Is it still worth the cost to teach compiling in 2020 ? A pedagogical experience through Hortensias compiler and virtual machine

Karim Baïna* and Boualem Benatallah†
*Alqualsadi research team (Innovation on Digital and Enterprise Architectures)
ADMIR Laboratory, Rabat IT Center,
ENSIAS, University Mohammed V in Rabat, Morocco
karim.baina@um5.ac.ma
†University of New South Wales, Australia
boualem@cse.unsw.edu.au

***Abstract*—With the disruption produced by extensive automation of automation due to advanced research in machine learning, and auto machine learning, even in programming language translation [Lachaux et al., 2020], the main goal of this paper is to discuss the following question *"Is it still worth the cost to teach compiling in 2020 ?"*. Our paper defends the *"Yes answer"* within software engineering majors. The paper also shares the experience of teaching compiling techniques course best practices since more than 15 years, presents and evaluates this experience through Hortensias, a pedagogical compiling laboratory platform providing a language compiler and a virtual machine. Hortensias is a multilingual pedagogical platform for learning end teaching how to build compilers front and back-end. Hortensias language offers the possibility to the programmer to customise the compiler associativity management, visualise the intermediary representations of compiled code, or customise the optimisation management, and the error management language for international students communities. Hortensias offers the possibility to the beginner programmer to use a graphical user interface to program by clicking. Hortensias compiling pedagogy evaluation has been conducted through two surveys involving in a voluntarily basis engineering students and alumni during one week. It targeted two null hypotheses : the first null hypothesis supposes that compiling teaching is becoming outdated with regards to current curricula evolution, and the second null hypothesis supposes Hortensias compiling based pedagogy has no impact neither on understanding nor on implementing compilers and interpreters. During fifteen years of teaching compiler engineering, Hortensias was a wonderful pedagogic experiment either for teaching and for learning, since vulgarising abstract concepts becomes very easier for teachers, lectures follow a gamification-like approach, and students become efficient in delivering versions of their compiler software product in a fast pace.**

***Index Terms*—Compiling, Compiler techniques, LL(1), One-address code pseudocode, Interpreter, Teaching compiling pedagogy.**

I. Introduction

According to [Milne and McAdam, 2011], over 12 Scottish universities offering applied computing or software engineering degrees, only 3 offer modules which study compiler design and implementation. This omission is often justified from a mistaken perception that the study of compilers is now irrelevant to modern software engineering practice. Since computing paradigms, and hardware facilities are increasingly evolving, new languages are born every month to propose high level models and hide complexities of distributed data computing, hardware management, etc.

Teaching compiler engineering, or at least basics of compiling mechanisms, remains a pre-requisite to be able to use 100% of advanced languages syntactic, semantic, and pragmatic features and benefit from their compiled code runtime performances. Concepts of regular expression, automaton, grammar, abstract syntactic tree, directed acyclic graph, control flow graph, virtual machine, memories types, machine code, and all their algorithms, are more than necessary for developing software engineering skills, and compiler coding is a wonderful experience to practice those concepts in realistic industry-like assignments. Being able to read, understand and detect drawback of a language grammar is a minimal skill for every computer engineer. Beside technical aspects of compiling, understanding and differentiating vocabulary, structure, semantics and pragmatics levels of a model or a meta-model is the basis of each information system modeling.

The remainder of this paper is structured as follows : section II provides paper background, section III presents followed research method, section IV presents the paper design artifact : Hortensias compiler in detail from front-end parser, semantic analyser, pragmatic analyser, IR generator, to back-end pseudo-code generator, optimiser, and back-end virtual machine and interpreter, section V presents the evaluation of Hortensias design artifact and Hortensias based pedagogy experience through a satisfaction survey approach and a literature review approach, section VI concludes the paper and provides some discussions. Finally, appendixes VII and VIII provide respectively Hortensias language formal basis – LL(1) grammar and Hortensias pseudo-code language language formal basis – LL(1) grammar.

This paper is accomplished thanks to a full-year sabbatical leave funded by The Ministry of National Education, Higher Education, Staff Training, and Scientific Research, Morocco.

