a été naturellement le premier choix dans les cours d'initiation à la naissance des formations en informatique. Il a été assez naturellement suivi par COBOL. En 1964, avec la naissance de BASIC, le choix s'étend, et en 1972, on rencontre aussi ALGOL, LISP, PL/1 et les autres langages structurés.

Il y'a un consensus à tendre vers le langage PASCAL à la fin des années 70 pour plusieurs raisons :

— Pascal est un langage presqu'écrit pour l'enseignement de la programmation ;
— la baisse de l'utilisation des sauts avec GOTO avec la lettre de Dijkstra [15] contre son usage excessif ;
— l'usage d'Environnements de Développements Intégrés (EDI) conviviaux pour le Pascal et de compilateurs ;
— la prolifération des PC.

Le déclin de Pascal commence à la fin des années 80/début 90 avec la naissance des langages objets et le fait qu'il ne soit pas vu comme un langage de l'entreprise/professionnel.

En 1995/96, Pascal est utilisé en initiation par 36 % de formateurs, suivi de C++ à 32 % avec 22 % qui sont prêts à basculer entre C/C++, java et Ada.

Aujourd'hui, Java, python et le langage C/C++ sont les langages les plus utilisés dans les cours d'initiation pour des raisons différentes [16, 17, 18, 13] :

— Pour python : l'aspect pédagogique, le coût et l'indépendance à la plateforme ;
— pour java : l'aspect objet, l'acceptation industrielle et l'indépendance à la plateforme.
— pour le langage C/C++ : l'acceptation industrielle , les performances et l'aspect système.

En synthèse, les langages basés sur BASIC était le choix il y'a 40 ans, Turbo Pascal était le consensus il y'a 30 ans mais depuis 20 ans trois langages se bousculent : C/C++, java et python avec pour ligne de défense l'objet, la pédagogie et l'acceptation industrielle [13, 18].

Les états n'étant pas aussi au même niveau d'exigences et de disponibilité de ressources, certaines recherches font ressortir des différences d'usage entre pays.

	C	C++	Java	Python	Pascal	Autres
États-Unis	4 - 5 %	19 -20 %	41 -49 %	24 - 27 %	7,74 %	
Europe	45.7 %	15 %	8.3 %	5.6 %	7.7 %	-
Portugal	48 %	-	22 %	-	-	-
Angleterre	23,6 %		46 %	13 %	-	-

TABLE 2.1 – Langages d'initiation par régions/états

Le tableau 2.1 donne les résultats de recherche de [16, 17] pour les États-Unis, de [19] pour l'Europe et de [13] pour le Portugal et l'Angleterre.

Au Sénégal, nos investigations montrent que 62,5% des institutions universitaires publiques utilisent le langage C pour l'initiation à l'algorithmique et à la programmation et 25% le langage Pascal :

- langage C : Université Cheikh Anta Diop de Dakar, Université Gaston Berger de Saint-Louis, Université Iba Der Thiam de Thies, Université du Sine-Saloum El-hadj Ibrahima Niass de Kaolack, École Superieure Polytechnique de Dakar ;
- Pascal : Université Alioune DIOP de Bambey, Université Assane Seck de Ziguinchor ;
- langage C et python : Université Virtuelle du Sénégal.

De nos jours, à quelques exceptions près, le monde académique suit le monde professionnel. Le changement le plus remarquable étant la montée en puissance de Python au détriment de Java et de la famille C [13].

Le choix du langage d'initiation a ainsi toujours été une question cruciale pour les formateurs. Ce choix est de plus en plus complexe, et continuera de l'être avec l'évolution trop rapide du domaine [13, 18]. Les motifs de choix du langage d'initiation qui ressortent de la recherche de [16] sont listés dans le tableau 2.2.

TABLE 2.2 – Motifs de choix du langage d'initiation à la programmation

Fonctionnalités offertes par le langage	26,19%
Facilité d'apprentissage	18,81%
Opportunités de travail pour les étudiants	14,76%
Popularité au niveau académique	13,10%
Tradition institutionnelle	8,57%
Choix du conseil pédagogique	5,95%
Disponibilité des enseignants et restrictions horaires	5%.

En synthèse, plusieurs critères ressortent de la littérature dans le cadre

du choix du premier langage [20] : coût, disponibilité d'une version étudiante/académique, acceptation industrielle, acceptation académique, dépendance aux architectures et systèmes, type de licence : libre / propriétaire, environnement de développement intégré, débogueur, sécurité du code, adaptation au web, paradigme et prise en compte de l'objet, communauté.

Les deux points les plus mis en avant sont l'aspect pédagogique du langage et l'adaptation au monde professionnel. Le paradigme est aussi jugé important sauf si le formateur s'aventure à utiliser un langage multi-paradigmes au risque de ne laisser l'apprenant distinguer clairement aucun paradigme.

Pour Dijkstra, le choix du premier langage est important parce qu'il détermine la pensée et l'expression de l'apprenant. Il existe beaucoup de propositions sur les langages professionnels et des pseudo langages mais jamais il n'y a eu de consensus. Même si la discussion sur l'usage d'un langage pédagogique a toujours lieu, ces langages ont souvent été abandonnés pour les langages professionnels, Pascal étant le seul ayant un peu survécu [14]. Le choix n'est donc pas facile entre simplicité, expressivité, disponibilité (compilateur, éditeur,...) et pédagogie.

Le débat sur la guerre des langages a toujours été et est encore d'actualité. Le plus important étant de faire des choix qui permettent aux apprenants de développer une pensée critique et de bonnes compétences de résolution de problèmes et d'avoir la capacité dans le futur de choisir le langage qui est en phase avec leurs besoins [13, 18].

Un aspect qu'il est important de citer ici est le choix de l'**environnement de développement** qui est utilisé pour l'initiation. Nous rencontrons des puristes qui sont pour l'usage de la ligne de commande avec un compilateur comme GCC ou JAVAC en couplage avec un éditeur de texte simple. Nous avons également des formateurs qui s'appuient sur des outils industriels populaires comme Eclipse. Entre les deux extrêmes, certains formateurs s'appuient sur des outils qui ont été développés pour l'apprentissage et qui excluent de l'environnement toute fonctionnalité jugée inutile pour un débutant. Cela s'explique par le fait que l'usage des environnements de développement professionnels dans la formation contraint enseignants et apprenants à se focaliser sur les langages et outils et non sur les aspects pédagogiques et sur la compétence de résolution de problèmes [21].

2.2.4 L'abandon et les causes d'abandon

La programmation et son apprentissage sont vus comme une activité difficile. La recherche s'est toujours intéressée à l'**abandon** perçu comme élevé et à ses causes.

Dans une étude transversale de 2007, [22] fait ressortir un taux de réussite de près de 67%, les 33% regroupant les échecs et les abandons. En 2014, une étude plus large de [23] confirme les résultats de [22]. En 2019, [22], dans une réplication de sa recherche, trouve un taux d'abandon et d'échec de 28% [24].

Ces études font ressortir que ces taux d'échec sont plus faibles dans les classes à petit **effectif** (plus petit que 30). Les recherches de [23] ne font pas ressortir de différences fortes sur les taux d'abandon et d'échec par rapport aux **langages** (figure 2.1 [23]).

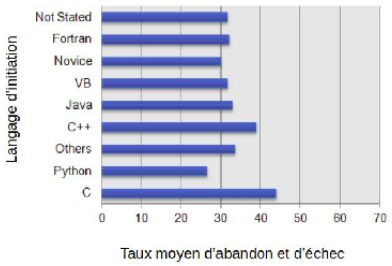

FIGURE 2.1 – Taux d'échecs par langages

La moyenne reste certes à 33 % mais il existe de grandes disparités entre **états** (figure 2.2 [23]).

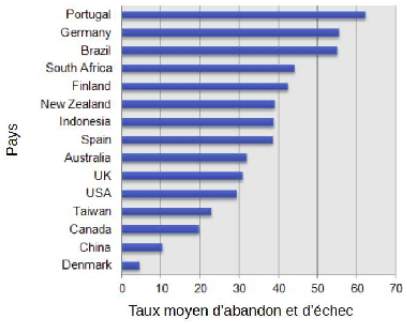

FIGURE 2.2 – Taux d'échecs par pays

Le taux d'abandon et d'échec n'est pas considéré comme alarmant pour [24] s'il est comparé à certaines matières de mathématiques. Toutefois, la croyance sur le taux d'abandon élevé et la difficulté du domaine est retenue du fait que le taux d'enrôlement a chuté à certaines périodes. Cela s'explique également par la perception des apprenants qui considèrent que la programmation est difficile.

Les **difficultés** rencontrées durant l'apprentissage de l'algorithmique et de la programmation sont nombreuses. Les défis aujourd'hui sont sur les aspects suivants [25, 26] :

- relever le niveau sur la compétence de résolution de problèmes et les compétences méta-cognitives ;
- améliorer les compétences sur la compréhension et la formulation du problème ;
- améliorer la compétence sur l'expression (choix de la bonne structure de données, choix de la bonne structure de contrôle ou itérative) et sur l'évaluation de la solution (débogage) ;
- diversifier les stratégies d'apprentissage : apprentissage par la pratique, apprentissage par les exemples, apprentissage par problème, codage en direct, apprentissage basé sur les traces, apprentissage basé sur le jeu, apprentissage par équipe et collaboratif ;
- prendre en compte l'hétérogénéité des classes et développer la communication enseignant-apprenant (feedback) ;
- adopter un paradigme, un langage de programmation et un séquencement adéquat par rapport au contexte.

L'aspect qui présente le plus grand défi aujourd'hui est le développement des compétences de résolution de problèmes. Leur faiblesse est vue comme une des causes d'abandon et la littérature montre même qu'une partie des étudiants ayant validé leur cours d'initiation sont en deçà du niveau attendu d'eux. Leur niveau reste faible sur les connaissances factuelles sur la programmation et ils sont aussi jugés inaptes à résoudre un problème tel qu'on l'attend d'eux. En synthèse, les apprenants ont aujourd'hui beaucoup plus de difficultés avec la résolution de problèmes qu'avec la syntaxe des langages [25, 26].

2.2.5 La programmation et la résolution de problèmes

La mise en place d'un programme implique de comprendre le problème, et d'utiliser une méthodologie pour proposer une solution. Il est certes nécessaire pour le futur informaticien de connaître les langages et leurs paradigmes pour pouvoir utiliser proprement les notions de programmation, les structures de données adéquates et routines mais il est aussi nécessaire pour lui de connaître les stratégies pour résoudre une classe de problèmes.

Pour [27], la programmation est une activité de conception, et en tant que telle, le résultat du processus de programmation n'est pas un programme en soi, mais plutôt un artefact qui remplit une fonction souhaitée. Les artefacts conduisent naturellement au concept de mécanisme : un mécanisme, qu'il

s'agisse d'un transistor, d'un boulon, d'un engrenage ou d'une instruction verbale, spécifie une chaîne d'actions qui, lorsqu'elle est mise en marche, produit un effet désiré. La programmation n'est donc pas une simple activité de codage : elle fait fortement appel à des compétences de résolution de problèmes et une forte méthodologie et participe ainsi au développement des compétences de résolution de problèmes de manière générale [28]. Plus particulièrement, de nos jours, avec la présence de la technologie, beaucoup de problèmes sont d'ordre computationnel et nécessitent de la personne un minimum de connaissances sur le fonctionnement de la technologie de manière générale et plus particulièrement sur le fonctionnement d'un programme. Cela explique assez aisément l'évolution croissante de l'introduction à la pensée informatique dans les formations [29, 30] .

La programmation fait fortement appel à des concepts jugés utiles dans la résolution de problèmes et permet ainsi de les développer :

1. la **classification** qui fait référence au fait de distribuer par classes, par catégories en recherchant dans les caractéristiques d'un problème les similarités avec un autre problème déjà résolu.

   Elle est importante dans l'identification, la catégorisation et la description des problèmes mais aussi dans la reconnaissance de motifs.

2. l'**abstraction à différents niveaux** qui fait référence au processus de généralisation partant des spécificités des systèmes vers l'information commune. Elle permet d'appréhender un problème et sa solution à différents niveaux. Cela offre un certain recul et une meilleure vue d'ensemble du problème et de sa solution.

3. la **systématisation algorithmique** relative au fait de concevoir les actions à accomplir sous forme d'une série de tâches (concept d'algorithme). Elle amène à expliquer les processus de manière formelle et à réfléchir aux tâches à accomplir sous forme d'une série d'étapes en s'appuyant sur le séquencement, l'alternative et l'itération.

4. la **décomposition** relative à la division d'une tâche en sous-tâches et d'un système en sous-systèmes. Cela rend le système plus facilement abordable par voie algorithmique. Cela facilite la résolution d'un problème complexe en le décomposant en plusieurs problèmes simples.

5. la **généralisation** qui consiste entre autres à réinvestir les solutions d'un problème pour aborder des problèmes similaires. On réalise ainsi que la solution à un problème peut servir à résoudre tout un éventail de problèmes similaires et à faire un transfert à d'autres domaines.

6. le **diagnostic** et la **remédiation** qui font référence à l'évaluation des solutions par l'identification et la correction des erreurs.

7. l'**optimisation** relative à l'amélioration des performances d'une solution à un problème.

Particulièrement dans le cadre de l'apprentissage de la programmation, [28] montre que certaines compétences se développent chez l'apprenant : des compétences mentales de haut niveau telles que la résolution de problèmes mathématiques et la pensée critique, des compétences individuelles personnelles telles que les compétences sociales particulièrement les compétences de collaboration et des compétences méta-cognitives telles que la compétence d'autorégulation. Son impact sur les compétences pédagogiques d'apprentissage de manière générale est aussi jugé élevé.

Nous nous sommes limités ici à montrer le lien qui existe entre la programmation informatique et la résolution de problèmes. Nous reviendrons dans la suite sur les compétences de résolution de problèmes de même que sur le développement de ces compétences durant l'apprentissage de la programmation.

2.2.6 L'initiation à la pensée informatique

Nous ne pouvons finir cette section sur l'initiation à la programmation sans aborder la **pensée informatique** (computational thinking). Son apprentissage, qui implique l'apprentissage de la programmation, s'est largement développé dans l'enseignement primaire, moyen et secondaire et dans les filières non scientifiques.

La pensée informatique, vulgarisée par Wing [31, 32], est définie comme "le processus réflexif impliqué dans la formulation de problèmes et de leurs solutions de manière que leur résolution puisse être effectuée par un agent de traitement de l'information" [32]. La recherche retient aujourd'hui que l'apprenant adulte ou enfant utilise la pensée informatique durant les séances de programmation et qu'il fait ainsi appelle à des concepts tels que l'abstraction à différents niveaux, la classification, l'itération, la généralisation, la décomposition, la recomposition, le diagnostic, la remédiation et beaucoup d'autres concepts pour la résolution de problèmes. Cette compétence est de nos jours indispensable pour les "digital natives". L'apprenant développe ainsi un esprit de créativité et des habiletés à bien analyser, modéliser et collaborer.

Wing trouve que la pensée informatique est "un ensemble d'attitudes et de connaissances universellement applicables, que nous gagnerions toutes et tous à apprendre et à maîtriser, et que nous devrions transmettre à nos enfants". Elle aide à appréhender les problèmes du monde réel avec une approche basée sur les concepts employés en programmation informatique. Par un transfert de compétences, l'individu formé à la programmation autant

que les programmeurs et développeurs s'appuiera sur les concepts présentés plus haut : l'abstraction à différents niveaux, la classification, l'itération, la généralisation, la décomposition, ...

L'apprentissage de la programmation dans l'**enseignement primaire, moyen et secondaire** date des années 60 avec l'introduction de logo pour l'apprentissage des mathématiques [33, 34, 35]. Avec logo, l'apprenant déplace une tortue à l'aide de commandes. Même si Papert a défendu l'idée qu'elle développe chez l'enfant des compétences de logique utiles à la résolution de problèmes, beaucoup de recherches n'ont pas trouvé à cette époque-là ces mêmes conclusions.

Ces dernières années avec le développement des interfaces et la démultiplication des langages de programmation visuelle, cette question retrouve un intérêt [36].

Dans les **formations universitaires non scientifiques**, la programmation est enseignée parce qu'elle développe chez l'apprenant des compétences de résolution de problèmes et de conception de systèmes ; Elle peut être abordée seule ou comme étant une composante de l'introduction à la pensée informatique.

C'est ainsi que certains pays et certaines institutions ont ainsi intégré l'apprentissage de la pensée informatique dans leur curricula en s'appuyant par exemple sur " 7 grandes idées " de l'informatique [36].

1. l'informatique est une activité humaine créative ;
2. l'abstraction réduit les informations et les détails pour se concentrer sur les concepts pertinents pour comprendre et résoudre les problèmes ;
3. les données et les informations facilitent la création de connaissances ;
4. les algorithmes sont des outils permettant de développer et d'exprimer des solutions à des problèmes de calcul ;
5. la programmation est un processus créatif qui produit des artefacts informatiques ;
6. les dispositifs et systèmes numériques, ainsi que les réseaux qui les interconnectent, permettent et favorisent les approches computationnelles pour résoudre les problèmes ;
7. l'informatique permet l'innovation dans d'autres domaines, notamment les sciences, les sciences sociales, les sciences humaines, les arts, la médecine, l'ingénierie et les affaires.

Du coté des **outils**, l'apprentissage de la programmation au primaire, au collège et au lycée et même quelques fois dans l'enseignement supérieur est

abordé à travers la programmation par bloc ou programmation créative. A l'école primaire, la programmation par bloc est plus utilisée tandis qu'au collège et au lycée, la programmation classique gagne du terrain. La programmation par bloc sur des environnements visuels et multimédia permet de créer de manière très simple des histoires interactives, des animations, des simulations, des jeux en assemblant des blocs.

Depuis logo, la conception de ces environnements est sous tendue par le concept de "plancher bas, plafond haut" (low floor, high ceiling) pour s'adresser au maximum de personnes : il devrait être facile pour le débutant de franchir le seuil minimal pour créer un programme (plancher bas), mais l'outil devrait être assez puissant et complet pour satisfaire les programmeurs avancés (plafond haut). Parmi les programmes qui respectent cette condition, nous pouvons citer alice, scratch, gamemaker ...

La plupart de ces outils respectent également le principe des trois niveaux de progression "usage-modification-création"(use-modify-create). Ils permettent ainsi de suivre un niveau d'apprentissage progressif qui permet de maintenir l'engagement de l'apprenant. Cela n'exclut pas l'introduction de concepts complexes tels que l'abstraction procédurale et de données, la pensée itérative et récursive, la décomposition de tâches, la pensée conditionnelle et le débogage. Beaucoup d'outils ont ainsi été développés et sont utilisés dans l'enseignement et cherchent à intégrer les aspects ludiques tout en prenant en compte les aspects émotionnels et motivationnels. Il peut toutefois arriver dans certaines formations que les outils du monde professionnel soient utilisés.

2.2.7 Conclusion

L'initiation à l'algorithmique et à la programmation est fondamentale dans les filières scientifiques, surtout celles à spécialité informatique. Elle est souvent abordée avec un langage de programmation en mode texte et des outils de développement du monde professionnel. Plusieurs défis sont à relever aujourd'hui pour cet initiation : le choix du langage de programmation et de l'environnement de développement, le séquencement de l'enseignement et le choix de l'approche, mais surtout l'intégration explicite des méthodes de résolution de problèmes pour combler la faiblesse des compétences de résolution de problèmes et/ou les développer.

Cet initiation se fait aujourd'hui à tous les niveaux du cursus scolaire et se généralise de plus en plus. Elle se fait dans les filières non scientifiques et dans le primaire, au collège et au lycée parce qu'elle permet de développer les compétences de résolution de problèmes. La programmation peut être enseignée comme matière indépendante ou dans un cours d'initiation à la

pensée informatique. Elle est enseignée avec les outils classiques ou avec de la programmation par bloc.

Il faut constater que dans les filières scientifiques les enseignants se focalisent sur la syntaxe du langage enseigné et intègrent assez faiblement les compétences de résolution de problèmes. La faiblesse de ces compétences est considérée comme une des principales causes d'abandon de même que leur non intégration explicite dans les curricula et les enseignements.

2.3 La résolution de problèmes

2.3.1 Introduction

La résolution de problèmes est généralement considérée comme l'activité la plus importante dans notre quotidien, donc la compétence la plus importante pour la vie [37].

Le champs de la résolution de problèmes est très vaste. Nous nous limiterons ici au domaine qui nous intéresse, c'est à dire la résolution de problèmes algorithmiques. Il faudra toutefois rappeler que l'impact de l'apprentissage de la programmation s'étend au delà des compétences de résolution de problèmes algorithmiques.

La résolution de problèmes algorithmiques fait référence au processus de conception d'un programme, à travers les méthodes d'analyse et de conception de programmes ou un guide de bonnes pratiques, pour avoir une solution correcte à un problème. Ces méthodes ont ensuite évolué en parallèle de la gestion de projets et des méthodes de conception d'applications de grande envergure, c'est ce qui a donné naissance à l'ingénierie logicielle. Ces méthodes ont, à une certaine période, été très utilisées, et plus tard beaucoup moins, dû à la lourdeur taxée à certaines d'entre elles pour des guides de bonnes pratiques qui s'en inspirent.

Dans cette partie, nous aborderons les compétences de résolution de problèmes dans une perspective liée à la psychologie cognitive dans la section 2.3.2. Nous reviendrons sur l'histoire des méthodes d'analyse et de conception de programmes dans la section 2.3.3. Nous traiterons ensuite des méthodes d'analyse et de conception de programmes elles-mêmes dans la section 2.3.4. Nous terminerons par la faiblesse des compétences de résolution de problèmes chez les novices dans la section 2.3.5 avant de conclure.

2.3.2 Les compétences de résolution de problèmes

Pour [38], un **problème** est vu théoriquement comme une difficulté de nature théorique ou pratique qui provoque une attitude inquisitrice chez un sujet et le conduit à l'enrichissement de ses connaissances. Il survient, selon [39], lorsqu'une personne a un objectif spécifique mais qu'elle ne sait pas comment l'atteindre. [38] rappelle que pour Josef Linhart c'est une relation interactive entre un sujet et son environnement, qui incorpore le conflit intérieur que le sujet résout en recherchant des transitions entre la condition initiale et la condition finale (objectif).

Pour [38], la réflexion d'un individu commence par la prise de **conscience**

de la **situation problématique**. Une situation problématique est définie comme étant la totalité des conditions qui déterminent la formation et les spécificités du problème. Il est ainsi retenu qu'une situation problématique a le potentiel de se transformer en un problème qui mérite une solution. Tout problème est donc lié à une situation problématique, mais toutes les situations problématiques ne se transforment pas en problèmes, car cette réalité dépend de l'individu, on parlera alors de **perceptibilité du problème**.

Ensuite viennent la **volonté de s'attaquer au problème** suivi de la **volonté de solutionner le problème** qui sont influencées par des facteurs émotionnels et motivationnels tels que l'intérêt et les convictions mais aussi les habiletés et compétences de l'individu.

Et c'est à la suite de cela que l'individu passe à la **résolution du problème**. La résolution de problèmes est un traitement cognitif visant à atteindre un objectif lorsqu'aucune solution n'est évidente [40, 41, 42].

Une définition assez complète de cette compétence ressort du cadre théorique qui sous-tend l'évaluation informatique de la compétence individuelle sur la résolution de problèmes dans le cadre de l'enquête PISA 2012[1] : la **compétence de résolution de problèmes** est définie comme la capacité de l'individu à utiliser les compétences cognitives pour comprendre les situations problématiques et les résoudre dans le cas où aucune solution évidente ne se présente. Pour [43], il s'agit d'une compétence de vie acquise, et chaque individu possède ses propres capacités de résolution de problèmes, apprises à des rythmes différents à travers diverses situations de la vie quotidienne.

L'individu s'appuiera sur des **stratégies de résolution de problèmes** qui peuvent être caractérisées comme un plan de la séquence d'étapes consistant en l'application de méthodes et de ressources appropriées qui conduisent à la réussite de la résolution du problème.

Les compétences de résolution de problème en **algorithmique et programmation** font référence à la capacité à analyser un problème et à concevoir et évaluer une solution. Elles s'appuient généralement sur l'usage des **méthodes d'analyse et de conception de programmes** ou un **guide de bonnes pratiques** mais aussi sur l'expérience.

Historiquement, beaucoup de méthodes ont trouvé leur origine en intelligence artificielle et en psychologie cognitive avec les recherches sur les pratiques des experts.

La psychologie cognitive s'est très tôt intéressée à une théorie de l'expertise. La grande différence entre un novice et un expert se trouve essentiellement sur l'application d'un processus de résolution de problèmes et l'étendue

1. Conceptual framework of the problem solving PISA 2012

du répertoire de connaissance spécifique au domaine de ce dernier. S'il faut dix ans d'apprentissage et de pratique pour transformer un novice de programmation en expert, quelles sont les compétences nécessaires pour cela ? Quels sont les compétences et les processus mentaux mis en œuvre par ces experts ?

Les **experts** disposent de schémas de connaissances spécialisés et efficacement organisés ; ils organisent leurs connaissances en fonction de caractéristiques fonctionnelles tel que la nature de l'algorithme plutôt que de se concentrer sur des éléments superficiels tel que le langage. Ils utilisent des stratégies génériques de résolution de problèmes tel que diviser pour régner mais aussi des stratégies spécifiques pour décomposer et comprendre le problème. Autant que dans tout autre domaine tel que les maths ou le jeu d'échec, la force de l'expert en programmation est de reconnaître, d'utiliser et d'adapter des motifs ou schéma. Ils sont prompts à s'appuyer sur un large éventail d'exemples, de sources de connaissances et de stratégies [44]. L'expert est beaucoup plus tourné vers les stratégies et s'appuie sur son répertoire de motifs connus pour générer une solution à un problème.

Le **novice** s'appuie sur une connaissance déclarative. Il essaie de comprendre et de créer une solution à partir d'un répertoire très peu fourni en motifs. C'est ce qui explique l'importance de la connaissance des différentes classes de problèmes et l'exercice de classification de problèmes [44, 45].

2.3.3 Des méthodes de résolution de problème à l'ingénierie logicielle

Comme nous l'avons souligné plus haut, l'apprentissage de la programmation à ses débuts avait pour objectif de faire apprendre un langage. Mais assez vite, dans les années 60, la programmation structurée apparaît et il est admis que tout programme peut être exprimé de façon hiérarchique et modulaire avec des primitives standardisées. Toutefois, il faut constater que si un programme n'est pas bien conçu, il aboutit rapidement à ce qui est communément appelé un "sac de nouilles" et il devient difficilement maintenable. Il a été rapidement nécessaire de réfléchir aux processus de conception des programmes dans un souci de maintenabilité mais également de communication : c'est la naissance des méthodes d'analyse et de conception de programmes.

Plusieurs méthodes voient le jour, telles que la logique de Construction de Programmes de Warnier [46], la méthode CORIG proposée par Mallet [47], la méthode déductive de Pair [48], ou encore la méthode basée objectifs/plans [27] et l'approche des arbres programmatiques. Une des approches

qui s'est fortement distinguée et qui a énormément influencée plusieurs autres méthodes et des générateurs d'applications est la Logique de Construction de Programmes (LCP) de Warnier [46]. La LCP a laissé son empreinte sur les approches objets et le génie logiciel qui naîtront plus tard. Nous sommes encore à l'ère de la programmation impérative et ces méthodes sont bien adaptées pour la conception dans ce cadre.

Dans le cadre de l'apprentissage, ces méthodes sont certes enseignées, mais sont introduites de manière implicite dans beaucoup d'enseignements. Les méthodes les plus utilisées dans l'enseignement sont celles guidées par les données comme la LCP.

Avec l'apparition de nouvelles notions dans les années 70, telles que le multi-utilisateurs, les interfaces graphiques et les bases de données, les tentatives de création de logiciels de grande ampleur, que nous nommerons ici applications, montrent des limites. C'est la naissance, en informatique de gestion, de la gestion de projets informatiques et des méthodes d'analyse et de conception d'applications qui ont une étendue plus vaste que les méthodes d'analyse et de conception de programmes. Le métier d'analyste/programmeur évoluera petit à petit vers les métiers de spécialistes en génie logiciel, concepteurs et développeurs.

C'est ainsi que les méthodes de conception de programmes se confondent avec les méthodes de gestion de projets et les méthodes de conception de systèmes d'informations/d'applications qui sont nées dans le cadre de l'informatique de gestion. Cela correspond, dans les méthodes de conception d'applications, à l'époque des méthodes dites à cycle de vie en cascade comme MERISE. Les méthodes de gestion de projets et les méthodes d'analyse et de conception d'applications ont une étendue plus vaste que les méthodes de conception de programmes, elles prennent en compte une dimension temporelle et beaucoup d'autres aspects (systèmes, bases de données, travail en équipe, ...) pour la réussite d'un projet de mise en place d'une application à grande envergure ou d'un système d'information. Elles intègrent très souvent des outils qui prennent en compte l'aspect conception de programme, c'est dans ce cadre qu'elles sont souvent citées dans les méthodes de conception de programmes. MERISE en est un exemple.

Dès la naissance de l'objet, des méthodes de conception de programmes objets voient le jour avec l'encapsulation et l'héritage. Elles sont en réalité souvent partie intégrante d'une méthode de conception d'applications, c'est ce qui explique le fait que la méthode de développement rapide d'application (Rapid Application Development -RAD) est vue comme la première méthode de conception de programmes objet alors qu'elle a une étendue plus large

de méthode de conception d'applications. Elle est vue comme la première méthode de conception en rupture avec la cascade et qui introduit le cycle itératif et incrémental.

Les méthodes qui naîtront plus tard telles que le Processus Unifié (Unified Processus - UP) et ses dérivées et les méthodes AGILES tel que XP et SCRUM sont citées au même titre.

La complexité du logiciel, avec dans les années 70 les bases de données, les interfaces graphiques, et dans les années 90 l'internet et beaucoup d'autres innovations a ainsi ouvert la voie à l'ingénierie logicielle. Il faut toutefois savoir que le développeur est toujours appelé à avoir une bonne démarche pour l'implémentation d'un programme ou d'un module en s'appuyant sur une méthode formelle ou un guide de bonnes pratiques, il y'aura toujours des lignes de code à écrire ...

2.3.4 Les méthodes de conception de programmes

Pour [3], "une méthode peut être considérée comme un guide explicite et systématique pour la recherche et la gestion des stratégies de résolution de problèmes d'une certaine classe". Elle explicite ce qui se dégage de commun dans des pratiques efficaces de résolution et le rationalise. Elle décrit les étapes de l'activité du programmeur entre représentation des objets du problème et décomposition du traitement [49].

Les méthodes sont ainsi des outils d'aide à la résolution des difficultés de différents niveaux à partir de l'énoncé du problème jusqu'à la mise en œuvre du programme et de sa documentation. Les méthodes peuvent être spécifiques à un domaine ou être générales, concerner tout ou partie du problème. Elles s'appuient essentiellement sur les tâches suivantes :

1. abstraction et modélisation du problème ;
2. hiérarchisation et organisation séquentielle des phases de résolution ;
3. mise en œuvre des solutions ;
4. évaluation des solutions et optimisation.

Les méthodes proposent des systèmes de représentation, avec souvent une composante graphique et aussi un langage avec un formalisme bien défini et s'accompagnent depuis assez longtemps d'outils d'aides (environnements de conception) qui sont aujourd'hui capables de générer du code et de la documentation. Beaucoup de méthodes sont indépendantes des langages de programmation et s'appuient sur trois idées clés à la base [3] :

1. la **structuration** qui fait référence aux trois structures de base : sequence, itération, condition ;

2. la **modularité** qui fait référence à la décomposition en module indépendants, réutilisables et éventuellement emboîtés ;
3. la **hiérarchisation** qui fait référence à la décomposition du problème en sous problèmes.

Dans les méthodes abordées dans l'enseignement, une forte importance est accordée au guidage par la structure des données (données d'entrées, résultats). Cela permet de ressortir clairement la relation entre données et traitements. Il est communément retenu que pour résoudre un problème il faut :

1. définir les données souhaitées en sortie du programme ;
2. déterminer les données d'entrée nécessaires ;
3. définir la structure du programme ;
4. écrire les instructions du programme ;
5. évaluer le programme

Les méthodes d'analyse et de conception de programmes ont été à une époque très fortement utilisées par les programmeurs et étaient intégrées dans certains enseignements. Elles le sont moins aujourd'hui laissant la place à des guides de bonnes pratiques. Nous allons présenter dans la suite deux méthodes d'analyse et de conception de programmes, certes anciennes, mais qui méritent d'être exposés pour plusieurs raisons :

— elles sont à la base de beaucoup de méthodes formelles utilisées aussi bien dans une approche de programmation procédurale que dans une approche objet.
— Elles ont également inspiré beaucoup de guides de bonnes pratiques
— ...et enfin, elles ont marqué beaucoup de générations de programmeurs.

Nous aborderons ainsi dans la suite de cette section la Logique de Construction de Programmes (LCP) de Warnier [46] et ensuite la méthode basée objectifs/plans [27].

Nous ferons une synthèse de ce qui ressort de commun dans la plupart des méthodes. Cela est très bien présenté dans le cadre proposé par [50] pour l'évaluation des apprenants sur la résolution de problèmes à la fin du cours d'introduction à la programmation.

2.3.4.1 La Logique de Construction de Programmes

L'objectif de la Logique de Construction de Programmes (LCP) est de concevoir l'organigramme de résolution du problème posé : dans une première

phase un organigramme général de la structure du programme et dans une deuxième phase un organigramme détaillé avec les instructions.

La LCP s'appuie sur la **théorie des ensembles** : un programme est vu comme un ensemble de données destinées à exécuter des fonctions. Toute collection de données constitue un ensemble au sens mathématique du terme. Tout ensemble doit être rigoureusement défini et en compréhension, de même que les relations entre ensembles.

C'est une **méthode hiérarchique** en **six étapes** guidée par les données. Abordons le par l'exemple : soit le problème du calcul des solutions d'un ensemble d'équations du second degré de paramètres a, b et c qui s'arrête lorsque a est égal à 0 :

1. la **première étape** consiste à déterminer la structure du résultat appelé **Fichier Logique de Sorties** (FLS) (figure 2.3) qui est considéré comme un **ensemble** en s'appuyant sur deux structures :
   — la **structure répétitive** pour représenter une liste d'information, c'est le cas de la liste d'équations ;
   — la **structure alternative** pour représenter des informations exclusives, c'est le cas du résultat d'une équation.

   L'**accolade** est utilisée pour indiquer une décomposition de données. La règle de décomposition est la suivante : on subdivise un ensemble de données s'il comprend des sous ensembles qui peuvent s'y trouver présents un nombre de fois différent de 1. La démarche hiérarchique fait apparaître la correspondance entre les différents sous ensembles.

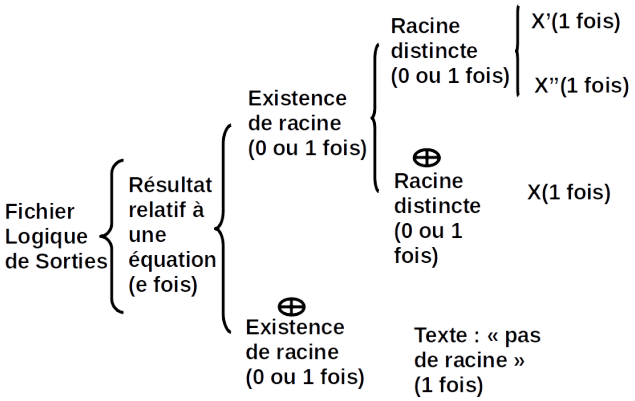

FIGURE 2.3 – Fichier Logique de Sorties

Le FLS est une liste de n résultats (niveau 1). Un résultat indique l'existence ou pas d'une racine (niveau 2). Chaque résultat correspondant à l'existence d'une racine peut être une racine double ou une

racine simple (niveau 3). En dernier, nous avons une information qui ne peut être décomposée x, x', x'' et le texte "pas de racine".

La structuration des données est indiquée par les **cardinaux** qui sont des facteurs d'occurrences : (0 ou 1 fois) pour l'alternative, (n fois) pour la répétition, (1 fois) pour toute information qui n'est plus décomposable. Le $\oplus$ est , ici, un ou exclusif. Le FLS doit s'accompagner d'un tableau qui indique les données d'entrée nécessaires (ici a, b et c), les calculs à effectuer (calcul de Δ, calcul de x, x' et x''), les conditions d'apparition des sous-ensembles conditionnés (ici signe de Δ).

2. La **deuxième étape** consiste à déterminer la structure des données d'entrée appelée **Fichier Logique d'Entrées** (FLE)(figure 2.4) avec la même démarche qu'avec le FLS sauf sur le critère de subdivision. On subdivise un ensemble de données à l'entrée s'il contient des sous-ensembles utilisés un nombre de fois différent de 1.

FIGURE 2.4 – Fichier Logique d'Entrées

3. La **troisième étape** consiste à la **validation** et à la création des **unités de traitements** : Vérifier que toutes les données de sortie sont obtenues par des données d'entrée.

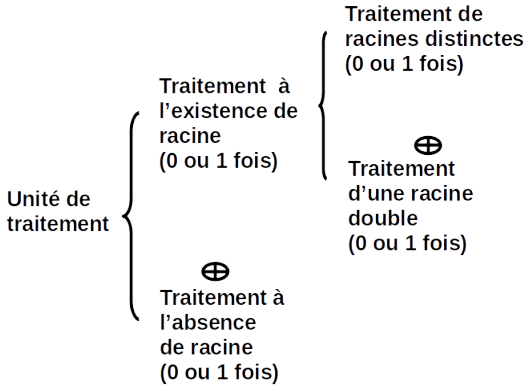

FIGURE 2.5 – Unité de traitement

Cela peut se faire en comparant les niveaux qui doivent avoir une concordance sur les données. Si tel n'est pas le cas, il y'a souvent

une nécessité de faire des traitements pour passer de l'ensemble d'un niveau du FLE à l'ensemble du même niveau du FLS.
Dans le cas de notre exemple, il y'a une concordance au premier niveau (Fichier d'entrées/Fichier de sorties). Il y'a également une concordance au niveau 2 (équation (e fois)/ résultats d'équation (e fois)). Au niveau 3 , il n'y a pas de concordance (a, b, c (1 fois) $\neq$ existence de racine(0 ou 1)) mais un traitement de l'ensemble d'entrée permet d'avoir les résultats (un traitement qui apparaît sur deux niveaux) (figure 2.5) : traitement en l'absence de racine/ traitement à l'existence de racine. le deuxième traitement (traitement à l'existence de racine), se décompose en traitement pour une racine simple/traitement pour une racine double.

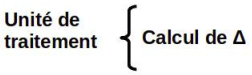

FIGURE 2.6 – Unité de traitement : calcul de Δ

Il est nécessaire que tous les critères d'identification dans la structure des résultats (FLS) apparaissent comme des données dans la structure des entrées (FLE). Autrement, il est nécessaire de créer des données intermédiaires, par une unité de traitement, à intégrer à la structure des données d'entrées. C'est le cas du calcul du déterminant qui sera une unité de traitement à intégrer à la suite du FLE (figure 2.6).

4. La **quatrième étape** est la **construction du programme** (figure 2.7) à partir du FLE auquel on a adjoint les données intermédiaires et les unités de traitement. La structure du programme est entièrement déterminée par la structure de ces éléments.

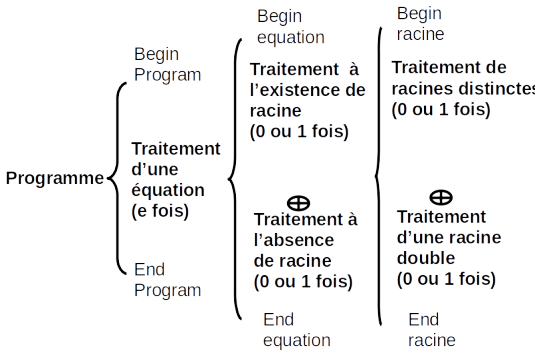

FIGURE 2.7 – Structure du programme obtenu

5. La **cinquième étape** est la construction de la structure du programme sous forme d'**ordinogramme** (figure 2.8). Il doit être isomorphe au programme ;

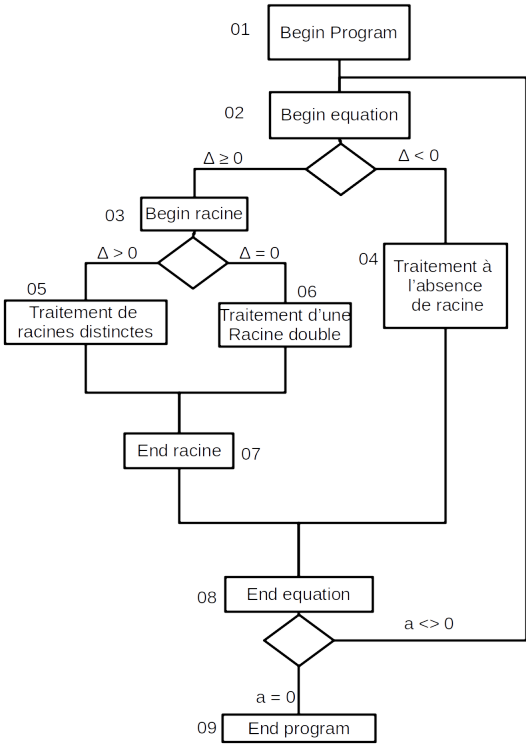

FIGURE 2.8 – Ordinogramme obtenu par la LCP

6. La **sixième étape** est la détermination des instructions pour avoir un **ordinogramme détaillé** (figure 2.9).

Les travaux de Warnier ont ainsi apporté à l'objet et à l'ingénierie logicielle des éléments clés :

1. l'analyse de la structure des données de sortie permet de construire la structure des données d'entrée d'un programme ;
2. la structure du programme correspond très exactement à la structure des données d'entrée ;
3. toute procédure peut s'exprimer avec trois structures élémentaires que sont la séquence, l'alternative et l'itération.

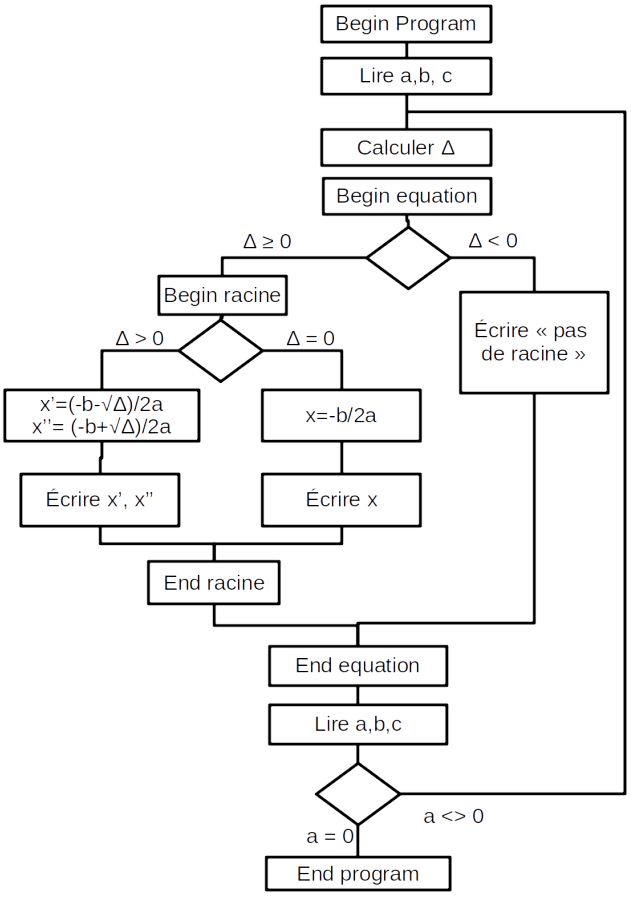

FIGURE 2.9 – Ordinogramme détaillé obtenu par la LCP

2.3.4.2 La méthode basée objectifs/plans

Dans le cadre de l'enseignement, les méthodes guidées par les données sont les plus utilisées. Les enseignants ont également tendance à demander à l'apprenant de décomposer le problème, les amenant à une démarche descendante.

Dans [27], l'auteur revient sur le fondement en psychologie cognitive de cette démarche d'analyse et de conception et les concepts de base. Cette démarche s'appuie en psychologie cognitive sur le fait que, comme nous l'avons souligné plus haut, la force de l'expert en programmation est de reconnaître, d'utiliser et d'adapter des motifs ou schéma et qu'il est prompt à s'appuyer sur un large éventail d'exemples, de sources de connaissances et de stratégies [44]. Les concepts de bases qui fondent cette démarche sont le concept d'objectif et le concept de plan.

Les **plans** sont des modèles de solutions pour des problèmes donnés et qui sont souvent bien maîtrisés par les experts. Nous donnons ici par exemple une liste non exhaustive de plans utilisés dans cette démarche par les experts : plan de la moyenne, plan de divisibilité, plan de la décomposition de nombre, plan d'initialisation, plan de protection d'exception (qui inclut le plan de la division par zéro), plan de la boucle contrôlée par compteur, plan de la boucle contrôlée par sentinelle, plan du min/max, plan de la somme, plan du décompte, plans des algorithmes de recherches, plans des algorithmes de tri, plan des arguments de la ligne de commande, plan d'usage de fichiers, plans de récursion (mono et multi-branches) ...

Nous faisons ressortir ces concepts dans une démarche de résolution de problèmes à cinq étapes :

1. L'**objectif** est à définir dans la première phase de la méthode qui correspond à la **phase de compréhension**. Il est important à cette étape de ressortir les **résultats** attendues et les **données utiles** pour y parvenir. Cet objectif sera décomposé en sous-objectifs dans la deuxième étape.

2. La **deuxième étape** est celle de **décomposition** du problème en sous-problèmes. Le programmeur pourra les décliner sous forme de **sous-objectifs**. Les compétences de décomposition sont fortement impactées par le niveau de connaissance des **plans**.

3. La troisième étape est celle de **composition des solutions au sous-problèmes**. La réussite de cette étape est entièrement déterminée par le niveau de connaissance des plans. La connaissance des motifs appelés ici plans est important pour le programmeur. Les sous-problèmes pourront correspondre à des **plans déjà connus, à adapter ou à concevoir**.

4. La **quatrième étape** qui est celle de **recomposition** s'appuie sur des stratégies. Dans [27], l'auteur identifie **quatre stratégies** :

   (a) Le **séquencement** qui consiste à adjoindre l'un des plans à l'autre. Par exemple, dans le programme de calcul de la moyenne de la figure 2.10, le plan de calcul de la moyenne produit une valeur qui est l'entrée du plan de sortie qui imprime la valeur à l'écran.

   (b) L'**emboîtement** qui consiste à faire entourer un plan par un autre. Par exemple, dans le programme de calcul de la moyenne de la figure 2.10, le plan de sortie (plan qui réalise l'objectif d'écrire la moyenne) est imbriqué dans le plan de protection de la division par zéro.

(c) La **fusion** : au moins deux plans sont à intercaler avec une partie commune.

Par exemple, pour résoudre le problème de la moyenne, le plan qui additionne les entrées et le plan qui compte le nombre d'entrées sont fusionnés (figure 2.10).

(d) L'**adaptation** : parfois, un plan déjà élaboré ne correspond pas tout à fait à ce dont on a besoin pour résoudre un problème. Il doit être modifié pour répondre aux besoins particuliers de la situation.

5. la **dernière étape** qui est l'**évaluation** s'appuie sur un aspect dynamique qui peut être la **simulation** ou la compilation et exécution en attente d'un feedback.

Il faudra remarquer que ce qui est désigné ici comme le concept de plan, est aujourd'hui appelé motif (pattern).

Prenons l'exemple du calcul de la moyenne de n variables entières tant que l'utilisateur n'a pas saisi 99999 par cette méthode des objectifs/plans. Une proposition de mécanisme est illustré dans la figure 2.10 [27].

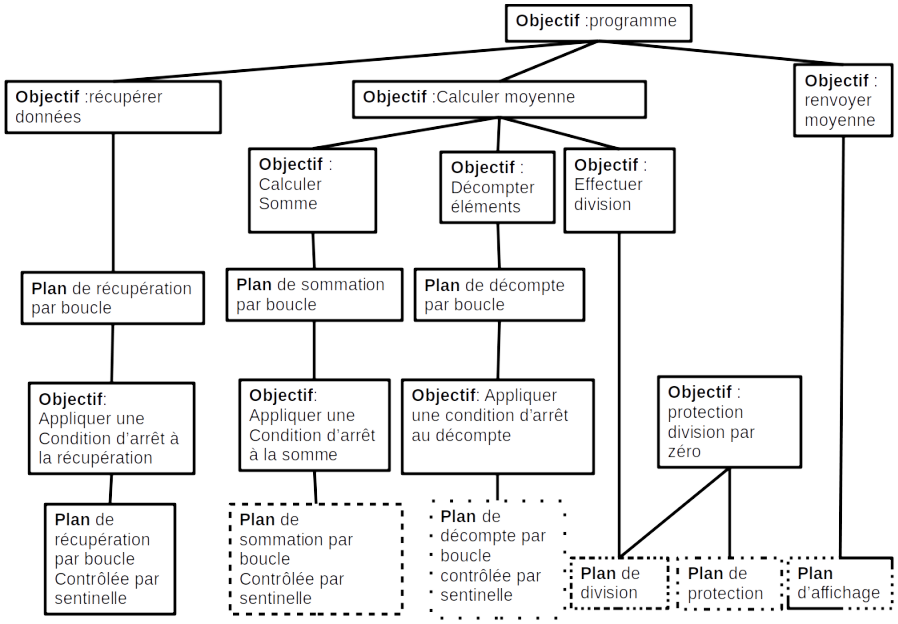

FIGURE 2.10 – Calcul de la moyenne par la méthode basée objectifs/plans

L'objectif est de calculer la moyenne de plusieurs entiers, la moyenne est la donnée de sortie et les nombres entiers sont les données d'entrée. Pour calculer la moyenne, il est nécessaire de calculer la somme des nombres et de

faire le décompte des nombres d'entiers saisis : ils sont des sous-objectifs. Il est aussi nécessaire de récupérer la séquence d'entiers. Le programmeur expert s'appuie sur quatre (4) motifs appelés ici plans : un plan de récupération des données, un plan de calcul de la somme, un plan de décompte du nombre d'éléments et enfin un plan d'arrêt avec une sentinelle. L'usage de ces concepts n'est pas lié au langage, l'expert sait les instancier quel que soit le langage. Les plans de récupération de la série d'entiers, de calcul de la somme et de décompte du nombre d'éléments vont être fusionnés et arrêtés par le plan de la sentinelle.

Ensuite l'expert passe au plan de la division qui nécessite un plan de protection de la division par zéro dans lequel il va être emboîté. Et en dernier vient le plan de sortie.

Nous avons le programme obtenu dans la figure 2.11 [27]. Les plans correspondant aux instructions sont indiqués par la forme des lignes des encadrés.

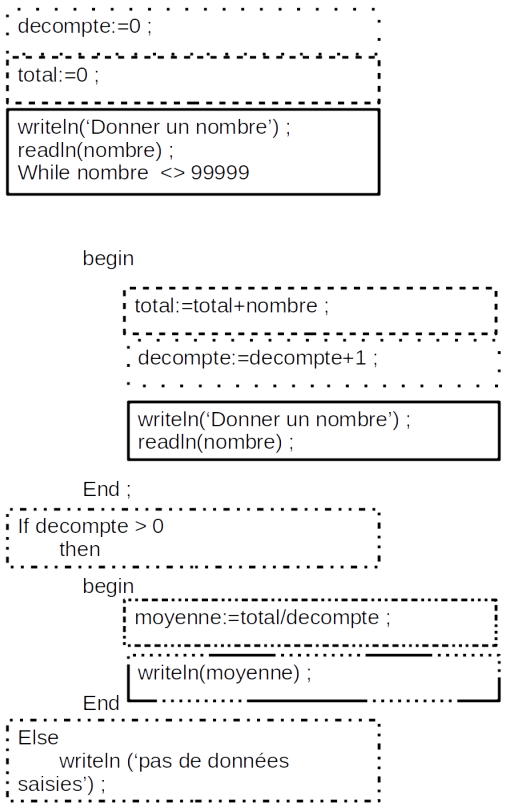

FIGURE 2.11 – Programme obtenu par la méthode basée objectifs/plans

2.3.4.3 Le commun aux méthodes

Les méthodes, comme nous l'avons dit plus haut, sont nombreuses. Dans le cadre des approches structurées, elles peuvent aujourd'hui être classées en deux familles :

1. les méthodes dites descendantes qui partent des données d'entrée vers les résultats en faisant une décomposition du problème.
2. les méthodes dites ascendantes qui partent des résultats vers les données d'entrée

Les méthodes basées sur le paradigme objet peuvent être également considérées comme étant une seule famille dans le cadre de la conception de programmes. Par contre, il existerait plusieurs typologies de classification en ingénierie logicielle telle que celle basée sur le cycle de vie.

Vu que nous nous situons ici dans le cadre de l'initiation, nous avons souhaité présenter le cadre proposé par [50] pour l'évaluation de l'apprenant sur les méthodes de conception de programmes. Les auteurs se sont intéressés à ce qui doit être évalué pour un débutant à la fin de son cours d'initiation.

Il est ainsi attendu à la fin de sa première année que l'apprenant sache résoudre un problème de manière à produire un code source compilable, un programme correct et exécutable. Il est donc attendu qu'il puisse suivre ces étapes de résolution de problèmes quel que soit le paradigme de programmation introduit. Cela reste également un objectif (éventuellement implicite) au fur et à mesure que les étudiants progressent dans leur programme, bien que le domaine d'application, ainsi que l'ampleur et la complexité des problèmes traités, changent.

Cette méthode se présente comme un processus itératif de cinq étapes :

1. **abstraction du problème/compréhension** : partant des spécifications l'apprenant doit être capable de ressortir les éléments importants de ces spécifications. Ensuite, les étudiants doivent modéliser ces éléments dans un cadre d'abstraction approprié, qui est probablement prédéterminé en fonction de l'approche utilisée (approche objet, procédurale, fonctionnelle) qui est fortement influencée par l'approche pédagogique. Il peut être attendu à cette étape que l'apprenant ressorte les entrées et sorties du programme mais aussi les objets et classes de base dans une approche objet.
2. **décomposition en sous-problèmes** : cette étape dépend de l'approche utilisée. L'apprenant aura à ressortir les liens entre classes / fonctions, selon l'approche utilisée. Pour une approche procédurale, une décomposition fonctionnelle est attendue de l'apprenant. Dans

une approche objet, il faudra finaliser la modélisation des classes et effectuer des factorisations des méthodes, spécialisation/généralisation à travers l'héritage des classes et les concepts de l'objet ...

3. **composition des solutions aux sous-problèmes** : c'est à ce stade surtout qu'il est attendu de l'apprenant qu'il ait des stratégies d'implémentation. Il décidera des structures de données et des techniques de programmation (stratégies). La granularité de la modularité doit être conforme aux standards et chaque solution de sous-problème devra être fonctionnelle.
4. **fusion des solutions aux sous-problèmes** : l'apprenant doit proposer un algorithme pour le contrôle de la fusion.
5. **évaluation de la solution et itération** : l'apprenant détermine à ce stade si les étapes ont conduit à une bonne solution pour le problème et agit en conséquence. Il peut retourner sur certaines étapes après avoir testé sa solution.

Ces étapes sont presque toujours présentes quelle que soit la méthode adoptée avec quelques spécificités pour certaines d'entre elles. C'est le cas de certaines méthodes agiles comme XP dans laquelle les tests jouent un rôle central et apparaissent dès la troisième étape.

Il faudra remarquer qu'à côté des méthodes formelles, documentées avec souvent un langage, une notation graphique et une démarche formelle, une école des bonnes pratiques a toujours existé aussi bien dans le monde professionnel que dans l'enseignement. Beaucoup de méthodes sont aujourd'hui des synthèses de bonnes pratiques aussi bien dans le cadre de l'approche objet que dans le cadre de la programmation structurée procédurale.

Dans le domaine de l'enseignement principalement, avec l'étendue des méthodes de conception d'applications, un ensemble de bonnes pratiques est utilisé pour le choix d'une méthode ascendante ou d'une méthode descendante pour la programmation structurée. La même philosophie est en vigueur pour l'introduction à la programmation avec une approche objet avec l'utilisation des diagrammes de classes pour la conception des éléments statiques d'un programme et l'utilisation d'un diagramme tel que le diagramme d'activité pour les éléments dynamiques.

2.3.5 La faiblesse des compétences de résolution de problèmes en initiation

Le cours d'introduction à l'algorithmique et à la programmation a pour objectif de former l'apprenant à l'art de la programmation. Il est attendu des apprenants de pouvoir concevoir et écrire des programmes simples pour

résoudre des problèmes simples. Les recherches de [50, 51] concluaient déjà que beaucoup d'étudiants à la fin du cours d'introduction ne savent tout simplement pas programmer.

Comme nous l'avons dit plus haut, la faiblesse des compétences de résolution de problèmes est une des causes d'abandon et d'échec dans l'initiation à la programmation. Plusieurs recherches montrent que même pour les étudiants ayant réussi, le niveau est en deçà du niveau attendu en programmation particulièrement sur les compétences de résolution de problèmes [25, 26, 45].

Dans une recherche menée par [52], ils proposent à des apprenants, non formés aux méthodes mais l'ayant appris implicitement, l'exercice du calcul de la moyenne dont la conception est connue pour vérifier leur application des stratégies. Seul un étudiant sur 42 a appliqué toutes les étapes attendues. Sur les huit aspects mesurés nécessaires pour une solution correcte, les novices étaient en moyenne capables de démontrer l'utilisation de quatre aspects. La maîtrise du contrôle de la division par zero est faible. Dès lors, [52] a défendu l'idée d'inclure explicitement les stratégies dans l'apprentissage et que cela amène à réfléchir sur des curricula et des méthodes qui permettent de réduire les taux d'abandon.

Pour une évaluation de programmeurs, experts et novices, sur une échelle basée sur la taxonomie de SOLO (tableau 2.3) sur la lecture de programmes [53], près de 87,5% des experts donnent la réponse relationnelle. Pour les novices, approximativement 30% des participants sont capables de donner un réponse relationnelle, 55% donnent une réponse multi-structurelle , 13% une réponse uni-structurelle et le pourcentage restant donne une réponse pré-structurelle. Cela implique que beaucoup de novices décrivent au mieux le code ligne par ligne. Moins du tiers sont capables de dire ce que fait réellement le code. Il est clair "qu'une étape essentielle pour être capable d'écrire des programmes est la capacité de lire un morceau de code et de le décrire de manière relationnelle" [53].

Pour [27], les constructions du langage ne constituent pas des obstacles pour les novices mais leur vrai problème est dans le fait de "mettre les pièces ensemble", de composer et de coordonner les composants d'un programme. Les programmeurs experts maîtrisent beaucoup plus que les constructions du langage. Ils ont constitué de grandes bibliothèques de solutions stéréotypées aux problèmes ainsi que des stratégies pour les coordonner et les composer. Il faut souligner que l'application des méthodes s'appuie sur des connaissances de différents niveaux et de différents domaines : connaissance du domaine, connaissance des types de structures de données (types de base, tableaux, listes, piles, arbres ...), connaissances d'algorithmes généraux (tri, récursivité, ...), spécificité des systèmes et langages ...

TABLE 2.3 – Taxonomie de Solo

Catégorie	Description
Relationel	Fournit un résumé de ce que fait le code en terme d'objectif du code.
Multistructurel	Une description ligne par ligne de l'ensemble du code est fournie. Un résumé des déclarations individuelles peut être inclus
Unistructurel	Fournit une description d'une partie du code (par exemple, décrit l'instruction if).
Prestructurel	Manque substantiellement de connaissances sur les concepts de programmation ou n'a aucun rapport avec la question.
Vide	Sans réponse

Les cours d'introduction sont dirigés par les connaissances factuelles parce qu'elles se focalisent sur la syntaxe des langages, les outils techniques avec quelques exercices. Il faut distinguer entre les connaissances factuelles (par exemple la syntaxe de la "boucle pour") qui sont de nature déclarative et les stratégies de programmation (utiliser "la boucle pour" dans un programme de manière appropriée)[44].

Pour [45], les revues australiennes montrent que les formateurs ne sont pas unanimes sur ce qui constitue l'aspect résolution de problèmes dans le cours d'introduction ; la partie dédiée à la résolution de problèmes varie fortement d'un cours à l'autre et certains enseignants disent que la résolution de problèmes n'est pas partie intégrante de leur cours. Plusieurs instructeurs ont estimé que les problèmes utilisés dans leur enseignement n'étaient pas d'une ampleur suffisante pour justifier l'enseignement de stratégies de résolution de problèmes de manière explicite. Pour [45], certains instructeurs ne distinguaient pas uniformément l'enseignement des stratégies de programmation de l'enseignement des connaissances en programmation dans leur module.

Les apprenants doivent ainsi être formés à analyser un problème par une démarche et acquérir ces bibliothèques de motifs pour s'appuyer dessus mais aussi connaître les stratégies pour les composer.

Pour [45], il faut clairement différencier l'apprentissage des connaissances factuelles (syntaxe du langage) de l'apprentissage des stratégies. Il trouve que cette idée n'a pas réellement réussi jusqu'à l'introduction de l'approche objet qui apporte avec lui le concept de réutilisation et celui de motif. Dans [44], les auteurs rappellent que certains avaient proclamé que le paradigme objet est une manière naturelle de conceptualiser et de modéliser les situations du

monde réel.

Cependant, même avec le paradigme objet, les difficultés sur l'abandon et la résolution de problèmes sont apparus. Elles ont été à l'origine taxées à l'étendue un peu vaste du programme en initiation avec une approche objet. En réalité, faire la correspondance entre les objets du monde réel et des objets du domaine de la programmation n'est pas aussi aisé pour le novice qu'il l'est pour l'expert ; ce dernier s'appuie aussi bien sur une vision objet que sur une vision procédurale tout en basculant de l'un à l'autre quand c'est nécessaire [44]. Donc le langage choisi et son paradigme n'a pas montré d'impact sur cet aspect. Pour [54], l'apprentissage de la programmation est l'apprentissage des motifs et stratégies qui facilitent l'apprentissage d'un nouveau langage de programmation. Cependant les novices n'appliquent pas naturellement les motifs, et il revient aux formateurs de "créer des exercices et des supports appropriés pour que les élèves extraient des modèles, les réutilisent, développent une aptitude à utiliser des modèles et créent les leurs" [45].

La clé pour encourager les novices réside tout simplement dans l'application des stratégies plutôt que dans l'acquisition des connaissances factuelles sur le langage quel qu'il soit [44].

2.3.6 Conclusion

Les compétences de résolution de problèmes constituent de nos jours la base pour un apprentissage réussi tout au long de la vie mais également pour la conduite des activités personnelles et pour une participation citoyenne à la vie de la communauté.

Dans le cadre de la programmation informatique, ces compétences font référence à l'usage des méthodes formelles de conception de programmes ou d'un guide de bonnes pratiques pour résoudre un problème.

Il ressort que ces compétences sont faibles chez les apprenants à la fin de leur cours d'initiation à l'algorithmique et à la programmation. Cela s'explique par le fait que les enseignements se focalisent sur la syntaxe des langages et n'intègrent pas toujours ces aspects de manière explicite du fait qu'ils n'apparaissent pas dans le curriculum ou que le formateur ne l'aborde pas explicitement par choix.

Ces compétences peuvent toutefois être développées durant l'apprentissage de la programmation.

2.4 Les stratégies de développement des compétences de résolution de problèmes durant l'initiation à la programmation

2.4.1 Introduction

La programmation est nativement un exercice de résolution de problèmes faisant énormément appel à de fortes compétences méta-cognitives beaucoup plus qu'un simple exercice de codage. Certes les compétences de résolution de problèmes sont faibles chez beaucoup d'apprenants à la fin du cours d'introduction mais ces compétences peuvent être développées durant l'apprentissage.

Durant l'initiation à la programmation, plusieurs facteurs peuvent avoir un impact positif sur les compétences de résolution de problèmes et plusieurs stratégies ont également été évaluées dans la recherche pour les promouvoir. Parmi les stratégies ayant montré un impact fort sur les compétences de résolution de problèmes, nous pouvons citer le guidage de l'apprenant et l'explicitation durant le processus d'apprentissage. Cette stratégie a été évaluée en enseignement présentiel et a montré un fort impact sur les compétences de résolution de problèmes. Le travail collaboratif a également eu un impact fort sur les compétences de résolution de problèmes aussi bien en enseignement présentiel qu'avec le numérique sur une FOAD ou un MOOC. Les feedbacks sur le code source ont montré un impact positif sur l'apprentissage mais ce sont particulièrement les feedbacks sur la sémantique du code source qui montrent un impact sur les compétences de résolution de problèmes.

Nous aborderons ainsi dans la section 2.4.2 le guidage de l'apprenant dans le processus de résolution de problèmes et l'explicitation. Nous présenterons ensuite les approches basées sur le travail collaboratif dans la section 2.4.3. Nous aborderons en dernier les approches basées sur les feedbacks sémantiques dans la section 2.4.4 avant de conclure.

2.4.2 Le guidage de l'apprenant dans le processus de résolution de problèmes et l'explication

Les spécialistes en psychologie cognitive se sont intéressés très tôt à la question de l'apprentissage de la programmation. La programmation et son apprentissage ne sont pas une simple activité de code basée sur une seule compétence mais font appel à des processus mentaux complexes. Cela est intrinsèquement lié à la nature complexe de cette activité ; l'apprenant fait appel à un ensemble de processus mentaux de haut niveau pour résoudre un

problème particulier [55].

L'une des stratégies qui a marqué des générations pour développer les compétences de résolution de problèmes durant l'initiation à la programmation est le guidage de l'apprenant dans le processus de résolution de problèmes et l'explicitation [27, 56, 57].

Assez tôt, [27], s'appuyant sur ses recherches et ceux d'autres spécialistes en psychologie cognitive, a défendu la nécessité de former les novices aux stratégies de résolution de problèmes durant l'initiation à la programmation. Il considère que le manque d'impact de la programmation sur la résolution de problèmes peut être dû au fait que les étudiants n'apprennent pas les idées clés qui sous-tendent la programmation : pourquoi le fait d'apprendre où placer un point-virgule en Pascal devrait-il conduire à une meilleure capacité de résolution de problèmes ? L'accent mis sur l'enseignement des constructions syntaxiques du langage de programmation conduit à mettre l'accent sur le programme en tant que résultat du processus de programmation alors qu'apprendre à programmer, c'est vraiment apprendre comment construire des mécanismes et comment construire des explicitations. Le besoin de construire des mécanismes et des explicitations transcende le domaine de la programmation : dans la vie quotidienne, les gens développent constamment des mécanismes et des explicitations pour résoudre des problèmes.

Il faut enseigner explicitement aux élèves que la programmation est une discipline de conception et que le résultat du processus de programmation n'est pas un programme en soi, mais plutôt un artefact qui remplit une fonction souhaitée. Cela facilitera le transfert des connaissances vers d'autres activités de résolution de problèmes.

C'est ainsi que [27] propose un curriculum totalement basé sur l'explicitation des connaissances tacites de l'expert et le guidage de l'apprenant pour le cours d'initiation. Ce curriculum est basé sur la méthodologie d'analyse et de conception à base d'objectifs et de plans que nous avons introduit dans la section sur les méthodes. En effet, un ensemble substantiel de recherches en psychologie cognitive sous-tendent l'idée de cette démarche basée sur les objectifs et les plans : les plans sont des unités d'organisation mentale à la lecture et l'écriture de programmes informatiques comme le sont les schémas dans la lecture et l'écriture des histoires. Le langage objectif/plan à travers la pratique est ainsi au cœur de la proposition de révision du programme d'enseignement de l'introduction à la programmation proposé par [27].

Le curriculum proposé par [27] se fonde sur deux choses :

1. Il faut **enseigner explicitement** à l'apprenant aussi bien les connaissances factuelles que les stratégies mais particulièrement les nom-

breuses **connaissances et stratégies utilisées par les experts**. Elles doivent être rendues explicites et enseignées aux étudiants dans les cours d'introduction à la programmation. Il est important pour lui de distinguer clairement les deux catégories générales de concepts : les connaissances d'une part et les stratégies d'utilisation des connaissances d'autre part. Il ne faut pas seulement enseigner les stratégies par une méthode expositive mais à travers la pratique sur des problèmes.

2. Le deuxième aspect est qu'il faut **nommer les concepts** et explicitement : connaissances et stratégies utilisées par les experts. Il est important que les connaissances et stratégies enseignées portent un nom. En effet, comment les étudiants peuvent-ils apprendre un concept si ce qu'ils doivent apprendre ne leur est pas explicitement enseigné et ne porte pas de nom ? Cette croyance selon laquelle il faut dégager les concepts tacites et les enseigner de manière explicite repose dans une certaine mesure sur l'hypothèse de Sapir-Whorf [58].

   L'attribution d'une étiquette ou d'un nom aux étapes, aux plans et aux stratégies permet d'établir la réalité de ces concepts. Par exemple, pour le plan, l'intention est de donner une idée de l'objectif du plan dans son nom (par exemple plan de protection de la division par zéro). Bien que les noms de plans puissent être pour le moins maladroits, ils donnent des étiquettes explicites aux structures qui devraient être utilisées lors de la résolution de problèmes.

   Il est ainsi donné aux apprenants un document d'une liste de plans utilisés par les experts portant des noms et il leur est demandé de l'enrichir au fil de l'apprentissage. Les étapes de la démarche et les stratégies utilisées seront explicitement nommées et utilisées sur des problèmes concrets.

C'est ainsi que le curriculum proposé par [27] débute par la présentation de la méthode descendante utilisée avec ces différentes étapes, l'importance des plans et l'exposé des stratégies de composition : le séquencement, la fusion et l'emboîtement. L'usage des termes pour désigner les concepts par les apprenants est jugé important.

Les connaissances sont ensuite introduites de même que les stratégies à travers la résolution de problèmes. Il est important de préciser que les plans sont introduits au fil du cours et les problèmes posés ne s'appuient que sur des plans connus. En effet, dans la phase de décomposition, [27] rappelle que pour enseigner le raffinement par étapes, il faut ajouter une heuristique cruciale que tout le monde connaît intuitivement, mais qui est rarement explicitée : un problème est à décomposer en sous-problèmes sur la base de problèmes

que vous avez déjà résolus et pour lesquels vous avez des solutions connues ou presque (plans). En d'autres termes, les apprenants doivent déjà posséder les primitives (plans) en lesquelles le problème sera décomposé afin de mener à bien une stratégie de raffinement par étapes. Cela développe également la confiance en soi de l'apprenant qui lui permettra plus tard de se juger apte à résoudre un problème dont il ne connaît pas à priori les plans.

Ainsi dans les étapes de résolution de problèmes :

1. Dans la phase de compréhension, il est important dans le libellé du problème que le formateur soit explicite pour guider l'apprenant. Les tournures peuvent faire ressortir l'objectif général et ressortir de manière explicite les entrées. Il peut être pratique de citer tout ou certains plans sur lesquelles vont s'appuyer les apprenants.
2. Dans la phase de décomposition, ils apprennent à raffiner sur la base des objectifs et sur la base de plans connus tels que nous l'avons dit plus haut.
3. Dans la phase de proposition de solutions aux sous-problèmes, ils s'appuient sur des plans connus ou presque. Si le plan n'est pas connu mais qu'il existe un plan similaire, il faudra adapter le plan, sinon il faudra en créer un. Il faut noter qu'en début d'apprentissage, il serait adéquat que les problèmes donnés ne s'appuient que sur des plans connus.
4. Dans la phase de composition, ils doivent s'appuyer sur les quatre stratégies de composition : le séquencement, la fusion, l'emboîtement et l'adaptation.
5. Concernant l'évaluation, [27] recommande à l'apprenant de faire une simulation pour l'évaluation même si certaines écoles s'opposent à cette méthode.

Il est important de rappeler que concernant l'explicitation, l'apprenant est au cœur du processus. Même si l'enseignant dans son scénario explicite énormément de choses, l'apprenant doit également apprendre à expliquer en utilisant nommément les concepts : les étapes de la démarche, les stratégies et les objectifs et plans. Il est également important de ressortir à travers des expressions les liens entre les concepts, entre objectifs et plans et aussi entre objectifs eux-mêmes avec des expressions telles que "parce que", "pour" ...

Ce genre d'expressions ont cette force de signaler la présence d'informations expliquant la relation entre les objectifs du problème et les plans instanciés dans le programme et entre les différents concepts.

Dans [27], l'auteur fait remarquer d'ailleurs que dans certains cours les étudiants sont formés à la lecture de code et au discours sur la conception de ce code.

Dans la même ligne que [27], [57] propose une démarche de guidage dans le processus et d'explicitation. En plus de le mettre dans le curriculum sur la séquence d'apprentissage, il propose de l'intégrer aux évaluations.

Toujours dans la même ligne, [56] proposent une approche également basée sur le guidage de l'apprenant dans le processus de résolution de problèmes et l'explicitation, ils ne s'appuient pas toutefois sur la méthode basée sur les objectifs et plans. Dans [56], les auteurs considèrent que la résolution de problèmes algorithmiques s'appuie sur un ensemble de compétences pour lesquelles ils proposent un ensemble d'activités pour les développer dans la conception d'un cours présentiel. Ces compétences et activités sont listées dans le tableau 2.4.

Le scénario sur lequel s'appuient [56] est un scénario incrémental pour permettre à l'apprenant d'évoluer en abstraction : avec une étape initiale d'apprentissage, une étape intermédiaire et une étape finale.

1. Dans l'étape initiale d'apprentissage , il faut sensibiliser l'apprenant sur les étapes du processus de résolution de problèmes en explicitant dans les problèmes proposés toutes les étapes et les tâches liées à ses étapes et en lui fournissant un feedback. Il faut entièrement le guider et expliciter les concepts et également l'amener à expliquer ce qu'il fait.
2. Dans l'étape intermédiaire, il faut réduire de manière croissante le soutien à l'apprenant dans les étapes et les tâches. Il faut ainsi réduire l'explicitation des étapes et des tâches qui les composent au fil de l'apprentissage. Cela revient ainsi à augmenter le degré d'abstraction des problèmes proposés.
3. Dans l'étape finale d'apprentissage, l'apprenant devrait atteindre le degré le plus élevé d'abstraction et être capable de résoudre un problème algorithmique avec une démarche maîtrisée mais non explicitée dans le problème.

Il est important dans ce scénario de fournir à l'apprenant un feedback à chacune des étapes de la résolution de problèmes et d'évaluer son niveau de compétences sur les différentes étapes et tâches.

L'objectif final est de rendre l'apprenant apte à résoudre un problème décrit avec un niveau d'abstraction élevé en suivant toutes les étapes de résolution de problèmes et en évaluant chacune des étapes jusqu'à la résolution du problème.

Ces méthodes et curricula ont été évalués et ont montré un impact positif en présentiel sur les compétences de résolution de problèmes et les compétences métacognitives de l'apprenant [27, 56, 57].

TABLE 2.4 – Activités pour le développement des compétences de résolution de problèmes

Compétence	Activité d'apprentissage
Compréhension du problème	Reformulation de l'énoncé du problème en termes d'état initial, de but, d'hypothèses et de contraintes.
Décomposition du problème	Identifier, nommer et énumérer les sous-tâches.
Raisonnement analogique	Identification des similitudes entre les problèmes. Distinction entre similitudes structurelles et superficielles. Sensibilisation aux erreurs courantes causées par la référence à des problèmes inadaptés.
Généralisation et abstraction	Extraction de prototypes de problèmes à partir de problèmes analogiques dans différents contextes.
Identification du prototype du problème	Introduction systématique et bien structurée de modèles algorithmiques. Catégorisation des problèmes.
Structure du problème, identification	Identifier la relation entre les sous-tâches. Schématiser la structure d'un problème à l'aide de diagrammes.
Évaluation et appréciation de l'efficacité et de l'élégance	Comparaison des solutions en termes d'efficacité et d'élégance.
Réflexion et tirage de conclusions	Réfléchir aux processus et stratégies de résolution de problèmes. Tirer des conclusions pour l'avenir.
Verbalisation des idées	Formulation d'une idée précise, différenciation entre une idée et sa mise en œuvre.

Pour [45], cette approche d'apprentissage démontre que l'apprentissage explicite peut être plus efficace que des mois d'apprentissage implicite. Les apprenants ayant suivi ce genre de formation utilisent un vocabulaire incluant la terminologie liée à la stratégie de résolution de problèmes et ont montré une plus grande confiance dans l'exactitude de leurs solutions. Dans le cadre de la mise en œuvre de méthodes apprises, les débutants construisent des structures de programmes bien hiérarchisées si la méthode est bien acquise.

2.4.3 Le travail collaboratif

Le travail collaboratif a toujours eu un impact positif sur l'apprentissage en général s'il est bien scénarisé. Il montre également un impact positif sur les compétences de résolution de problèmes dans le cadre de l'apprentissage de la programmation. Parmi les stratégies de travail collaboratif qui ont montré un impact très positif sur ces compétences durant l'apprentissage de la programmation, nous pouvons citer l'instruction par les pairs (peer instruction) et l'apprentissage par découverte guidée (Process Oriented Guided Inquiry Learning - POGIL [2]) [59, 60, 61].

Nous rappelons que l'**instruction par les pairs** est une méthode introduite par [62]. Elle consiste à ce que les apprenants répondent à une question, et ensuite discutent de la question en petits groupes avant de répondre une nouvelle fois à la question. Dans cette méthode, les apprenants sont sensés avoir pris connaissance du cours avant de venir en classe et sont sensés pouvoir convaincre leurs pairs de leur réponse.

L'**apprentissage par découverte guidée** a également montré un impact positif sur les compétences de résolution de problème [59, 61]. Le système POGIL est une pédagogie centrée sur l'élève qui l'aide à maîtriser les concepts spécifiques à une discipline tout en développant des compétences qui sont essentielles dans les carrières professionnelles et qui contribuent à l'apprentissage et à la réussite scolaire des élèves [63]. POGIL se concentre sur sept compétences de processus : la communication orale et écrite, le travail d'équipe, la résolution de problèmes, la pensée critique, la gestion, le traitement de l'information et l'évaluation (auto-évaluation et métacognition). Cette focalisation sur les compétences de processus fait que les activités POGIL suivent les cycles d'apprentissage Exploration -Invention - Application (Explore-Invent-Apply) dans lesquels les élèves explorent un modèle, inventent leur propre compréhension d'un concept clé et appliquent leur compréhension à un contexte différent [64]. Lorsque les équipes d'étudiants travaillent sur une activité POGIL, chaque étudiant se voit attribuer un rôle spécifique pour faciliter l'apprentissage du contenu et des compétences de processus. Les rôles courants sont les suivants :

1. manager : veille à ce que l'équipe reste concentrée sur sa tâche et respecte le calendrier ;
2. contrôleur : enregistre les réponses pour l'équipe et s'assure que tous les membres ont les bonnes réponses ;

2. https ://pogil.org/

3. porte-parole : présente les réponses ou les questions à la classe, à l'instructeur ou aux autres équipes ;
4. analyste : réfléchit à la façon dont l'équipe travaille et à la façon dont elle pourrait s'améliorer à l'avenir.

Cette approche se développe de plus en plus dans les cours d'informatique ou il est demandé aux apprenants de travailler sur des projets par cette méthode.

2.4.4 Le feedback sur la sémantique du code

Le feedback est, pour [65], une information fournie par un agent (enseignant, pair, parent ...) sur la performance d'un individu. Pour [66], dans le cadre des apprentissages, "les feed-back sont des informations informelles ou intentionnelles, positives ou négatives, uniques ou séquentielles, fournies par soi-même, autrui ou un dispositif de manière immédiate ou différée, axées sur la tâche, les processus, l'autorégulation ou la personne, engendrés par les diverses conséquences de l'activité de l'individu et présentant une fonction affective liée à la motivation pour réaliser les tâches et des fonctions cognitives d'évaluation de l'activité en fonction d'un objectif ou d'une norme, d'aide à la réalisation de tâches et au développement des connaissances et conduisant un apprenant à combler l'écart entre le résultat de son activité actuelle et le résultat attendu à la fin de l'apprentissage."

Cette définition qui ressort clairement les caractéristiques montre que le feedback n'a un effet positif sur l'apprentissage que s'il respecte un certain nombre de caractéristiques [67].

L'impact des feedbacks sur l'apprentissage, quel que soit le domaine, n'est plus à démontrer aujourd'hui [68, 69, 65]. Selon l'information véhiculée par le feedback et ses caractéristiques, il peut avoir un impact positif sur les dimensions cognitives, méta-cognitives, motrices, affectives ou émotionnelles ou encore sur l'engagement de l'apprenant ou la réussite [69]. C'est ce qui explique la multitude d'outils qui existent dans ce cadre.

Dans nos travaux, nous nous intéressons aux feedbacks sur la sémantique du code source qui ont montré un impact positif sur les compétences de résolution de problèmes. Leur automatisation durant l'initiation à l'algorithmique et à la programmation pourrait améliorer les gains d'apprentissage et participer aussi au développement des compétences de résolution de problèmes, en particulier pour les étudiants qui sont dans des classes nombreuses où le temps de l'instructeur est limité. Nous nous intéressons particulièrement au processus d'analyse de la sémantique du code source qui est à la base de ces systèmes.

Nous avons en réalité trouvé très peu de travaux sur l'analyse automatique de la sémantique de code source qui s'explique, pour [70], par le fait que la sémantique d'un programme est non calculable. Dans une revue de la littérature basée sur plus d'une centaine d'outils de feedbacks automatisés dans le cadre de la programmation, [71, 72] constatent que les feedbacks se concentrent principalement sur l'identification des erreurs et beaucoup moins sur la résolution des problèmes, donc beaucoup moins sur la sémantique du code source.

Les outils rencontrés dans ce contexte peuvent être classés en deux familles :

1. les outils basés sur une **approche dynamique** dans laquelle le code source est exécuté.
2. les outils basés sur une **approche statique** ou le code est analysé sans exécution. Cette famille d'outil peut être décomposée en deux catégories :
   (a) les outils basés sur une **approche manuelle** dans laquelle le code est analysé par un expert humain ;
   (b) les outils basés sur une **approche automatique** dans laquelle le code est automatiquement analysé par le système pour générer un indicateur.

Dans la catégorie des outils basés sur une **approche dynamique**, nous pouvons citer EPFL Grader [73] qui est un outil d'évaluation automatique de code source. EPFL grader est un outil utilisé sur le MOOC "Introduction to programming with C++" (MOOC présenté par Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit via la plateforme Coursera à l'Ecole Polytechnique Fédérale de Lausanne, Suisse) [73]. Pour chaque session, ce cours a proposé un ensemble d'exercices de programmation aux participants. Les exercices pouvaient être résolus en soumettant un code source sous forme de fichier dans le navigateur web. Les solutions soumises par les étudiants sont compilées et testées sur un ensemble d'entrées. En retour, les étudiants reçoivent un score et un feedback automatique sur la façon dont leur code s'est comporté dans le test.

L'outil est basé sur deux composantes principales :

1. une batterie de tests unitaires où les programmes des étudiants sont exécutés avec une entrée standard et évalués pour savoir s'ils produisent une sortie correcte.
2. un vérificateur de style qui peut déduire des points pour un mauvais style dans les programmes en cours d'exécution et donner un feedback automatique.

Les méthodes basées sur les tests unitaires montrent une difficulté : ils requièrent beaucoup d'efforts de la part des formateurs.

Dans la catégorie des outils basés sur une **approche statique manuelle**, nous pouvons citer ALGO+ [74, 75], qui est un outil d'évaluation automatique basé sur l'ontologie ALGOSKILL [76]. Dans ALGO+, les feedbacks sont certes automatisés pour l'apprenant et peuvent porter sur la sémantique du code source, cependant l'analyse de la sémantique du code source est faite par un expert humain. Les soumissions des apprenants dans ce système sont évaluées en les comparant à un ensemble de propositions référentes déjà évaluées par un instructeur. Les propositions référentes sont celles qui sont détectées fréquemment dans les soumissions des étudiants, correctes ou erronées par comparaison. Lorsqu'une soumission n'est pas similaire à une solution référente mais qu'elle commence à devenir fréquente, elle est alors envoyée à un instructeur qui l'évalue, lui associe un feedback et l'ajoute aux solutions référentes. Toute futur soumission similaire recevra le feedback proposé par l'instructeur sur cette solution référente.

Les approches manuelles, autant que les approches dynamiques, demandent également beaucoup d'efforts de la part des formateurs qui sont sensés évaluer les codes sources.

Dans la catégorie des méthodes basées sur une **approche statique automatique** d'analyse de la sémantique du code source, nous pouvons citer les travaux de [70, 77]. En réalité, c'est la seule approche aboutie que nous avons rencontrée dans le cadre des méthodes automatiques d'analyse de la sémantique de code source.

Il faut noter que les approches automatiques ont cet avantage de demander moins d'effort aux formateurs et offrent un deuxième avantage, l'immédiateté du feedback.

Dans la méthode proposée par [70, 77], une solution de réference est proposée par le formateur pour chaque problème. Les auteurs se proposent de calculer la distance entre le code source proposé par l'apprenant et la solution modèle proposée par le formateur. Le cadre d'application est l'apprentissage des scripts shell BASH. Les auteurs se proposent d'identifier les opérations à effectuer pour passer d'un code source à un autre, et ainsi calculer une distance correspondant à la somme des coûts de chacune des opérations à réaliser en s'inspirant de la distance d'édition de LEVENSHTEIN [78] entre deux chaînes de caractères.

Dans une première étape, les auteurs transforment les codes sources en arbre de syntaxe abstraite (AST) [70, 77](figure 2.12), qui, à la différence des arbres d'analyse, ne représentent pas les nœuds et les branches qui n'affectent

pas la sémantique d'un programme.

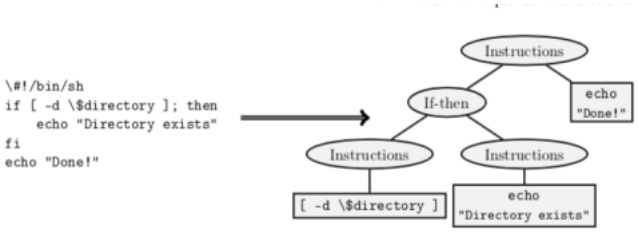

FIGURE 2.12 – Exemple de transformation de script en AST

Dans une deuxième étape, ils transforment l'AST en chaîne de token [70, 77] (figure 2.13) :

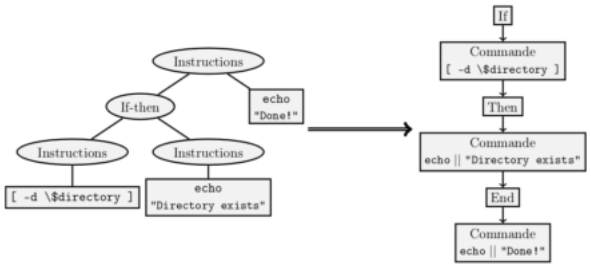

FIGURE 2.13 – Transformation d'un AST en chaîne de token

Dans une troisième étape, ils déterminent le coût des opérations de base pour la transformation d'un code source (code de l'apprenant) en un autre (code de l'expert). En effet, dans le cas de la comparaison de codes sources, les coûts de ces opérations de base ne peuvent être tous égaux comme le rappellent les auteurs : la substitution de deux instructions visant les mêmes objectifs ne doit pas avoir le même coût que la substitution d'une commande par une autre ayant une fonctionnalité très différente. Les auteurs se proposent d'attribuer un coût à chacune des opérations selon l'importance des modifications à réaliser pour passer d'un code source à un autre.

Nous rappelons que la distance de LEVENSHTEIN est définie par un coût calculé à partir du nombre minimum d'opérations de base pour passer d'une chaîne de caractère à une autre :

— suppression d'un caractère
— insertion d'un caractère
— substitution d'un caractère par un autre

Les coûts associés à chacune de ces opérations de base étant tous fixés à 1 pour la mesure de LEVENSHTEIN.

Sur les 17 structures proposées par le langage BASH, [70, 77] éliminent trois qui n'apparaissent pas dans leur jeu de données : Until, Subshell et pipeline. Ils adoptent une symétrisation de manière à ce que le coût de la substitution de A par B est égal à celui de B par A et le coût de l'insertion et de la suppression sont fixé à 1. Ils considèrent également que les token ne se substituent que par groupe : {Command, Assignation}, {If, Case}, {Then, Else, CaseItem}, et {For, While}. Ils ont ainsi à calculer 23 coûts avec un échantillon de 733 scripts indépendants après nettoyage dont 397 sont corrects. θ étant le vecteur des 23 coûts, $d(S, C, \theta)$, la distance entre les scripts C et S sous les 23 coûts.

Ils ont utilisé une méthode d'apprentissage automatique pour définir ces 23 coûts et avoir ainsi un indicateur. Ils proposent de calculer $\overline{\theta}$ qui minimise $Score(\theta)$ par l'algorithme de minimisation de descente de gradient.

$$Score(\theta) = \frac{\sum_{(S,C)\in Correct} \sqrt{d(S,C,\theta)}}{\sum_{(S,C)\in Incorrect} \sqrt{d(S,C,\theta)}}$$

$Correct$ est l'ensemble des paires composées des scripts corrects et $Incorrect$ l'ensemble des paires composées des scripts incorrects du jeu d'apprentissage et des corrections associées. La fonction $Score$ est faible quand les scripts corrects sont associés à de faibles distances, et les scripts incorrects à de fortes distances.

Les auteurs ont validé leur indicateur par deux méthodes :

— la bonne classification d'un script comme correct ou incorrect. leur indicateur présente des performances légèrement supérieures à celles d'un classifieur aléatoire ;

— la comparaison du coût de transformation du script en celui de l'expert avec la note d'un formateur sur 105 nouveaux scripts. ils ont observé une corrélation inverse entre la valeur de l'indicateur et le score attribué par une évaluation humaine : plus la notation humaine d'un programme est élevée, plus la distance entre ce programme et la solution du problème posé est faible.

Cette approche est très intéressante mais comme le signalent les auteurs, la corrélation n'est pas extrêmement élevée :

— corrélation de Kendall à -0.319 (valeur de p inférieure à $2,710^{-6}$)

— corrélation de pearson à -0.446 (valeur de p égale à $1,910^{-6}$)

2.4.5 Conclusion

Développer les compétences de résolution de problèmes durant l'initiation à l'algorithmique et à la programmation est aujourd'hui un défi.

Plusieurs stratégies sont aujourd'hui utilisées aussi bien en enseignement présentiel qu'avec des outils numériques pour venir en appui dans un enseignement présentiel ou dans un enseignement à distance ou sur un MOOC.

Parmi les premières stratégies figurent le guidage de l'apprenant dans le processus de résolution de problèmes et l'explicitation qui a eu un impact positif sur les compétences de résolution de problèmes.

Le travail collaboratif a également montré un impact positif sur les compétences de résolution de problèmes.

Les feedbacks sur la sémantique du code source de l'apprenant ont également un impact positif sur ces compétences. Les outils basés sur une approche dynamique ou une approche statique manuelle d'analyse de la sémantique de code source nécessitent beaucoup d'efforts de la part des formateurs. Nous avons rencontré très peu d'approches qui automatisent le processus d'analyse de la sémantique du code source. Les outils basés sur une approche statique automatique s'appuient sur des méthodes d'apprentissage automatique mais jusque là avec des taux de précision assez faible.

2.5 Conclusion

Dans cette première partie nous avons fait l'état de l'art sur l'initiation à l'algorithmique et à la programmation, la résolution de problèmes et le développement des compétences de résolution de problèmes durant l'initiation à l'algorithmique et à la programmation.

L'initiation à l'algorithmique et à la programmation est un fondamental dans la formation d'un informaticien. Ce dernier est sensé être apte à analyser, concevoir et implémenter une solution à un problème algorithmique. Malheureusement, les taux d'abandon et d'échec sont encore relativement élevés. La faiblesse des compétences de résolution de problèmes est vue comme une des causes principales. Ces compétences sont d'ailleurs faibles chez beaucoup d'apprenants ayant réussi.

Les compétences de résolution de problèmes font référence à une démarche formelle ou basée sur un guide de bonnes pratiques permettant d'analyser un problème algorithmique, de proposer et d'évaluer une solution à ce problème.

Ces compétences peuvent être développées chez l'apprenant durant l'initiation à la programmation à travers plusieurs méthodes évaluées en enseignement présentiel mais aussi à travers des outils numériques.

Parmi ces stratégies nous pouvons citer le guidage de l'apprenant dans le processus de résolution de problème et l'explicitation. Nous pouvons également citer le travail collaboratif. Le guidage est une stratégie qui a été évaluée en présentiel et a eu un impact positif sur ces compétences de même que le travail collaboratif qui a été évalué aussi bien en présentiel qu'à distance.

Nous pouvons aussi citer les feedbacks sur la sémantique du code source qui ont montré un impact positif sur les compétences de résolution de problèmes.

Il faut toutefois remarquer que nous avons rencontré peu de travaux dans cet axe, particulièrement sur les méthodes basées sur des approches automatiques d'analyse de la sémantique du code source. Les méthodes basées sur une approche dynamique ou une approche statique manuelle requièrent beaucoup d'efforts de la part des formateurs. Les méthodes basées sur une approche automatique font appel à des méthodes d'apprentissage automatique mais donnent encore des taux de précisions relativement faibles.

Chapitre 3

Contributions

3.1 Introduction

Développer les compétences de résolution de problèmes durant l'initiation à la programmation est un vrai défi aujourd'hui dans les formations à vocation scientifique. Le temps de formation est limité et le nombre d'apprenants élevé, l'usage d'un EIAH reste une bonne alternative dans ce contexte. Beaucoup d'outils sont utilisés dans le cadre des EIAH pour améliorer l'apprentissage mais pas particulièrement pour s'adresser à la compétence de résolution de problèmes. Nous proposons une approche d'accompagnement de l'apprenant avec un EIAH, IDE4SCAPSS, pour le développement des compétences de résolution de problèmes durant l'initiation à la programmation. Cette approche s'appuie sur deux stratégies :

- Le **guidage de l'apprenant** dans le processus de résolution du problème et l'**explicitation**. L'apprenant est accompagné dans plusieurs aspects dans ce sens :
  - il est guidé dans un processus de conception simple.
  - il bénéficie d'explicitations à travers des problèmes détaillés et d'une aide.
  - il est appelé à expliciter lui même certains choix par exemple par l'activité de reformulation de problèmes.
  - il bénéficie de feedbacks dans le processus de conception.
- L'analyse et la comparaison automatique de la sémantique du code source de l'apprenant avec un code source expert afin de lui proposer un **feedback sémantique**. Nous proposons un système d'analyse de la sémantique du code source basé sur une approche statique automatique qui nécessite peu d'efforts de la part du formateur à la différence des méthodes basées sur une approche dynamique ou une approche statique manuelle. A la différence des méthodes basées sur une approche automatique qui s'appuient sur de l'apprentissage automatique avec le taux d'erreur que cela implique, notre méthode de comparaison s'appuie sur les mathématiques symboliques et le calcul formel et fournit ainsi une meilleure précision.

Nous rappelons que ces deux stratégies ont montré un impact positif sur les compétences de résolution de problèmes.

Nous présenterons ainsi dans la section 3.2 la stratégie de guidage de l'apprenant proposée dans notre système et l'explicitation. Nous aborderons ensuite dans la section 3.3 le système d'analyse et de comparaison sémantique de codes sources sur lequel repose notre système et nous terminerons dans la section 3.4 par les aspects relatifs aux choix techniques et à l'implémentation avant de conclure.

3.2 Guidage de l'apprenant et explicitations dans le processus de résolution de problèmes

3.2.1 Introduction

IDE4SCAPSS s'appuie sur un cadre théorique inspiré par les travaux sur le guidage de l'apprenant en présentiel et l'impact des feedback sur la sémantique du code. Comme nous l'avons vu dans l'état de l'art, le guidage de l'apprenant et l'explicitation sur un scénario d'apprentissage incrémental a eu un impact positif sur les compétences de résolution de problèmes durant l'apprentissage de la programmation. Ces scénarios intègrent des feedbacks sur le processus qui ont un impact positif sur l'apprentissage.

Nous aborderons dans la section 3.2.2 le cadre théorique sur lequel s'appuient nos travaux. Nous présenterons ensuite dans la section 3.2.3 le processus de guidage de l'apprenant et les aspects relatifs à l'explicitation avant de conclure

3.2.2 Cadre théorique

Notre objectif est de venir en appui à l'apprenant dans le développement des compétences de résolution de problèmes avec un outil numérique. Nous proposons ainsi un environnement de développement intégré simple, IDE4SCAPSS, dans lequel l'apprenant peut éditer du code mais aussi être guidé dans le processus de conception et aidé sur la sémantique du code source.

Les travaux sur le **guidage de l'apprenant** et l'explicitation montrent clairement l'impact positif de ces stratégies sur l'apprenant en présentiel. Une analyse des difficultés rencontrées dans le cadre du développement des compétences de résolution de problèmes montre également que la **compréhension du problème** est centrale et critique dans la stratégie adoptée.

C'est ainsi que nous proposons un outil basé sur le principe du guidage de l'apprenant dans le processus de résolution de problèmes. Il s'appuie sur une méthode de résolution de problèmes descendante simple guidée par les données et qui intègre l'aspect compréhension du problème [79, 80] :

1. abstraction/compréhension du problème ;
2. détermination des sorties du problème ;
3. détermination des entrées du problème ;
4. proposition de solutions ;

5. évaluation de la solution.

Le **scenario incrémental** de ces stratégies et les **explicitations** ont également montré un impact positif sur ces compétences. Les feedbacks sur l'activité de l'apprenant que nous pouvons classer dans les explicitations jouent également un rôle important. Nous introduisons ce principe de scénario incrémentale et d'explicitation à travers trois niveaux :

1. Le **niveau débutant** dans lequel le libellé des problèmes est détaillé en réduisant l'abstraction. Nous faisons clairement ressortir dans le libellé les étapes de résolution de problèmes demandées, les objectifs et résultats attendus. L'apprenant est entièrement guidé à ce niveau : il lui est explicitement demandé de donner les sorties du problème et leur type et de donner les entrées du problème et leur type. Il doit bénéficier d'un feedback à chacune des étapes du processus.
2. Le **niveau intermédiaire** dans lequel le libellé du problème est un peu plus abstrait, les éléments d'aide sous forme d'explicitations doivent être également réduits.
3. Le **niveau expert** dans lequel il est directement demandé à l'apprenant de résoudre un problème avec très peu d'éléments d'aide dans le libellé du problème et dans le processus de résolution du problème.

Notre système propose des problèmes sélectionnés par des experts et dont les solutions font appel aux structures séquentielles, conditionnelles et itératives et de différents niveaux de difficulté. La liste des problèmes actuellement disponibles dans la base est donnée dans l'annexe A.

Nous rappelons que les aspects relatifs à l'explicitation doivent être visibles dès le début de la résolution de problèmes à travers les libellés. c'est ainsi que les problèmes proposés par les experts dans l'outil sont déclinés pour les trois niveaux cités plus haut : débutant, intermédiaire et expert. Nous avons un exemple avec le problème du maximum de trois variables dans le tableau 3.1.

Le dernier aspect théorique sur lequel se fonde notre outil est l'impact positif des **feedbacks sémantiques** sur les compétences de résolution de problèmes. Cet aspect est intégré à notre stratégie dans l'étape de proposition de solution. L'apprenant est appelé à cette étape à faire de l'ordonnancement du code source expert mélangé ou à proposer un code fonctionnel. Le code source fonctionnel proposé par l'apprenant sera analysé et comparé à un code source expert sur la sémantique.

TABLE 3.1 – Variantes du problème du maximum de trois entier par niveau

Niveau	Libellé
Débutant	Nous souhaitons mettre en place un programme qui permet de récupérer au clavier trois variables entières puis calcule dans une variable entière le maximum des trois variables. 1. Il vous est d'abord demandé de fournir la sortie de ce programme et son type de données. 2. Il est ensuite demandé de donner les entrées et leur type, nous rappelons que nous souhaitons calculer le maximum de trois variables entières 3. Proposez une solution à ce problème et évaluez la. **éléments d'aide** : vous pouvez comparer deux variables et ensuite comparer le résultat à la troisième variable.
Intermédiaire	Proposer un programme qui calcule le maximum de trois nombres entiers saisis au clavier. Le maximum sera calculé dans une variable entière et affiché.
Expert	Proposer un programme qui calcule et affiche le maximum de trois nombres entiers saisis au clavier.

3.2.3 Processus de guidage de l'apprenant

Le guidage de l'apprenant consiste à lui proposer des activités permettant pour chaque étape du processus de résolution de problèmes de développer les compétences relatives à cette étape.

Comme nous l'avons souligné plus haut, nous avons privilégié, pour des débutants, une méthode de résolution de problèmes simple guidée par les données et qui inclus l'aspect compréhension du problème à travers la reformulation.

Dans la première version, introduite dans [79], l'apprenant est appelé a suivre les étapes suivantes : détermination des données d'entrées et de leur type, détermination des données de sorties et de leur type, proposition d'une solution par ordonnancement du code source expert mélangé, évaluation de la solution.

Cette version a été étendue dans [80] pour prendre en compte l'aspect compréhension du problème qui est essentiel dans le processus de conception et introduire les feedbacks sémantiques.

L'apprenant est ainsi appelé à suivre un processus de résolution de problème en cinq (5) étapes (figure 3.2) :

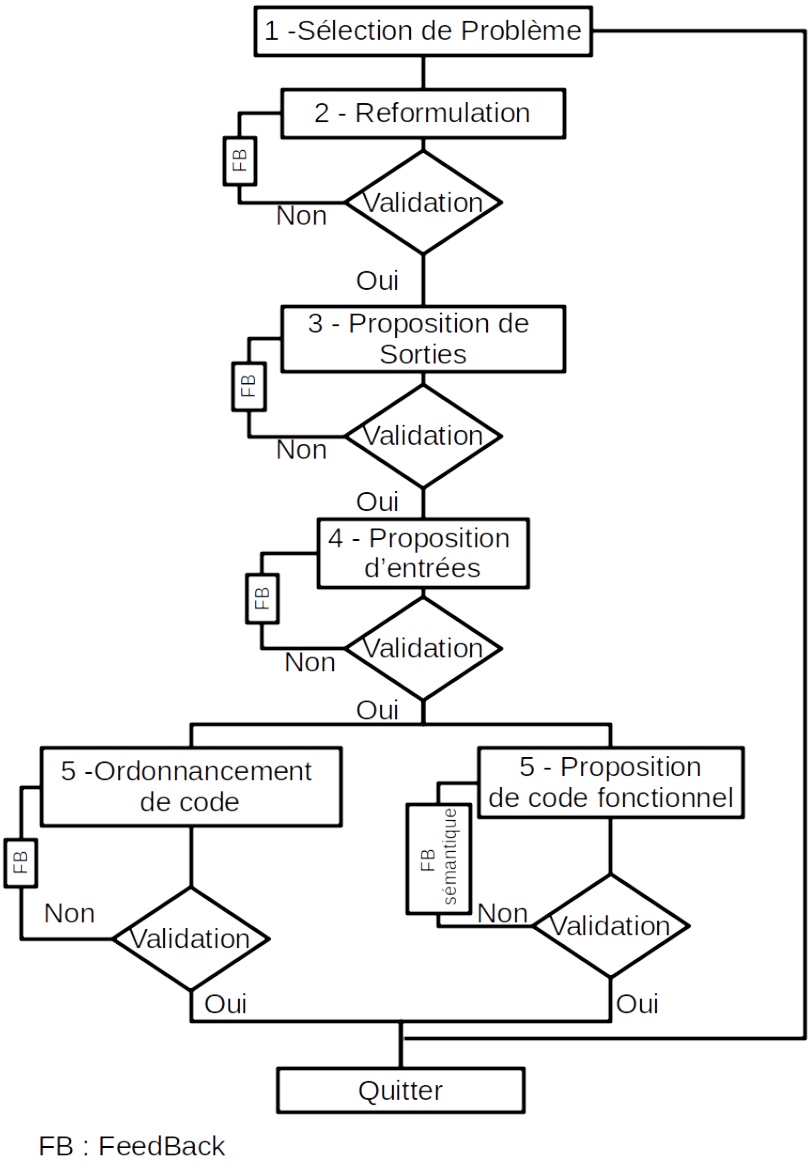

FIGURE 3.1 – Processus suivi par l'apprenant sur IDE4SCAPSS

1. **Reformulation du problème** : cette activité permet de développer les compétences liées à la compréhension du problème. Il est demandé à l'apprenant à cette étape de proposer une reformulation du problème qui fait ressortir les objectifs, les sorties et les entrées. Cette activité participe fortement à une meilleure compréhension du problème et participe au développement de cette compétence.

2. **Identification des données de sorties et de leur type** : à cette étape, l'apprenant est appelé à fournir le nombre de données de sorties attendu pour le problème et leur type. Les problèmes proposés par le système sont des problèmes dont les sorties sont bien identifiées de même que les entrées pour éviter toute ambiguïté. Les propositions de l'apprenant sont comparées aux propositions de l'expert du domaine. Cette étape n'est validée que lorsque le nombre de sorties et les types de données proposés par l'apprenant sont identiques à ceux proposés par l'expert.

3. **Identification des données d'entrée et de leur type** : une fois les sorties validées, l'apprenant est appelé à fournir le nombre d'entrées et leur type. Cette étape également n'est validée que lorsque le nombre d'entrées et les types proposés sont identiques à ceux proposés par l'expert. La méthode sur laquelle nous nous appuyons étant guidée par les données, ces deux étapes sont très importantes parce qu'elles amènent l'apprenant à ressortir les données et à mener une réflexion sur le lien entre les données et les traitements.

4. **Proposition de solution :** une fois les trois premières étapes validées, l'apprenant peut accéder à la zone d'édition de code source. Il a le choix entre deux tâches :
   - **Ordonnancement de code source** : Le code source expert mélangé est proposé à l'apprenant sur l'éditeur. Il lui revient d'ordonner correctement les instructions.
   - **Proposition de code fonctionnel** : Le système affiche le code source de déclaration des variables, le code source de récupération des variables d'entrées et celui d'affichage des variables de sortie. Il est ensuite demandé à l'apprenant de proposer un code fonctionnel en complément du code donné. Il faut noter que l'apprenant n'est appelé à proposer que le code fonctionnel du programme. En réalité, le code fonctionnel du programme est la partie du code source qui fait réellement la résolution du problème.

5. **Évaluation de la solution** : L'apprenant après avoir ordonnancé le code source expert mélangé ou après avoir proposé un code fonctionnel pourra évaluer sa solution.

   Pour l'ordonnancement, l'évaluation se fait sur la vérification de l'ordre proposé pour les instructions. Pour l'option de proposition de code fonctionnel l'évaluation se passe en deux étapes :

   (a) la compilation classique du code pour la vérification lexicale et syntaxique ;

(b) la comparaison sémantique du code source de l'apprenant au code source de l'expert, cette étape fait l'objet de la section suivante.

Prenons l'exemple simple du problème de détermination du maximum de trois variables entières. Le langage de programmation d'application utilisé ici est le langage PASCAL. Le problème est ainsi libellé pour le niveau intermédiaire :

"Proposer un programme qui calcule le maximum de trois nombres entiers saisis au clavier. Le maximum sera calculé dans une variable entière et affiché."

L'apprenant ayant choisi de résoudre ce problème sera appelé à proposer une reformulation, puis une sortie de type entier et enfin trois entrées de type entier. Une fois ces trois étapes validées, il aura le choix entre l'ordonnancement du code source expert ou la proposition de code fonctionnel. Pour la proposition de code fonctionnel, le système lui affichera le code suivant en l'autorisant à proposer un code fonctionnel :

```
Program MaxTroisEntiers;
    {Sorties}
    Var maxi: integer;
    {Entrées}
    i1,i2,i3 :integer;
begin
    {Lecture des entrées}
    write('Donner le premier entier: ');
    readln(i1);
    write('Donner le second entier:');
    readln(i2);
    write('Donner le troisième entier: ');
    readln(i3);

  {Proposez votre code fonctionnel ici}

    {Affichage des sorties}
    writeln('le max est : ', maxi);
    readln();
End.
```

FIGURE 3.2 – Code source proposé par IDE4SCAPSS

Le code fonctionnel proposé par l'apprenant sera ensuite évalué par le sys-

tème d'analyse et de comparaison de la sémantique de codes sources présenté dans la section suivante.

Pour chaque problème, les experts ont proposé une solution experte composée du nombre d'entrées et de leur type, du nombre de sorties et de leur type, d'un code source expert décomposé en un code de déclaration des variables, un code de récupération des données d'entrée, un code d'affichage des données de sortie et d'un code fonctionnel expert.

Pour chaque problème proposé par les experts, nous retrouvons ainsi dans la base de données :

1. le libellé du problème (trois niveaux de difficulté) ;
2. le nombre de sorties et leur type ;
3. le nombre d'entrées et leur type ;
4. le code source de déclaration des variables ;
5. le code source de récupération des variables d'entrée ;
6. le code source d'affichage des variables de sortie ;
7. le code fonctionnel solution de l'expert ;
8. la valeur sémantique de chaque code fonctionnel expert est pré-calculée et conservée dans la base d'exercices. Le calcul de cette valeur sémantique fait l'objet de la section 3.3.

Nous avons dans le tableau 3.2 un exemple avec les données du problème de détermination du maximum de trois variables entières.

3.2.4 Conclusion

Le guidage de l'apprenant et l'explicitation dans une approche basée sur un scenario incrémentale a montré un impact positif sur les compétences de résolution de problèmes durant l'initiation à la programmation en présentiel. Les feedbacks sémantiques ont également un impact positif sur ces compétences. Ces aspects fondent le cadre théorique qui sous-tend la conception de notre outil. Cet outil vient en appui à l'apprenant dans le développement de ces compétences par une méthode de résolution de problèmes simple guidée par les données et intégrant la compréhension du problème. Les problèmes proposés couvrent les bases de l'initiation en faisant appel à la séquence, le test de contrôle et l'itération avec une version pour chaque niveau : débutant, intermédiaire et avancé. L'apprenant est ensuite guidé à travers un processus de résolution de problèmes en cinq (5) étapes.

TABLE 3.2 – Données du problème max de trois entiers dans la base de données

Libellés	voir table 3.1
nombre de sorties	1
types des sorties	integer
nombre d'entrées	3
types des entrées	integer, integer, integer

code de déclaration des variables	`{Sorties}`
	`Var maxi: integer;`
	`{Entrées}`
	`i1,i2,i3 :integer;`

code de récupération des entrées	`{lecture des entrées}`
	`write('Donner le premier entier: ');`
	`readln(i1);`
	`write('Donner le deuxième entier:');`
	`readln(i2);`
	`write('Donner le deuxième entier: ');`
	`readln(i3);`

code d'affichage des sorties	`{Affichage des sorties}`
	`writeln('le max est : ', maxi);`
	`readln();`

code fonctionnel	`{Code fonctionnel}`
	`if i1>i2 then`
	`  maxi:=i1`
	`else`
	`  maxi:=i2;`
	`if i3>maxi then`
	`maxi:=i3;`

Sémantique du code expert	$\begin{bmatrix} l1 > l2\ AND\ l1 > l3 & l1 \\ l2 > l1\ AND\ l2 > l3 & l2 \\ l3 \geq l2\ AND\ l3 \geq l1 & l3 \end{bmatrix}$

3.3 Analyse et comparaison sémantique de codes sources

3.3.1 Introduction

Notre système analyse et compare le code fonctionnel proposé par l'apprenant au code fonctionnel de l'expert ; le code de récupération des entrées et d'affichage des sorties étant donné par le système comme précisé plus haut. Notre système s'appuie sur les propriétés mathématiques des instructions introduites en cours d'initiation : l'assignation, la séquence, la condition et l'itération. A travers un système de calcul formel, il fixe une valeur sémantique initiale aux variables d'entrée et à travers un processus de chaînage, détermine la valeur sémantique que les variables de sortie devraient avoir à la fin de l'exécution du code fonctionnel.

Nous présenterons en premier les concepts de base dans la section 3.3.2. Nous exposerons ensuite le cas du calcul de la valeur sémantique d'un programme séquentiel dans la section 3.3.3. Dans la section 3.3.4, nous ferons une généralisation pour la prise en compte de la condition, le processus de calcul des valeurs sémantiques et les algorithmes sont introduits dans cette section. Nous terminerons par la conclusion.

3.3.2 Concepts de bases

Pour tout problème P, nous noterons S_P^e la solution proposée par l'expert constituée de :

1. $I_{S_P^e}$ l'ensemble des entrées proposées par l'expert ;
2. $O_{S_P^e}$ l'ensemble des sorties proposées par l'expert ;
3. le code de saisie des entrées $CI_{S_P^e}$
4. le code d'affichage des sorties $CO_{S_P^e}$
5. le code fonctionnel expert $CF_{S_P^e}$.

Une solution S_P proposée par un apprenant dans notre système est constituée de la reformulation R_{S_P}, des entrées I_{S_P}, des sorties O_{S_P} et du code fonctionnel CF_{S_P}. La solution n'est considérée correcte que si elle fournit respectivement les mêmes entrées et les mêmes sorties que celle de l'expert, I_P et O_P, et ensuite un code fonctionnel CF_{S_P} qui fournit le même résultat que celui de l'expert i.e. la même valeur sémantique pour l'ensemble des sorties.

Le code fonctionnel de l'apprenant est une suite d'instructions finie $CF_{S_P} = (I_n)_n$, I_n étant une instruction. Il est composé des instructions introduites

en cours d'initiation : l'assignation, la séquence, la condition et l'itération et d'expressions basées sur les types de bases (booléen, entier, réel, caractère) et les opérateurs qui leur sont applicables.

Nous noterons $SV(S_P)$ la sémantique de la solution qui n'est rien d'autre que la sémantique du code fonctionnel $SV(CF_{S_P})$.

Le code fonctionnel de l'apprenant est ainsi une suite d'instructions qui peut être vue comme une fonction qui devrait donner en sortie la valeur sémantique attendue des sorties et ayant en entrée la valeur sémantique initiale des données d'entrée :

$$CF_{S_P} : SV(I) \mapsto SV(O)$$

3.3.3 Cas de la séquence d'instructions

3.3.3.1 Définitions

Dans le cadre d'un programme séquentiel, la valeur finale des variables de sorties ne dépend d'aucune condition. Il est le résultat de l'exécution séquentielle du programme sans branchement et sans condition. Le code fonctionnel est alors une séquence d'assignations. $CF = (I_n)_n$, I_n assignation. Nous définissons les concepts suivants :

Definition 1 (Valeur sémantique initiale d'une variable d'entrée) *La valeur sémantique initiale $SV(i)$ d'une variable d'entrée i est un littéral l_i de même type que i fixé par notre système.*

Pour le problème de l'échange de deux variables entières x et y, nous aurons par exemple $SV(x) = l_x$ et $SV(y) = l_y$, l_x et l_y étant des littéraux entiers.

Pour le problème du calcul de la somme et de la moyenne de trois variables entières données par l'utilisateur au clavier. Les trois variables d'entrée seront initialisées par notre système avec des littéraux entiers :
$SV(i1) = l_1$; $SV(i2) = l_2$; $SV(i3) = l_3$.

Une assignation $V := Expr(V_1, \ldots, V_n)$ est une instruction qui associe à une variable V la valeur de l'expression $Expr$ située à droite de l'opérateur. Dans un programme séquentiel, les variables utilisées dans l'expression, $V_1, \ldots, V_n$, sont déjà initialisées soit en tant que variables d'entrée ou à travers des instructions d'affectation exécutées avant l'instruction en cours.

Definition 2 (Valeur sémantique d'une variable) *La valeur sémantique $SV(V)$ d'une variable V est la valeur calculée par chaînage des instructions*

du code fonctionnel et exprimée à partir de la sémantique des variables d'entrée.

Par exemple, pour deux variables x et y telles que $SV(x) = l_x$ et $SV(y) = l_y$ et une variable z. A la suite de l'instruction $z := x+y$, la valeur sémantique de z sera $SV(z) = l_x + l_y$.

Definition 3 (Valeur sémantique d'un programme) *Pour un problème P, la valeur sémantique d'une solution au problème S_P, de code fonctionnel CF_{S_P}, est égale à l'ensemble des valeurs sémantiques des variables de sorties du problème à la fin du programme.*

Par exemple, pour le problème du calcul de la somme et de la moyenne arithmétique de trois variables entières données par l'utilisateur. Si somme et moyenne sont les variables de sorties proposées par l'expert alors pour une proposition de solution S_P

$$SV(S_P) = \{SV_{S_P}(somme), SV_{S_P}(moyenne)\}$$

Definition 4 (équivalence sémantique de deux variables) *Deux variables $v1$ et $v2$ sont sémantiquement équivalentes et notées $v1 \equiv v2$ si*

$$SV(v1) = SV(v2)$$

.

Definition 5 (équivalence sémantique de deux programmes) *Deux programmes $S1$ et $S2$, proposés pour un problème P, sont sémantiquement équivalents et notés $S1 \equiv S2$ si $SV(S1) = SV(S2)$ i.e.*

$$\forall o \in O_P, SV_{S1}(o) \equiv SV_{S2}(o)$$

3.3.3.2 Processus de calcul

Une proposition de programme dans le cadre d'une séquence est constituée d'une séquence d'assignations.

Une affectation est une instruction qui fixe la valeur de la variable impactée par la valeur de l'expression qui lui est affectée. Une expression est une combinaison finie de symboles organisée selon des règles qui dépendent du contexte. Lors de l'initiation, les symboles peuvent désigner des constantes, des variables, des opérations arithmétiques, algébriques ou logiques et des regroupements pour déterminer l'ordre de priorité des opérations (parenthèses). Les opérateurs sont unaires ou binaires.

Notre système initialise les valeurs sémantiques des variables d'entrées et à travers un processus de chaînage effectue le calcul de la valeur sémantique des variables pour chaque instruction d'une proposition de programme.

Proposition 1 (Valeur sémantique d'une expression) *Considérons une variable V et n variables $V_1, \ldots, V_n$ telles que $SV(V_i)$ est défini et $Expr$ une expression dépendant des variables $V_1, \ldots, V_n$.*

1. *La valeur sémantique de l'expression $Expr(V_1, \ldots, V_n)$ est définie par :*

$$SV(Expr(V_1, \ldots, V_n)) = Expr(SV(V_1), \ldots, SV(V_n)))$$

2. *La valeur sémantique de la variable V impactée par une assignation I ayant pour membre droit $Expr$ est définie par :*

$$SV(V) = SV(Expr)$$

Preuve 1 *La première partie de la proposition est à démontrer pour les quatre types de base (entier, réel, booléen, caractère) et les opérateurs qui s'y appliquent mais nous nous limiterons à exposer ici la preuve pour le type entier puis nous la généraliserons pour les autres types de base. Nous ferons la preuve pour les symboles au sens large(constantes et variables), les variables étant des symboles.*

Nous allons montrer que la première partie de la proposition reste vraie pour une constante seule et une variable seule, pour l'application d'un opérateur unaire à une constante ou une variable et enfin faire une preuve par récurrence pour n variables (ou constantes) liées par des opérateurs binaires.

cas de la constante

Pour toute expression $Expr$ telle que l'expression est constituée d'un seul symbole s.
Si s est une constante entière de valeur C , la valeur sémantique de s est $SV(s) = C$
Après l'évaluation de l'expression $Expr$, on aura $SV(Expr) = C$, alors

$$SV(Expr) = SV(s)$$

cas d'une variable

Si s est une variable entière, telle que $SV(s) = E(l_1, \ldots, l_n)$, E étant une expression à résultat entier dépendant des littéraux $l_1, \ldots, l_n$.
Après l'évaluation de l'expression, on aura $SV(Expr) = E(l_1, \ldots, l_n)$, alors

$$SV(Expr(V)) = Expr(SV(V))$$

cas des opérateurs unaires

Pour toute expression $Expr$ telle que l'expression $Expr$ est constituée d'un symbole s et d'un opérateur unaire $*$ (les opérateurs unaires qui s'appliquent

aux entiers sont le - et le + unaires) : $Expr = *s$
si s est une constante entière de valeur C,
Après l'évaluation de l'expression $Expr$, on aura $SV(Expr) = *C$, alors

$$SV(Expr) = *SV(s)$$

si s est une variable entière telle que $SV(s) = E(l_1, \ldots, l_n)$, E étant une expression à résultat entier dépendant des littéraux $l_1, \ldots, l_n$.
Après l'évaluation de l'expression $Expr$, on aura $SV(Expr) = *E(l_1, \ldots, l_n)$, alors

$$SV(Expr(V)) = Expr(SV(V))$$

preuve par récurrence pour les opérateurs binaires

*Pour toute expression $Expr$ telle que l'expression est constituée de deux symboles $s1$ et $s2$ et d'un opérateur binaire * (les opérateurs binaires qui s'appliquent aux entiers sont la somme, le produit, la soustraction, la division entière, la division réelle et le modulo, on s'assurera que la division ne se fera pas par zero) : $Expr = s1 * s2$*
si $s1$ et $s2$ sont des constantes entières de valeur respective $C1$ et $C2$. Après l'évaluation de l'expression $Expr$, on aura $SV(Expr) = C1 * C2$, alors

$$SV(Expr) = SV(s1) * SV(s2)$$

si $s1$ est une constante entière de valeur $C1$ et $s2$ une variable entière V de valeur sémantique $E(l_1, \ldots, l_n)$, E étant une expression à résultat entier dépendant des littéraux $l_1, \ldots, l_n$.

Après l'évaluation de l'expression $Expr(V)$, on aura $SV(Expr(V)) = C1 * E(l_1, \ldots, l_n)$, alors

$$SV(Expr(V)) = C1 * SV(V) = Expr(SV(V))$$

si $s1$ et $s2$ sont deux variables entières,V_1 et V_2, de valeur sémantique respective $E(l_1, \ldots, l_n)$ et $E'(l'_1, \ldots, l'_m)$. Après l'évaluation de l'expression $Expr$, on aura $SV(Expr(V_1, V_2)) = E(l_1, \ldots, l_n) * E'(l'_1, \ldots, l'_m) = SV(s1) * SV(s2)$, alors

$$SV(Expr(V_1, V_2)) = Expr(SV(V_1), SV(V_2))$$

Supposons que la proposition est vraie pour $n-1$ opérateurs binaires et n symboles, des variables, liés par ces opérateurs dans une expression :

$$SV(Expr(V_1, \ldots, V_n)) = Expr(SV(V_1), \ldots, SV(V_n)))$$

Montrons qu'elle reste vraie pour n opérateurs binaires et $n+1$ symboles. Considérons l'expression $Expr(V_1, \ldots, V_{n+1})$. On peut la décomposer en une expression $Expr'(V_1, \ldots, V_n)$ à n symboles liés à un symbole V_{n+1} par un opérateur binaire $*$: $Expr(V_1, \ldots, V_{n+1}) = Expr'(V_1, \ldots, V_n) * V_{n+1}$.
On a ainsi :

$$\begin{aligned} SV(Expr(V_1, \ldots, V_{n+1})) &= SV(Expr'(V_1, \ldots, V_n) * V_{n+1}) \\ &= SV(Expr'(V_1, \ldots, V_n)) * SV(V_{n+1}) \\ &= Expr'(SV(V_1), \ldots, SV(V_n)) * SV(V_{n+1}) \\ &= Expr(SV(V_1), \ldots, SV(V_n), SV(V_{n+1})) \end{aligned}$$

alors

$$SV(Expr(V_1, \ldots, V_{n+1})) = Expr(SV(V_1), \ldots, SV(V_{n+1}))$$

Le principe de la preuve reste le même pour les autres types de base, nous pouvons ainsi généraliser cette propriété aux types de base et aux opérateurs qui s'y appliquent.

La deuxième partie de la proposition est évidente.

Cette proposition nous permet de calculer la valeur sémantique de toutes les variables par chaînage et remplacement des variables participant à une expression par leur valeur sémantique.

Exemple 1 *Prenons un exemple avec le code fonctionnel suivant pour le problème du calcul de la moyenne de trois variables entières données par l'utilisateur au clavier :*

$m := i_2 + i_1$;
$m := m + i_3$;
$m := m/3$;

Les trois variables d'entrée, i_1, i_2 et i_3 seront initialisées par notre système avec des littéraux entiers : $SV(i_1) = l_1$; $SV(i_2) = l_2$; $SV(i_3) = l_3$.

Pour la première instruction, nous aurons $SV(m) = SV(i_2) + SV(i_1) = l_2 + l_1$.

Pour la deuxième instruction, nous aurons $SV(m) = SV(m) + SV(i_3) = (l_2 + l_1) + l_3$.

Pour la dernière instruction, nous aurons $SV(m) = SV(m)/3 = (l_2 + l_1 + l_3)/3$.

Il faut cependant s'assurer de la conservation des propriétés des opérateurs dans les expressions pour valider par exemple que l'affectation $v := v1 + v2$ est équivalente sémantiquement à $v := v2 + v1$.

Proposition 2 (Propriétés des opérateurs) *Les opérateurs conservent leurs propriétés dans la valeur sémantique d'une expression.*

Preuve 2 *Considérons l'opérateur arithmétique + et une expression à deux variables $V1$ et $V2$ tel que $SV(V1) = l_1$ et $SV(V2) = l_2$, l_1 et l_2 étant des littéraux entiers.*
Nous avons $SV(V1 + V2) = l_1 + l_2$ et $SV(V2 + V1) = l_2 + l_1$
or $l_1 + l_2 = l_2 + l_1$ alors

$$SV(V1 + V2) = SV(V1) + SV(V2) = SV(V2) + SV(V1) = SV(V2 + V1)$$

Nous nous limitons ici à exposer la preuve pour la commutativité de l'addition. Elle se démontre de la même manière pour toutes les propriétés des opérateurs du langage et les types de données auxquels ils s'appliquent.

Ces deux propositions sont suffisantes pour nous permettre de calculer la valeur sémantique d'un programme à code fonctionnel séquentiel. Ils nous permettent également de vérifier l'équivalence sémantique entre deux variables donc entre deux programmes à code fonctionnel séquentiel.

Le cadre ainsi définit et le système de comparaison de code séquentiel à fait l'objet de nos travaux dans [80]. Il fallait cependant prendre en compte les autres structures de la programmation. Nous avons ainsi étendu ce travail dans [81] pour prendre en compte le cas des programmes à code fonctionnel conditionnel qui est l'objet de la section 3.3.4.

3.3.4 Généralisation à la structure conditionnelle

3.3.4.1 Définitions

Les concepts définis pour la séquence ne permettent pas de prendre en compte le cas d'un problème qui nécessite une solution conditionnelle. Pour la prise en compte de la condition nous avons généralisé les concepts introduits pour la séquence à la condition en utilisant une matrice comme valeur sémantique. Nous généralisons ici les concepts définis précédemment.

Definition 6 (Valeur sémantique initiale d'une donnée d'entrée) *Pour une variable d'entrée i, la valeur sémantique initiale $SV(i)$ est définie comme une matrice M[2*1] initialisée à [[ϕ]][l_i]], où l_i est un littéral de même type que i fixé par notre système et ϕ indique que l'expression sémantique l_i ne dépend d'aucune condition.*

Exemple 2 *Prenons l'exemple du calcul de la somme et de la moyenne de trois variables entières données par l'utilisateur au clavier. Les trois variables d'entrée seront initialisées par notre système avec des littéraux entiers :*
$SV(i_1) = [[\phi][l_1]]$ *;* $SV(i_2) = [[\phi][l_2]$ *;* $SV(i_3) = [[\phi][l_3]$.

Considérons un deuxième exemple avec le calcul du signe d'une variable entière x donnée par l'utilisateur. La valeur sémantique initiale de x sera initialisée par notre système :
$SV(x) = [[\phi][l]]$ *;*

Definition 7 (Valeur sémantique d'une variable) *La valeur sémantique d'une variable V(ou d'une expression E) , notée SV(V) (SV(E)), est la valeur de la variable exprimée en fonction des valeurs sémantiques des variables d'entrée. Elle est définie comme une matrice M[2*n] telle que :*

$$SV_{Prog}(V) = \begin{bmatrix} CE_1 & SE_1 \\ CE_2 & SE_2 \\ \dots & \dots \\ CE_n & SE_n \end{bmatrix}$$

- $CE_{Prog}(V) = [CE_1, \dots, CE_n]$, *appelé* ***vecteur des conditions****, est constitué d'expressions logiques booléennes que nous appellerons des* ***expressions conditionnelles*** *;*
- $SE_{Prog}(V) = [SE_1, \dots, SE_n]$, *appelé* ***vecteur d'expressions sémantiques****, est composé d'expressions dont chacune est dépendante de la condition de même ligne. Elles expriment la valeur sémantique de la variable relative à la condition de même ligne.*

Exemple 3 *Pour le calcul de la somme et de la moyenne de trois variables entières, si le code fonctionnel proposé par l'expert est le suivant :*
$somme := i_1 + i_2 + i_3$;
$moyenne := somme/3$;

alors la valeur sémantique des variables somme et moyenne à la fin de l'exécution du code fonctionnel expert est :
$SV(somme) = [[\phi][l_1 + l_2 + l_3]]$
$SV(moyenne) = [[\phi][(l_1 + l_2 + l_3)/3]]$
elles constitueront la valeur sémantique du programme. Les propositions des apprenants devraient avoir la même valeur sémantique.

Pour le cas de la détermination du signe d'une variable entière x, supposons que le code fonctionnel proposé par l'expert est le suivant :

```
if x>0 then
    r:='+'
else if x<0 then
    r:='-'
else
    r:='0';
```

La valeur sémantique du résultat, r, à la fin du code sera égale à :

$$SV(r) = \begin{bmatrix} l > 0 & '+' \\ l < 0 & '-' \\ l = 0 & '0' \end{bmatrix}$$

Cette valeur est celle qui sera attendue de toute proposition d'apprenant.

Remarque 1 (Valeur sémantique inconditionnelle) *Les variables d'entrée ne dépendent d'aucune condition autant que toutes les variables d'un programme séquentiel. Nous qualifierons leur valeur sémantique d'inconditionnelle.*

Remarque 2 (Optimisation des valeurs sémantiques) *Les valeurs sémantiques expriment la dépendance d'une expression sémantique par rapport à la condition de même ligne. Lorsque la même expression sémantique apparaît deux ou plusieurs fois dans le vecteur sémantique, nous procédons à un regroupement des lignes concernées en connectant les conditions par le "ou logique".*

Cela peut être lié à la nature complexe des conditions mais aussi à un code non optimal mais sémantiquement correct de l'apprenant.

Exemple 4 *Prenons l'exemple du cas de la détermination du signe d'une variable entière x, supposons que le code fonctionnel proposé par un apprenant soit le suivant :*

```
if x>5 then
    r:='+'
else if x<0 then
    r:='-'
else if x>0 then
    r:='+'
else
    r:='0'
```

La valeur sémantique du résultat, r, à la fin du code sera égale à :

$$SV(r) = \begin{bmatrix} l > 5 & '+' \\ l < 0 & '-' \\ l > 0 \ and \ l \leq 5 & '+' \\ l = 0 & '0' \end{bmatrix}$$

Le regroupement des lignes ayant pour valeur sémantique '+' avec pour condition $(l > 5 \ or \ (l > 0 \ and \ l \leq 5))$ *qui équivaut à* $l > 0$ *nous ramène à une matrice optimale :*

$$SV(r) = \begin{bmatrix} l > 0 & '+' \\ l < 0 & '-' \\ l = 0 & '0' \end{bmatrix}$$

Definition 8 (Équivalence sémantique de variables) *Deux variables* V *et* V' *tels que*

$$SV(V) = \begin{bmatrix} CE_1 & SE_1 \\ CE_2 & SE_2 \\ \dots & \dots \\ CE_n & SE_n \end{bmatrix} et \quad SV(V') = \begin{bmatrix} CE'_1 & SE'_1 \\ CE'_2 & SE'_2 \\ \dots & \dots \\ CE'_n & SE'_n \end{bmatrix}$$

sont sémantiquement équivalentes, noté $V \equiv V'$*, si :*
$\forall i \in [1-n], \exists! j \in [1-n] / SE_i = SE'_j \ and \ CE_i \equiv CE'_j$

La définition de l'équivalence de deux programmes (définition 5) est conservée telle qu'elle a été définie pour la séquence.

3.3.4.2 Processus de calcul et algorithmes

Nous rappelons que la construction syntaxique de la condition if en EBNF est :

<if-statement> : := if <expression> then <statement> [else <statement>]

Nous introduisons ici une première propriété liée à la construction syntaxique de la condition.

Propriété 1 *L'expression de la structure conditionnelle s'applique aux instructions du then et son complémentaire s'applique aux instructions du else. Au cas ou le else lui même imbrique un autre if, les instructions du deuxième if seront impactées par la conjonction du complémentaire de la condition du premier if avec la condition du second if. L'impact de la condition sur les instructions selon la construction syntaxique du if est ainsi donné par la table 3.3.*

TABLE 3.3 – Impact de la condition sur les instructions

Forme du if	Structure du If	Condition à appliquer
IF-THEN	if CE then	
	I	CE
IF-THEN	if CE_1 then	
-ELSE	I1	CE_1
	else	
	I2	$\overline{CE_1}$
IF-THEN	if CE_1 then	
-ELSE IF	I_1	CE_1
	else if CE_2 then	
	I_2	$\overline{CE_1}$ *and* CE_2
	else if CE_3 then	
	I_3	$\overline{CE_1}$ *and* $\overline{CE_2}$ *and* CE_3
	...	
	else if CE_{n-1} then	
	I_{n-1}	$\overline{CE_1}$ *and* ... $\overline{CE_{n-2}}$ *and* CE_{n-1}
	else	
	I_n	$\overline{CE_1}$ *and* ... $\overline{CE_{n-1}}$ *and* $\overline{CE_{n-1}}$

Une fois les conditions calculées, elles s'appliquent aux variables qui sont dans les blocs d'instructions relatives à ces conditions.

Il faudra noter que les conditions ont une valeur sémantique qui peut être inconditionnelle ou pas. Pour une condition à valeur sémantique inconditionnelle, l'application de cette condition à une variable, consiste à appliquer l'expression sémantique de la condition à la variable. Pour une condition dépendant d'expressions conditionnelles, les expressions sémantiques de la condition qui sont des expressions logiques, dépendent des expressions conditionnelles de la condition, l'application de cette condition à une variable, consiste à appliquer l'expression sémantique de la condition et (conjonction) l'expression conditionnelle dont elle dépend à la variable.

Nous introduisons ainsi une deuxième propriété relative à l'application d'une condition à une variable.

Propriété 2 *Considérons une condition CE et une variable V ayant pour valeur sémantique :*

$$SV(V) = \begin{bmatrix} CE_1 & SE_1 \\ CE_2 & SE_2 \\ \dots & \dots \\ CE_n & SE_n \end{bmatrix}$$

1. *Si CE a une valeur sémantique inconditionnelle, $SV(CE) = [[\phi][SE_{CE}]]$, alors la sémantique de l'application de CE à la variable V, que nous notons $App(CE, V)$ est :*

$$SV(App(CE, V)) = \begin{bmatrix} SE_{CE}\ and\ CE_1 & SE_1 \\ SE_{CE}\ and\ CE_2 & SE_2 \\ \dots & \dots \\ SE_{CE}\ and\ CE_n & SE_n \end{bmatrix}$$

2. *si CE a une valeur sémantique dépendant de conditions sous la forme :*

$$SV(CE) = \begin{bmatrix} CE'_1 & SE'_1 \\ CE'_2 & SE'_2 \\ \dots & \dots \\ CE'_m & SE'_m \end{bmatrix}$$

, alors

$$SV(App(CE, V)) = \begin{bmatrix} CE'_1\ and\ SE'_1\ and\ CE_1 & SE_1 \\ \dots & \dots \\ CE'_1\ and\ SE'_1\ and\ CE_n & SE_n \\ \dots & \dots \\ \dots & \dots \\ CE'_m\ and\ SE'_m\ and\ CE_1 & SE_1 \\ \dots & \dots \\ CE'_m\ and\ SE'_m\ and\ CE_n & SE_n \end{bmatrix}$$

Toute cette logique découle de la construction syntaxique de la structure conditionnelle if.

Nous introduisons ici l'algorithme 1 ($Combine(CE, V)$) tiré de la propriété 2. Il prend en entrée une condition et une variable et retourne la valeur sémantique de l'application de la condition à la variable.

Autant que pour la séquence, il est nécessaire de calculer la valeur sémantique des variables et expressions qui apparaissent dans les séquences

Algorithme 1 Combine (CE,V)

Entrées: CE : Condition, V : variable
Sorties: $SV(CE, V)$: sémantique de l'application de CE à V

si SV(CE)[1,1] =ϕ **alors**
  pour i=1 to len(SV(V)[1]) **faire**
    $SV(CE, V)[1, i] \leftarrow SV(CE)[2, 1]\ AND\ SV(V)[1, i]$ {condition}
    $SV(CE, V)[2, i] \leftarrow SV(V)[2, i]$ {expression sémantique}
  fin pour
sinon
  pour i=1 to len(SV(CE)[1]) **faire**
    pour j=1 to len(SV(V)[1]) **faire**
      $SV(CE, V)[1, i+j] \leftarrow SV(CE)[1, i]\ AND\ SV(CE)[2, i]\ AND\ SV(V)[1, j]$
      {condition}
      $SV(CE, V)[2, i + j] \leftarrow SV(V)[2, j]$ {expression sémantique}
    fin pour
  fin pour
fin si
return $SV(CE, V)$

d'instructions qui constituent les blocs d'instructions. il faut noter que les variables participant à une expression peuvent avoir une valeur sémantique conditionnelle ou inconditionnelle. Autant que pour les séquences, nous introduisons la proposition suivante :

Proposition 3 (Valeur sémantique d'une expression) *Considérons l'expression $Expr(V_1, ..., V_n)$ composée d'opérateurs et des variables $V_1, ..., V_n$ telle que la valeur sémantique de chacune de ces variables est définie par :*

$$SV(V_i) = \begin{bmatrix} CE_1^i & SE_1^i \\ CE_2^i & SE_2^i \\ \dots & \dots \\ CE_{m_i}^i & SE_{m_i}^i \end{bmatrix}$$

alors

$$SV(Expr(V_1, ..., V_n)) = \begin{bmatrix} CE_1^1\ and\ \dots\ and\ CE_1^n & Expr(SE_1^1, SE_1^2 \dots, SE_1^n) \\ \dots & \dots \\ CE_{m_1}^1\ and\ \dots\ and\ CE_1^n & Expr(SE_{m_1}^1, SE_1^2 \dots, SE_1^n) \\ \dots & \dots \\ \dots & \dots \\ CE_{i_1}^1\ and\ \dots\ and\ CE_{i_n}^n & Expr(SE_{i_1}^1, SE_{i_2}^2 \dots, SE_{i_n}^n) \\ \dots & \dots \\ \dots & \dots \\ CE_{m_1}^1\ and\ \dots\ and\ CE_{m_n}^n & Expr(SE_{m_1}^1, SE_{m_2}^2 \dots, SE_{m_n}^n) \end{bmatrix}$$

Preuve 3 *Concernant l'expression, la preuve de sa construction est la même que pour la séquence.*

Pour la construction de la matrice et de la condition, considérons que l'expression $Expr(V_1, V_2)$ *est constituée de deux variables* V_1 *et* V_2 *à valeur sémantique :*

$$SV(V_1) = \begin{bmatrix} CE_1 & SE_1 \\ CE_2 & SE_2 \\ \dots & \dots \\ CE_n & SE_n \end{bmatrix} et \quad SV(V_2) = \begin{bmatrix} CE'_1 & SE'_1 \\ CE'_2 & SE'_2 \\ \dots & \dots \\ CE'_m & SE'_m \end{bmatrix}$$

Les expressions sémantiques de $Expr(V_1, V_2)$, $Expr(SE_i, SE'_j)$ *dépendent en même temps de la condition* CE_i *et* CE'_j. *Le vecteur de conditions est donc le produit cartésien des deux vecteurs de conditions avec l'opérateur logique AND. Le vecteur d'expressions sémantiques est le produit cartésien des deux vecteurs d'expressions sémantiques auquel il est appliqué l'expression. Nous avons ainsi :*

$$SV(Expr(V_1, V_2)) = \begin{bmatrix} CE_1 \ AND \ CE'_1 & Expr(SE_1, SE'_1) \\ CE_1 \ AND \ CE'_2 & Expr(SE_1, SE'_2) \\ \dots & \dots \\ CE_n \ AND \ CE'_m & Expr(SE_n, SE'_m) \end{bmatrix}$$

Supposons que la proposition est vraie pour $n-1$ *variables* $V_1, ..., V_{n-1}$ *:*

$$SV(Expr(V_1, ..., V_{n-1})) = \begin{bmatrix} CE^1_1 \ and \ \dots \ and \ CE^{n-1}_1 & Expr(SE^1_1, SE^2_1 \dots, SE^{n-1}_1) \\ \dots & \dots \\ CE^1_{m_1} \ and \ \dots \ and \ CE^{n-1}_1 & Expr(SE^1_{m_1}, SE^2_1 \dots, SE^{n-1}_1) \\ \dots & \dots \\ \dots & \dots \\ CE^1_{i_1} \ and \ \dots \ and \ CE^{n-1}_{i_{n-1}} & Expr(SE^1_{i_1}, SE^2_{i_2} \dots, SE^{n-1}_{i_{n-1}}) \\ \dots & \dots \\ \dots & \dots \\ CE^1_{m_1} \ and \ \dots \ and \ CE^{n-1}_{m_{n-1}} & Expr(SE^1_{m_1}, SE^2_{m_2} \dots, SE^{n-1}_{m_{n-1}}) \end{bmatrix}$$

Démontrons la pour n *variables. Considérons la variable*

$$SV(V_n) = \begin{bmatrix} CE^n_1 & SE^n_1 \\ CE^n_2 & SE^n_2 \\ \dots & \dots \\ CE^n_{m_n} & SE^n_{m_n} \end{bmatrix}$$

Comme pour la séquence on aura

*$Expr(V_1, ..., V_n) = Expr'(V_1, ..., V_{n-1}) * V_n$ ou * est un opérateur binaire. Le vecteur de conditions est donc le produit cartésien du vecteur de conditions de $Expr'$ et de celui de V_n avec l'opérateur logique AND. Le vecteur d'expressions sémantiques est le produit cartésien du vecteur d'expression sémantique de $Expr'$ et de celui de V_n auquel il est appliqué l'expression. On aura donc*

$$SV(Expr(V_1, ..., V_n)) = \begin{bmatrix} CE_1^1 \text{ and } \dots \text{ and } CE_1^n & Expr(SE_1^1, SE_1^2 \dots, SE_1^n) \\ \dots & \dots \\ CE_{m_1}^1 \text{ and } \dots \text{ and } CE_1^n & Expr(SE_{m_1}^1, SE_1^2 \dots, SE_1^n) \\ \dots & \dots \\ \dots & \dots \\ CE_{i_1}^1 \text{ and } \dots \text{ and } CE_{i_n}^n & Expr(SE_{i_1}^1, SE_{i_2}^2 \dots, SE_{i_n}^n) \\ \dots & \dots \\ \dots & \dots \\ CE_{m_1}^1 \text{ and } \dots \text{ and } CE_{m_n}^n & Expr(SE_{m_1}^1, SE_{m_2}^2 \dots, SE_{m_n}^n) \end{bmatrix}$$

Nous introduisons ici l'algorithme 2 qui permet de calculer la valeur sémantique d'une expression. Il est basée sur la proposition 3.

Algorithme 2 SemanticExpression SV($Expr(V1, \dots, Vn)$)

Entrées: $Expr$: Expression, $V1, \dots, Vn$: variables participant à l'expression.

Sorties: $SV(Expr)$: sémantique de $Expr(V1, \dots, Vn)$

```
SVTemp ← φ
SVFinal ← SV(V1)
taille ← len(SVFinal)
pour i=2 to n faire
  pour j=1 to taille faire
    pour k=1 to len(V[i]) faire
      SVTemp[1, j+k] ← SVFinal[1, j] AND SV(Vi)[1, k] {condition}
      SVTemp[2, j+k] ← SVFinal[2, j], SV(Vi)[2, k] {paramètres de
      l'expression sémantique}
    fin pour
  fin pour
  SVFinal ← SVTemp
  taille ← len(SVFinal)
  SVTemp ← φ
fin pour
SVFinal ← [SVFinal[1], Expr(SVFinal[2])]
return SVFinal
```

Les définitions, propriétés et propositions définies en amont sont suffisantes pour le calcul des **valeurs sémantiques des variables d'un programme**. Elles permettent donc de calculer la sémantique d'un programme.

L'algorithme 3 *SemanticProgram* permet de calculer la sémantique d'un code fonctionnel, donc celle d'un programme.

Algorithme 3 SemanticProgram $SV(Op)$

Entrées: $SV(Ip)$: Sémantique des Entrées
Op : Sorties
FC : Code fonctionnel
Sorties: $SV(Op)$: sémantique des données de sortie
$AST \leftarrow AST(FC)$ {transformation en AST}
$Conditions \leftarrow \phi$ {liste des conditions à appliquer}
pour tout Noeud N in AST **faire**
si type(N) =IfStatement **alors**
$CE \leftarrow Condition(N))$
$Conditions.add(CE)$
sinon si type(N) =ElseStatement **alors**
$CE \leftarrow Conditions.last()$ {récupérer le dernier élément}
$Conditions.replacelast(\overline{CE})$ {remplacer le dernier élément par son complémentaire}
sinon si type(N) =EndIfStatement **alors**
$Conditions.pop()$ {enlever le dernier élément}
sinon si type(N) =Assignation **alors**
$V \leftarrow variable(N)$
$Expr \leftarrow Expr(N)$
$SV\_Expr \leftarrow SemanticExpression(Expr)$
$SV(V) \leftarrow Combine(Conditions, SV\_Expr)$
fin si
fin pour
return $\{SV(V), V \in Op\}$

Dans une première étape, nous transformons le code source en arbre de syntaxe abstraite (AST) qui ne représente pas les nœuds et les branches qui n'affectent pas la sémantique d'un programme.

L'algorithme fait ensuite un parcours en profondeur de l'arbre, il permet de construire les conditions à appliquer aux expressions et variables (conf. propriété 1). Les conditions sont ensuite appliquées aux instructions du bloc concerné en faisant appel à deux autres routines que nous avons définies plus haut : *Combine* (algorithme 1) et *SemanticExpression* (algorithme 2).

Les méthodes suivantes sont également utilisées :

— $Var(N)$ qui permet de récupérer la variable d'une assignation ;
— $Expr(N)$ qui permet de récupérer l'expression de droite d'une assignation ;

— $Condition(N)$ qui permet de récupérer l'expression conditionnelle d'une structure if.

3.3.5 Conclusion

L'analyse et la comparaison sémantique de code source permettent de mettre en place un système de feedback basé sur la sémantique du code. Les feedbacks sur la sémantique du code ont montré un impact positif sur l'apprentissage, et particulièrement sur les compétences de résolution de problèmes. Le système que nous avons présenté ici s'appuie sur les propriétés des instructions. Nous utilisons des mathématiques symboliques et à travers du calcul formel nous déterminons la valeur de sortie des programmes.

L'avantage d'un tel système est de nécessiter moins d'effort de la part des formateurs que les systèmes basés sur une approche dynamique ou une approche statique manuelle. Il offre également une meilleure précision que les systèmes automatiques d'analyse et de comparaison sémantique basés sur l'apprentissage automatique.

3.4 Implémentation

3.4.1 Introduction

Dans cette partie, nous allons présenter l'implémentation du prototype de notre système, IDE4SCAPSS. Nous aborderons les choix architecturaux qui ont été fait dans la section 3.4.2, puis les tests expérimentaux effectués avec des experts et une simulation dans la section 3.4.3 avant de conclure.

3.4.2 Architecture du système

Nous présentons dans le schéma 3.3 l'architecture de notre système. L'élément central du système est le moteur d'analyse et de comparaison sémantique qui pré-calcule les valeurs sémantiques des solutions expertes et les conserve dans la base de données. Le calcul de la valeur sémantique des soumissions d'apprenants se fait à l'évaluation après une compilation classique.

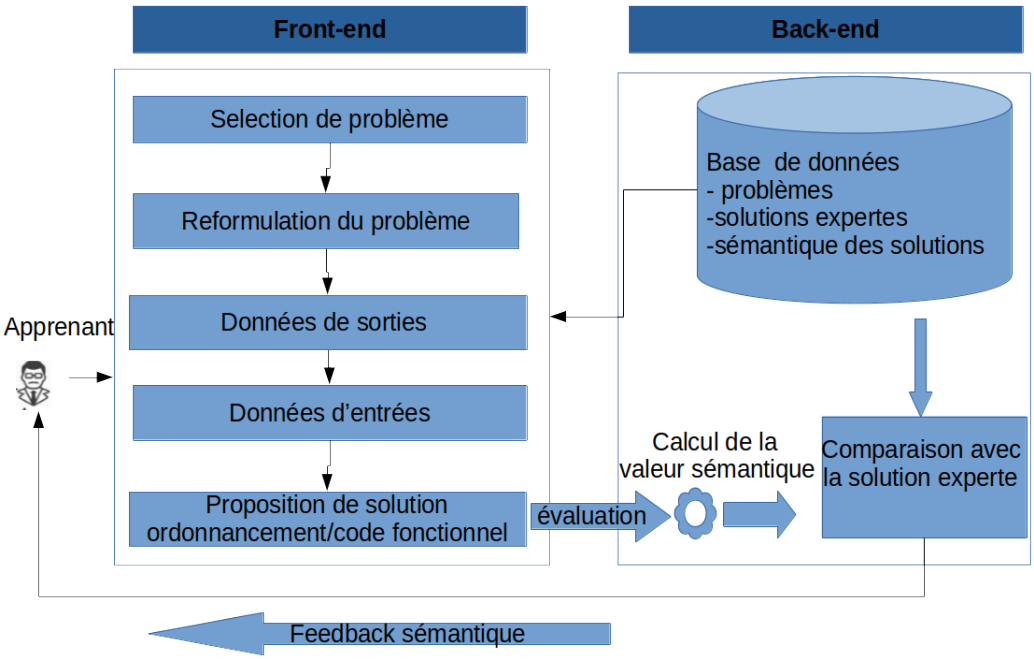

FIGURE 3.3 – Architecture du système

L'apprenant suit les différentes étapes proposées par l'outil :

1. reformulation du problème ;

2. proposition des sorties et de leur type ;
3. proposition des entrées et de leur type ;
4. proposition de solution : il peut choisir d'ordonnancer le code source expert ou de proposer un code fonctionnel ;
5. évaluation de la solution.

   Pour l'ordonnancement, le système évalue l'ordre proposé des instructions. Pour la proposition de code fonctionnel, le système effectue une compilation classique du code avant de faire une analyse de la sémantique du code source et une comparaison avec la solution experte.

Le fonctionnement du processus d'évaluation est décrit dans le schéma 3.4.

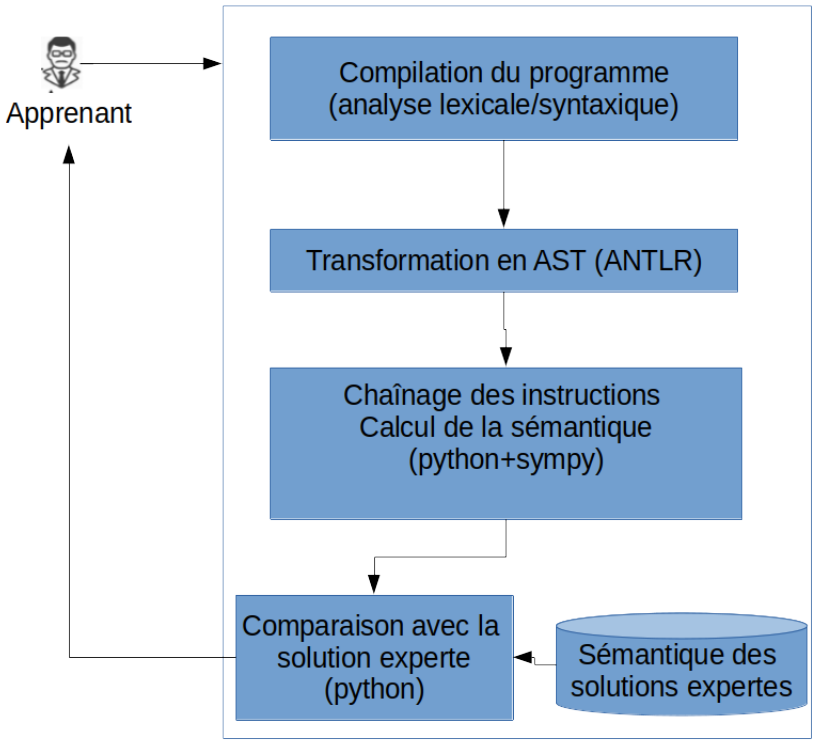

FIGURE 3.4 – Processus d'évaluation d'une proposition sur IDE4SCAPSS

Une compilation classique est effectuée avant d'effectuer une analyse sémantique et une comparaison avec la solution experte. Nous rappelons que

notre cadre d'application est le langage Pascal qui est utilisé dans certaines de nos formations comme langage d'initiation.

Notre implémentation a été faite avec le langage python qui reste beaucoup plus professionnel et offre un ensemble de paquets qui ont facilité notre implémentation.

Le système génère l'arbre de syntaxe abstraite du code source. Nous nous appuyons sur le paquet ANTLR (www.antlr.org) qui est un framework libre de construction de compilateurs. Le système parcourt en profondeur l'arbre et fait un chaînage des instructions pour construire la sémantique des variables. Toutes les expressions sémantiques sont converties pour faciliter la comparaison en nous appuyant sur le paquet Sympy (www.sympy.org), une librairie python pour les mathématiques symboliques et le calcul algébrique.

Le comparateur est un module autonome. Cette architecture offre l'avantage de rendre accessible le système en tant qu'application autonome mais aussi en mode web.

3.4.3 Expérimentation et simulation

Notre système étant basé sur du calcul formel, l'analyse et la comparaison sémantique de code source sur des exemples de programmes est toujours exacte. Les experts ont testé l'outil sur les solutions expertes des problèmes qui constituent la base et sur des solutions erronées avec des erreurs sémantiques classiques chez les apprenants telles que l'écrasement de variable. Les codes sources sémantiquement corrects sont toujours évalués corrects par le système d'analyse. Les codes sources ayant des erreurs sémantiques sur le résultat également sont toujours évalués incorrects. Cet aspect a été évalué sur les problèmes à solution basée sur les structures séquentielles et conditionnelles de la base de problèmes (Annexe A).

Nous allons montrer une simulation sur deux exemples. Le premier exemple est le problème de l'échange de deux variables, $P1$ qui est un programme séquentiel. Le second, $P2$, est le problème du calcul du signe du produit de deux variables entières sans faire l'opération qui est un problème conditionnel.

Pour le premier problème, la valeur sémantique des variables x et y est initialisé par notre système par deux littéraux entiers :
$SV(x) = [[\phi][l_x]]$; $SV(y) = [[\phi][l_y]]$;

La solution experte dans la base de données est :
t :=x ;
x :=y ;

y :=t ;

Nous donnons dans la table 3.4 le processus de calcul des valeurs sémantiques pour la solution experte S^e_{P1}.

TABLE 3.4 – Calcul de la sémantique de l'échange de variables - expert

Ligne	Code	Calcul de la sémantique
1	t :=x ;	SV(t)=SV(x)=[[ϕ]][l_x]]
2	x :=y ;	SV(x)=SV(y)=[[ϕ]][l_y]]
3	y :=t ;	SV(y)=SV(t)=[[ϕ]][l_x]]

$SV(S^e_{P1}) = \{SV(x) = [[\phi][l_y]], SV(y) = [[\phi][l_x]]\}$

Considérons deux solutions d'apprenants, respectivement S1 et S2 :

t :=y ;

y :=x ;

x :=t ;

et

x :=y ;

y :=x ;

Le processus de calcul des valeurs sémantiques des deux solutions est donné dans le tableau 3.5.

TABLE 3.5 – Calcul de la sémantique de l'échange de variables - apprenant

Solution	Ligne	Code	Calcul de la sémantique
S1	1	t :=y ;	SV(t)=SV(y)=[[ϕ]][l_y]]
	2	y :=x ;	SV(y)=SV(x)=[[ϕ]][l_x]]
	3	x :=t ;	SV(x)=SV(t)=[[ϕ]][l_y]]
S2	1	x :=y ;	SV(x)=SV(y)=[[ϕ]][l_y]]
	2	y :=x ;	SV(y)=SV(x)=[[ϕ]][l_y]]

La sémantique des deux programmes est la suivante :

SV(S1)={SV(x)=[[ϕ]][l_y]], SV(y)=[[ϕ]][l_x]]}

SV(S2)={SV(x)=[[ϕ]][l_y]], SV(y)=[[ϕ]][l_y]]}.

S1 a la même sémantique que le programme expert tandis que S2 a une sémantique différente parce que l'apprenant a fait une erreur d'écrasement de variable.

Pour le problème $P2$, la valeur sémantique des variables x et y est également initialisée par notre système par deux littéraux entiers :

$SV(x) = [[\phi][l_x]]$; $SV(y) = [[\phi][l_y]$;

TABLE 3.6 – Calcul de la sémantique du signe du produit - expert

Ligne	Code foncionnel	Calcul de la sémantique
1	if $(x > 0)$ and $(y > 0)$ then	
2	r :='+' ;	[[C1]['+']]
3	else if $(x < 0)$ and $(y < 0)$ then	
4	r :='+' ;	$\begin{bmatrix} C1 & '+' \\ \overline{C1}\ and\ C2 & '+' \end{bmatrix}$
5	else if $(x > 0)$ and $(y < 0)$ then	
6	r :='-' ;	$\begin{bmatrix} C1 & '+' \\ \overline{C1}\ and\ C2 & '+' \\ \overline{C1}\ and\ \overline{C2}\ and\ C3 & '-' \end{bmatrix}$
7	else if $(x < 0)$ and $(y > 0)$ then	
8	r :='-' ;	$\begin{bmatrix} C1 & '+' \\ \overline{C1}\ and\ C2 & '+' \\ \overline{C1}\ and\ \overline{C2}\ and\ C3 & '-' \\ \overline{C1}\ and\ \overline{C2}\ and\ \overline{C3}\ and\ C4 & '-' \end{bmatrix}$
9	else	
10	r :='0' ;	$\begin{bmatrix} C1 & '+' \\ \overline{C1}\ and\ C2 & '+' \\ \overline{C1}\ and\ \overline{C2}\ and\ C3 & '-' \\ \overline{C1}\ and\ \overline{C2}\ and\ \overline{C3}\ and\ C4 & '-' \\ \overline{C1}\ and\ \overline{C2}\ and\ \overline{C3}\ and\ \overline{C4} & '0' \end{bmatrix}$

nous avons une solution experte et le calcul de sa valeur sémantique dans le tableau 3.6.

$C1 = (l_x > 0)\ and\ (l_y > 0)$ - $C2 = (l_x < 0)\ and\ (l_y < 0)$

$C3 = (l_x > 0)\ and\ (l_y < 0)$ - $C4 = (l_x < 0)\ and\ (l_y > 0)$

Les différentes combinaisons de conditions seront simplifiées par du calcul formel avec les propriétés logiques des expressions. Nous nous limitons à donner ici deux exemples :

$$\begin{aligned} \overline{C1}\ and\ C2 &= \overline{(l_x > 0)\ and\ (l_y > 0)}\ and\ ((l_x < 0)\ and\ (l_y < 0)) \\ &= ((l_x \leq 0)\ or\ (l_y \leq 0))\ and\ ((l_x < 0)\ and\ (l_y < 0)) \\ &= (l_x < 0)\ and\ (l_y < 0) \end{aligned}$$

$$\begin{aligned}
\overline{C1}\ and\ \overline{C2}\ and\ \overline{C3}\ and\ \overline{C4} &= \overline{(l_x > 0)\ and\ (l_y > 0)}\ and\ \overline{((l_x < 0)\ and\ (l_y < 0))}\ and \\
&\quad \overline{(l_x > 0)\ and\ (l_y > 0)}\ and\ \overline{((l_x < 0)\ and\ (l_y < 0))} \\
&= ((l_x \leq 0)\ or\ (l_y \leq 0))\ and\ ((l_x \geq 0)\ or\ (l_y \geq 0))\ and \\
&\quad ((l_x \leq 0)\ or\ (l_y \geq 0))\ and\ ((l_x \geq 0)\ or\ (l_y \leq 0)) \\
&= ((l_x = 0)\ or\ (l_y \leq 0))\ and\ ((l_x = 0)\ or\ (l_y \geq 0)) \\
&= (l_x = 0)\ or\ (l_y = 0)
\end{aligned}$$

Toutes simplifications faites, la valeur sémantique de r est donnée par :

$$SV(r) = \begin{bmatrix} (l_x > 0)\ and\ (l_y > 0) & '+' \\ (l_x < 0)\ and\ (l_y < 0) & '+' \\ (l_x > 0)\ and\ (l_y < 0) & '-' \\ (l_x < 0)\ and\ (l_y > 0) & '-' \\ (l_x = 0)\ or\ (l_y = 0) & '0' \end{bmatrix}$$

Après la phase d'optimisation, la valeur sémantique du programme sera :

$$\text{SV(P1)=}\left\{\text{SV(r)=}\begin{bmatrix} ((l_x > 0)\ and\ (l_y > 0))\ or\ ((l_x < 0)\ and\ (l_y < 0)) & '+' \\ ((l_x > 0)\ and\ (l_y < 0))\ or\ ((l_x < 0)\ and\ (l_y > 0)) & '-' \\ (l_x = 0)\ or\ (l_y = 0) & '0' \end{bmatrix}\right\}$$

Toute proposition d'apprenants est évaluée et comparée à cette valeur.

3.4.4 Conclusion

Nous avons présenté dans cette section l'architecture et l'implémentation de notre approche de soutien au développement des compétences de résolution de problèmes durant l'initiation à la programmation.

Le développement du prototype a été fait avec le langage python pour le guidage et l'analyse et la comparaison de la sémantique de codes sources Pascal. Le moteur d'analyse est autonome, l'application est accessible en mode web ou comme outil autonome.

Des tests ont été effectués sur les solutions expertes et des solutions erronées avec des erreurs sémantiques classiques et une simulation sur deux exemples de problèmes a été présentée.

3.5 Conclusion

Dans ce chapitre nous avons présenté les contributions de cette thèse. Nous avons exposé la stratégie de guidage de l'apprenant proposée et le cadre théorique sur lequel se fondent nos travaux. nous sommes ensuite revenus sur le système d'analyse et de comparaison de la sémantique de codes sources et enfin sur l'implémentation du système.

L'approche de guidage que nous avons adoptée s'appuie sur une méthode simple guidée par les données et qui intègre l'aspect compréhension du problème. Le guidage de l'apprenant dans le processus de résolution de problèmes et l'explicitation dans un scénario incrémental accompagné de feedbacks génériques mais aussi sur la sémantique du code source est le cadre qui sous-tend nos travaux. Ces stratégies ont montré un impact positif sur les compétences de résolution de problèmes.

La méthode d'analyse et de comparaison sémantique de codes sources que nous proposons est basée sur une approche automatique mais plus particulièrement sur du calcul formel et des mathématiques symboliques. Elle se distingue ainsi des méthodes statiques manuelles et des méthodes dynamiques qui nécessitent beaucoup d'efforts de la part des formateurs. Elle se distingue aussi des méthodes automatiques que nous avons rencontrées qui sont basées sur des méthodes d'apprentissage automatique avec des taux de precision assez faibles.

Le prototype de IDE4SCAPSS a été implémenté avec le langage python. il est accessible en mode web ou comme application autonome et donne accès aux fonctionnalités de guidage de l'apprenant et d'analyse et de comparaison de la sémantique de codes sources. Le moteur d'analyse et de comparaison sémantique est l'élément central du système, il a été testé par des experts et fournit une évaluation correcte de la sémantique d'un code source séquentiel et conditionnel.

Chapitre 4

Conclusion et perspectives

Dans ce chapitre, nous allons d'abord présenter les différentes contributions de cette thèse. Nous allons ensuite exposer les limites de nos propositions et proposer des perspectives.

4.1 Résumé des contributions

Dans cette thèse, nous avons proposé une approche pour le développement des compétences de résolution de problèmes durant l'initiation à l'algorithmique et à la programmation [79, 81, 80].

Dans un premier temps, nous avons effectué une revue des approches de soutien au développement des compétences de résolution de problèmes durant l'initiation à la programmation.

Nous avons ensuite proposé une approche qui s'appuie sur deux stratégies :

1. le guidage de l'apprenant et l'explicitation des concepts dans le processus de conception de programmes.
2. l'analyse et la comparaison de la sémantique de codes sources de l'apprenant avec des codes experts afin de lui proposer un feedback sémantique.

Nous avons d'abord introduit une stratégie de guidage de l'apprenant en quatre (4) étapes dans [79]. Cette stratégie est guidée par les données avec une activité de proposition de solution par ordonnancement d'un code source expert mélangé.

Cette proposition a été étendue dans [80] pour prendre en compte le fait que la compréhension du problème est centrale et critique dans le processus de conception de programmes. Cela se fait par l'activité de reformulation de problèmes. Nous avons ainsi proposé une stratégie de guidage de l'apprenant simple qui s'appuie sur une méthode de résolution de problèmes en cinq (5) étapes :

1. reformulation du problème ;
2. détermination des sorties et de leur type ;
3. détermination des entrées et de leur type ;
4. proposition de solution par ordonnancement de code expert ou par proposition de code fonctionnel
5. évaluation de la solution par comparaison sémantique.

La stratégie proposée couvre des problèmes faisant appel à l'usage des structures séquentielles, conditionnelles et itératives avec des niveaux de difficulté différents.

Nous avons ensuite proposé une approche automatique d'analyse et de comparaison sémantique de codes sources basée sur le calcul formel [81]. Dans ce système le formateur ne fournit que les solutions expertes. Il nécessite ainsi moins d'efforts de la part du formateur que les systèmes basés sur une approche dynamique ou ceux utilisant une approche statique manuelle.

Nous avons implémenté un prototype du système nommé IDE4SCAPSS pour le guidage, l'analyse et la comparaison sémantique de programmes codés en Pascal. Le système a été évalué sur des solutions expertes de problèmes utilisant des structures séquentielles et/ou conditionnelles.

4.2 Limites et Perspectives

Dans cette section, nous discutons des limites de notre travail et de quelques unes de ses perspectives.

L'approche d'analyse et de comparaison de la sémantique de codes sources proposée reste intéressante au sens ou elle s'appuie sur les mathématiques symboliques et le calcul formel. Toutefois, elle ne prend en compte que les structures séquentielles et/ou conditionnelles. Certes, les problèmes à solution séquentielle ou conditionnelle sont les premiers abordés durant l'initiation à la programmation. Mais la prise en compte de ceux à solution itérative permettrait de couvrir tous les types de problèmes. Nous envisageons d'étendre cette approche pour la prise en compte des problèmes à solution itérative.

Notre approche a pour cadre d'application le langage Pascal qui est utilisé dans certaines de nos institutions pour le cours d'initiation de la programmation. L'extension de l'approche aux autres langages de programmation est une piste à exploiter particulièrement pour le langage C qui est encore très utilisé en initiation dans notre pays mais aussi pour le langage Python qui est en train de monter.

Le système d'analyse et de comparaison de la sémantique de codes sources a été évalué sur des solutions proposées par des experts intégrant des bonnes et mauvaises solutions. L'évaluation de l'outil dans un contexte d'apprentissage réel est envisagé. Cette évaluation portera aussi bien sur l'utilisabilité, la convivialité, la robustesse et la performance de l'outil mais surtout son impact réel sur les compétences de résolution de problèmes.

A l'état actuel, notre système est capable d'analyser un programme Pascal et d'indiquer à l'apprenant si la solution proposée est sémantiquement

correcte ou fausse. Nous comptons améliorer les feedbacks pour prendre en compte les solutions partiellement correctes. C'est le cas lorsque pour un problème à trois sorties, la sémantique des deux sorties est correcte et celle de la troisième fausse.

L'exploitation des traces d'apprentissage sur l'outil est une piste à considérer d'une part pour l'analyse des processus suivis par les apprenants et d'autre part pour l'amélioration du système et des feedbacks.

Bibliographie

[1] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. Generation CS : the growth of computer science. *ACM Inroads*, 8(2) :44–50, May 2017.

[2] Susan Einhorn. Microworlds, computational thinking, and 21st century learning. *LCSI White Paper*, 2012.

[3] Janine Rogalski, Renan Samurçay, and J.-M. Hoc. L'apprentissage des méthodes de programmation comme méthodes de résolution de problème. *Le travail humain*, pages 309–320, 1988. Publisher : JSTOR.

[4] William F. Atchison, Samuel D. Conte, John W. Hamblen, Thomas E. Hull, Thomas A. Keenan, William B. Kehl, Edward J. McCluskey, Silvio O. Navarro, Werner C. Rheinboldt, Earl J. Schweppe, William Viavant, and David M. Young. Curriculum 68 : Recommendations for academic programs in computer science : a report of the ACM curriculum committee on computer science. *Communications of the ACM*, 11(3) :151–197, March 1968.

[5] Richard H. Austing, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes. Curriculum '78 : recommendations for the undergraduate program in computer science - a report of the ACM curriculum committee on computer science. *Communications of the ACM*, 22(3) :147–166, March 1979.

[6] Elliot B. Koffman, Philip L. Miller, and Caroline E. Wardle. Recommended curriculum for CS1, 1984. *Communications of the ACM*, 27(10) :998–1001, October 1984.

[7] Sónia Rolland Sobral. CS1 and CS2 curriculum recommendations : learning from the past to try not to rediscover the wheel again. In *World Conference on Information Systems and Technologies*, pages 182–191. Springer, 2020.

[8] Matthew Hertz. What do" CS1" and" CS2" mean ? Investigating differences in the early courses. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 199–203, 2010.

[9] Allen B. Tucker. Computing Curricula 1991. *Communications of the ACM*, 34(6) :68–84, June 1991.

[10] Carl Chang, Peter J Denning, Gerald Engel, Robert Sloan, Doris Carver, Richard Eckhouse, Willis King, Francis Lau, Susan Mengel, Pradip Srimani, Eric Roberts, Russell Shackelford, Richard Austing, C Fay Cover, Gordon Davies, Andrew McGettrick, G Michael Schneider, and Ursula Wolz. Composition of the Curriculum 2001 Joint Task Force. page 240.

[11] Lillian Cassel, Alan Clements, Gordon Davies, Mark Guzdial, Renée McCauley, Andrew McGettrick, Bob Sloan, Larry Snyder, Paul Tymann, and Bruce W. Weide. Computer Science Curriculum 2008 : An Interim Revision of CS 2001. Technical Report, Association for Computing Machinery, New York, NY, USA, 2008. Num Pages : 108.

[12] ACM Computing Curricula Task Force, editor. *Computer Science Curricula 2013 : Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* ACM, Inc, January 2013.

[13] Sónia Rolland Sobral. 30 years of cs1 : Programming languages evolution. In *ICERI2019 Proceedings*, 12th annual International Conference of Education, Research and Innovation, pages 9197–9205. IATED, 11-13 November, 2019 2019.

[14] Etienne Vandeputa and Julie Henryb. Apprendre à programmer Comment les enseignants justifient-ils le choix d'un outil didactique ? *De 0 à 1 ou l'heure de l'informatique à l'école*, page 325, 2018.

[15] Edsger W. Dijkstra. Letters to the editor : go to statement considered harmful. *Communications of the ACM*, 11(3) :147–148, 1968. Publisher : ACM New York, NY, USA.

[16] Onyeka Ezenwoye. What language ?-The choice of an introductory programming language. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2018.

[17] Brett A. Becker and Keith Quille. 50 years of cs1 at sigcse : A review of the evolution of introductory programming education research. In *Proceedings of the 50th acm technical symposium on computer science education*, pages 338–344, 2019.

[18] Sónia Rolland Sobral. Cs1 : C, java or python ? tips for a conscious choice. In *ICERI2019 Proceedings*, 12th annual International Conference of Education, Research and Innovation, pages 2512–2519. IATED, 11-13 November, 2019 2019.

[19] Veljko Aleksić and Mirjana Ivanović. Introductory programming subject in European higher education. *Informatics in Education*, 15(2) :163–182, 2016. Publisher : Vilnius University Institute of Data Science and Digital Technologies.

[20] Sónia Rolland Sobral. The first programming language and freshman year in computer science : characterization and tips for better decision making. In *World Conference on Information Systems and Technologies*, pages 162–174. Springer, 2020.

[21] Po-Yao Chao. Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Computers & Education*, 95 :202–215, 2016.

[22] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *AcM SIGcSE Bulletin*, 39(2) :32–36, 2007. Publisher : ACM New York, NY, USA.

[23] Christopher Watson and Frederick WB Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44, 2014.

[24] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming : 12 years later. *ACM inroads*, 10(2) :30–36, 2019. Publisher : ACM New York, NY, USA.

[25] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, (99) :1–14, 2018.

[26] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory Programming : A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion, pages 55–106, New York, NY, USA, 2018. ACM. event-place : Larnaca, Cyprus.

[27] Elliot Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9) :850–858, 1986.

[28] Shahira Popat and Louise Starkey. Learning to code or coding to learn ? A systematic review. *Computers & Education*, 128 :365–376, 2019.

[29] Joseph A. Lyon and Alejandra J. Magana. Computational thinking in higher education : A review of the literature. *Computer Applications*

in Engineering Education, 28(5) :1174–1189, 2020. Publisher : Wiley Online Library.

[30] Peng Chen, Yang Tian, Wei Zhou, and Ronghuai Huang. A systematic review of computational thinking : Analysing research hot spots and trends by CiteSpace. In *Proceedings of the 26th International Conference on Computers in Education, Manila, The Philppines*, pages 26–30, 2018.

[31] Jeannette M. Wing. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A : Mathematical, Physical and Engineering Sciences*, 366(1881) :3717–3725, 2008.

[32] Jeannette M. Wing. Computational thinking. *Communications of the ACM*, 49(3) :33–35, 2006.

[33] Wallace Feurzeig, Seymour A. Papert, and Bob Lawler. Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments*, 19(5) :487–501, 2011.

[34] Wallace Feurzeig and George Lukas. LOGO—A programming language for teaching mathematics. *Educational Technology*, 12(3) :39–46, 1972.

[35] Wallace Feurzeig. Toward a culture of creativity : A personal perspective on Logo's early years and ongoing potential. *International Journal of Computers for Mathematical Learning*, 15(3) :257–265, 2010.

[36] Shuchi Grover and Roy Pea. Computational thinking in K–12 : A review of the state of the field. *Educational Researcher*, 42(1) :38–43, 2013.

[37] David H. Jonassen. Toward a design theory of problem solving. *Educational technology research and development*, 48(4) :63–85, 2000. Publisher : Springer.

[38] Jiří Dostál. Theory of Problem Solving. *Procedia - Social and Behavioral Sciences*, 174 :2798–2805, February 2015.

[39] Karl Duncker and Lynne S. Lees. On problem-solving. *Psychological monographs*, 58(5) :i, 1945. Publisher : American Psychological Association.

[40] Richard E. Mayer and Merlin C. Wittrock. Problem solving. *Handbook of educational psychology*, 2 :287–303, 2006.

[41] Richard E. Mayer and Merlin C. Wittrock. Problem-solving transfer. *Handbook of educational psychology*, pages 47–62, 1996.

[42] Richard E. Mayer. *Thinking, problem solving, cognition*. WH Freeman/Times Books/Henry Holt & Co, 1992.

[43] Ashok Kumar Veerasamy, Daryl D'Souza, Rolf Lindén, and Mikko-Jussi Laakso. Relationship between perceived problem-solving skills and

academic performance of novice learners in introductory programming courses. *Journal of Computer Assisted Learning*, 35(2) :246–255, 2019. Publisher : Wiley Online Library.

[44] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming : A review and discussion. *Computer science education*, 13(2) :137–172, 2003.

[45] Michael de Raadt. A review of Australasian investigations into problem solving and the novice programmer. *Computer Science Education*, 17(3) :201–213, 2007.

[46] Warnier Jean-Dominique. *Les procédures de traitement et leurs données*. Éditions d'Organisation, 1971.

[47] Robert A. Mallet. La méthode informatique. 1971. Publisher : Hermann.

[48] C. Pair. La construction des programmes. 1979.

[49] Claude Pair. L'apprentissage de la programmation. In *Colloque francophone sur la didactique de l'informatique*, pages 75–85. Association EPI, 1988.

[50] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM. event-place : Canterbury, UK.

[51] Raymond Lister, William Fone, Robert McCartney, Otto Seppälä, Elizabeth S Adams, John Hamer, Jan Erik Moström, Beth Simon, Sue Fitzgerald, Morten Lindholm, Kate Sanders, and Lynda Thomas. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. page 32, 2004.

[52] Michael de Raadt, Mark Toleman, and Richard Watson. Training strategic problem solvers. *ACM SIGCSE Bulletin*, 36(2) :48–51, 2004. Publisher : ACM New York, NY, USA.

[53] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Conferences in Research and Practice in Information Technology Series*, 2006.

[54] Michael J. Clancy and Marcia C. Linn. Patterns and pedagogy. *ACM SIGCSE Bulletin*, 31(1) :37–42, 1999. Publisher : ACM New York, NY, USA.

[55] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research*, pages 19–26, 2014.

[56] Orna Muller and Bruria Haberman. A Course Dedicated to Developing Algorithmic Problem-Solving Skills–Design and Experiment. 2009. Publisher : Citeseer.

[57] Michael De Raadt, Richard Watson, and Mark Toleman. Teaching and assessing programming strategies explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95*, pages 45–54. Australian Computer Society, Inc., 2009.

[58] Edward Sapir. Selected Writings in Language, Culture, and Personality, ed. *DG Mandelbaum*, 1949.

[59] Aman Yadav, Clif Kussmaul, Chris Mayfield, and Helen H. Hu. POGIL in computer science : faculty motivation and challenges. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 280–285, 2019.

[60] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. A multi-institutional study of peer instruction in introductory computing. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 358–363, 2016.

[61] Aman Yadav, Chris Mayfield, Sukanya Kannan Moudgalya, Clif Kussmaul, and Helen H. Hu. Collaborative Learning, Self-Efficacy, and Student Performance in CS1 POGIL. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 775–781, 2021.

[62] Catherine H. Crouch and Eric Mazur. Peer instruction : Ten years of experience and results. *American journal of physics*, 69(9) :970–977, 2001. Publisher : American Association of Physics Teachers.

[63] David M. Hanson. *Instructor's guide to process-oriented guided-inquiry learning*. Pacific Crest Lisle, IL, 2006.

[64] Richard Samuel Moog and James Nelson Spencer. *Process oriented guided inquiry learning (POGIL)*, volume 994. American Chemical Society Washington, DC, 2008.

[65] John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1) :81–112, 2007. Publisher : Sage Publications Sage CA : Thousand Oaks, CA.

[66] Christelle Bosc-Miné. Caractéristiques et fonctions des feed-back dans les apprentissages. *L'Année psychologique*, Vol. 114(2) :315–353, 2014. Publisher : NecPlus.

[67] Valerie J. Shute. Focus on formative feedback. *Review of educational research*, 78(1) :153–189, 2008. Publisher : Sage Publications.

[68] Birgit Harks, Katrin Rakoczy, John Hattie, Michael Besser, and Eckhard Klieme. The effects of feedback on achievement, interest and self-evaluation : the role of feedback's perceived usefulness. *Educational Psychology*, 34(3) :269–290, 2014. Publisher : Taylor & Francis.

[69] Benedikt Wisniewski, Klaus Zierer, and John Hattie. The power of feedback revisited : A meta-analysis of educational feedback research. *Frontiers in Psychology*, 10 :3087, 2020. Publisher : Frontiers.

[70] Julien Broisin and Clément Herouard. Soutien à l'apprentissage de la programmation : conception et évaluation d'un indicateur sémantique. In *9e Conference Environnements Informatiques pour l'Apprentissage Humain (EIAH 2019)*, pages 235–246, 2019.

[71] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 41–46, 2016.

[72] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1) :1–43, 2018. Publisher : ACM New York, NY, USA.

[73] Anis Bey, Patrick Jermann, and Pierre Dillenbourg. A Comparison between two automatic assessment approaches for programming : An empirical study on MOOCs. *Journal of Educational Technology & Society*, 21(2) :259–272, 2018.

[74] Anis Bey and Tahar Bensebaa. ALGO+, an assessment tool for algorithmic competencies. *2011 IEEE Global Engineering Education Conference (EDUCON)*, pages 941–946, 2011.

[75] Anis Bey and Tahar Bensebaa. Assessment makes perfect : improving student's algorithmic problem solving skills using plan-based programme understanding approach. *International Journal of Innovation and Learning*, 14(2) :162–176, 2013. Publisher : Inderscience Publishers Ltd.

[76] Taki Belhaoues, Tahar Bensebaa, Meriem Abdessemed, and Anis Bey. AlgoSkills : an ontology of Algorithmic Skills for exercises description and organization. *Journal of e-Learning and Knowledge Society*, 12(1), 2016.

[77] Julien Broisin and Clément Hérouard. Design and evaluation of a semantic indicator for automatically supporting programming learning. In *Proceedings of The 12th International Conference on Educational Data Mining (EDM 2019)*, pages 270–275. ERIC, 2019.

[78] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966. Issue : 8.

[79] Gorgoumack Sambe and Adrien Basse. Ontology Based Framework For Learning Algorithm. *International Journal of Scientific and Technology research*, 9(01) :5, 2020.

[80] Gorgoumack Sambe, Khadim Drame, and Adrien Basse. Towards a Framework to Scaffold Problem-solving Skills in Learning Computer Programming. In *Proceedings of the 13th International Conference on Computer Supported Education*, pages 323–330, May 2021.

[81] Gorgoumack Sambe, Adrien BASSE, and Khadim Drame. Towards a semantic comparison system of source code for support in learning programming. The 3rd International Conference on Modern Educational Technology (ICMET 2021), 2021.

Annexe A

Base de données des problèmes

A.1 Problèmes à solution séquentielle

TABLE A.1: Problèmes à solution séquentielle

numéro	titre	énoncé
1	double entier	Écrire un programme qui demande un nombre entier à l'utilisateur, puis calcule et affiche son double
2	carré entier	Écrire un programme qui demande un nombre entier à l'utilisateur, puis calcule et affiche son carré
3	périmètre rectangle	Écrire un programme qui calcule et affiche le périmètre d'un rectangle. La longueur et la largeur du rectangle sont demandées à l'utilisateur.
4	périmètre cercle	Écrire un programme qui calcule et affiche le périmètre d'un cercle. Le rayon est demandé à l'utilisateur.
5	somme de deux entiers	Écrire un programme qui demande à l'utilisateur de saisir deux nombres entiers, calcule puis affiche la somme des deux nombres.
6	somme de deux réels	Écrire un programme qui demande à l'utilisateur de saisir deux nombres réels, calcule puis affiche la somme des deux nombres.
7	moyenne de deux entiers	Écrire un programme qui demande à l'utilisateur de saisir deux nombres entiers, calcule puis affiche la moyenne des deux nombres.
8	échange de deux variables	Écrire un programme qui demande à l'utilisateur de saisir deux nombres entiers, échange leurs valeurs puis les affiche.
9	moyenne de trois entiers	Écrire un programme qui calcule et affiche la moyenne arithmétique de trois variables entières dont les valeurs sont demandées à l'utilisateur.
10	somme et moyenne de deux entiers	Écrire un programme qui calcule et affiche la somme et la moyenne arithmétique de deux variables entières dont les valeurs sont demandées à l'utilisateur.

TABLE A.1: Problèmes à solution séquentielle

numéro	titre	énoncé
11	somme et moyenne de trois entiers	Écrire un programme qui calcule et affiche la somme et la moyenne arithmétique de trois variables entières dont les valeurs sont demandées à l'utilisateur.
12	conversion durée	Écrire un programme qui demande à l'utilisateur de saisir une durée donnée en secondes et convertit cette durée en heure, minute et seconde.

A.2 Problèmes à solution conditionnelle

TABLE A.2: Problèmes à solution conditionnelle

numéro	titre	énoncé
13	signe d'un nombre	Écrire un programme qui demande à l'utilisateur de saisir un nombre entier, puis affiche son signe : '+' si le nombre est positif ou nul et '-' si le nombre est négatif.
14	signe nombre v2	Écrire un programme qui demande à l'utilisateur de saisir un nombre entier, puis affiche '+' si le nombre est positif, '0' si le nombre est nul et '-' si le nombre est négatif.
15	maximum de deux nombres	Écrire un programme qui demande à l'utilisateur de saisir deux nombres réels, puis affiche le maximum entre les deux nombres dans une variable max.
16	maximum de deux nombres v2	Écrire un programme qui demande à l'utilisateur de saisir deux nombres réels a et b, puis affiche -1 si $a < b$ et 1 si $a \geq b$.
17	maximum de deux nombres v3	Écrire un programme qui demande à l'utilisateur de saisir deux nombres réels a et b, puis affiche -1 si $a < b$, 0 si $a = b$ et 1 si $a > b$.
18	maximum de trois nombres	Écrire un programme qui demande à l'utilisateur de saisir trois nombres réels x, y et z, puis calcule et affiche le maximum entre les trois nombres.
19	signe produit	Écrire un programme qui demande deux nombres réels à l'utilisateur et l'informe du signe du produit : '-' si le produit est négatif, '+' s'il est positif et '0' s'il est nul. NB : on ne doit pas calculer le produit.

TABLE A.2: Problèmes à solution conditionnelle

numéro	titre	énoncé
20	une minute après	Écrire un programme qui demande à l'utilisateur de saisir l'heure actuelle avec trois variables (représentant l'heure, les minutes et les secondes), puis calcule et affiche l'heure une minute après. Par exemple : 14h 32, une minute après, donne 14h 33 ; 14h 59, une minute après, donne 15h 0 ; (Ne pas oublier le cas de minuit)
21	calculette	Écrire un programme permettant de simuler le fonctionnement d'une calculatrice. Dans cet exercice, on demande à l'utilisateur de saisir deux nombres et un opérateur qui est un caractère parmi +, -, *, /, puis on calcule et affiche le résultat correspondant. NB : dans le cas d'une division, on vérifiera bien que le dénominateur est non nul.
22	triangle	Étant données trois longueurs réelles a, b et c qui désigne les longueurs des trois cotés d'un triangle. On suppose que c est le plus grand coté du triangle. Écrire un programme qui détermine : si a, b et c peuvent être les côtés d'un triangle (O/N). si oui (O), le triangle en question est-il un triangle rectangle ? le triangle en question est-il un triangle isocèle ? Indications : si $a+b \geq c$, les longueurs correspondent à un triangle. si $a^2 + b^2 = c^2$, le triangle est rectangle. si $a^2 + b^2 = c^2$ et $a = b$, le triangle est isocèle et rectangle.

TABLE A.2: Problèmes à solution conditionnelle

numéro	titre	énoncé
23	équation du second degré	Écrire un programme qui calcule puis affiche le nombre de solutions et les solutions d'une équation du second degré de la forme $ax^2 + bx + c$. Utiliser pour cela le déterminant Δ. Pour rappel, si Δ est négatif, il n'existe pas de solution, si Δ est nul il existe une unique solution qui est $\frac{-b}{2a}$ et si Δ est positif, il y'a deux solutions $\frac{-b-\sqrt{\Delta}}{2a}$ et $\frac{-b+\sqrt{\Delta}}{2a}$.

A.3 Problèmes à solution itérative

TABLE A.3: Problèmes à solution itérative

numéro	titre	énoncé
24	somme des nombres à 10	Écrire un programme qui calcule puis affiche la somme des nombres de 1 à 10
25	somme des nombres à n	Écrire un programme qui calcule puis affiche la somme des nombres de 1 à n, n est demandé à l'utilisateur
26	factoriel 10	Écrire un programme qui calcule puis affiche le factoriel de 10
27	factoriel n	Écrire un programme qui calcule puis affiche le factoriel de n, n est demandé à l'utilisateur
28	puissance n-ième d'un nombre	Écrire un programme qui calcule puis affiche la puissance n-ième d'un entier x, x et n sont donnés par l'utilisateur.
29	suite numérique	Écrire un programme qui calcule puis affiche le n-ième terme de la suite suivante, n est demandé à l'utilisateur. $U_0 = 2$ $U_1 = 3$ $U_n = U_{n-1} + 2 * U_{n-2}$ pour tout $n \geq 2$
30	Calcul de pi (π)	Écrire un programme qui calcule puis affiche la valeur approchée de π. Le programme affiche le résultat trouvé lorsque le terme $\frac{1}{x}$ est plus petit que ϵ. ϵ est demandé à l'utilisateur Approximation de pi : $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \ldots$