II. Background

A. Why teaching compiling in computer engineering majors may be still worthful in 2020 ? - Opportunities

In the digitalisation era and the fourth industrial revolution, we are observing many IT evolutions accompanying the new digital/smart governments, companies, organisations, cities, citizen requirements. Those IT evolutions concern, among others, new computing scale-in/scale-out capabilities and architectures, new programming execution environments paradigm shifts, and last but not least the effervescence of exciting and innovating frameworks and programming languages. All this create a healthy biodiversity IT environment. However, bridging the gap between layers evolving in different directions is a heavy task that should involve many research, development, and engineering efforts. Compiling teaching contribute to providing engineers with a **good abstraction capability and technology independence** within this continuously evolving context (languages, frameworks, execution environments, hardware virtualisations, architectures, etc.), and thus insures a good career evolution without getting stuck and being dependent to some specific language, framework, execution environment, hardware, or IT architecture.

1) **Hardware Change caused by computing scale-in and rapid evolution and obsolescence of hardware :** (i) *Multi-core processor Computing* creates new challenges where classical languages and compilers do not use all capabilities of the CPU, and the more CPUs are becoming complex, the more compiling innovations need to follow their complexity to optimise the CPU usage. For instance, [Doerfert et al., 2019] proposes multi-core code compiling optimisations in OpenMP context. (ii) *Graphical processor unit -GPU- Computing* Compilers should discover, and expose sufficient instruction-level parallelism, find loop-style parallelism for vector/pipeline units and larger granularity parallelism for multi-GPU situations. For instance in the context of computing consuming deep learning models, there is a real need for compilers to efficiently produce GPU code for Deep Neural Networks (e.g. Triton language and compiler [Tillet et al., 2019]). In the same context, new profiles were born: "Deep Learning Compiler Software Engineer". (iii) *Quantum Computing* requires compilers that should optimise applications (toolflows) and abstraction layers and bridge the gap between the hardware size and reliability requirements of quantum computing algorithms and the physical machines. [Chong et al., 2017] discusses potential programming languages and compiler design for quantum hardware.
2) **Computing Architectures Change caused by computing scale-out (or Cloud to Edge computing transition) :** (i) *Distributed Databases (relational or NoSQL or Blockchain based), Big Data Computing* involve clusters of machine hosting distributed data blocks and parallel batch processing elements on those data blocks need continuously evolving compilers in the same evolution pace of both Big Data analytics languages and computing infrastructures (Hadoop, λ-Architecture, Kappa-Architecture, Smack-Architecture, IoDA (IoT-Big Data end-to-end Integrated Architecture) [Hibti et al., 2019], etc.). [Burdick et al., 2013] illustrates compiling and optimising execution plan systems and methods of Machine Learning algorithms high level operations towards a MapReduce environment low level operations. (ii) *High Performance Computing and Data in motion real time analysis processing* architectures needs optimisations of every data movement or in memory computing, and compilers should be aware of the complexity of those parallel and distributed architectures and APIs (e.g. MPI, Flink, Akka, Ignite, Storm, Spark Streaming, etc.). (iii) *Cloud Computing, Local Cloud Computing, and Mobile Computing* with rich support of Fog computing (local digital video processors, digital twins, haze cascading pattern [Hibti et al., 2019], etc.), Edge and Mobile computing (mobiles, IoT, smart sensors/actuators, robots/drones, etc.).
3) **Baby boom of Programming environments, Containerisation and Virtualisation diversity :** (i) *Diversity of new programming frameworks supporting digitalisation era* for IA (e.g. Tensorflow, Keras, Caffe, Torch, etc.-, for Big Data (Hadoop, Yarn, Akka, Samza, Kudu, etc.), for IoT (Zetta, DeviceHive, ThingSpeak, Mainflux, Thinger.io, etc.) requires framework awareness of compilers and optimisers without being framework specific. (ii) *Diversity of compilers target virtual environments* varying from virtual machines (e.g. JVM, CLI), source- and target-independent optimiser, and universal code generator (e.g. LLVM), universal/polyglot virtual machines (e.g. GraalVM, Truffle), Microservices architecture and Devops virtual machines and containers (e.g. Docker, Kubernetes, and Mesos, Swarm), etc. leverage compilers to suit softwarisation, reusability and agility needs of containerisation and virtualisation industry.
4) **Cacophony caused by Programming Polyglossia, and Multilinguism :** (i) *Multitude of programming languages and new paradigms* C/C++, Clang, Java, Python, Ruby, Erlang, Scala, Perl, JavaScript, AQL, R, Octave, etc. supporting new paradigms (object, functional, logic, actor, aspect, agent, artificial intelligence & machine learning, neural networks & deep learning, relational, nosql, etc.) requires machine language compiling and optimising , inter-language compiling and co-habitation frames and virtual machines (e.g. LLVM, GraalVM). (ii) *Multitude of database querying languages, data stores, and data flow languages* (e.g. SQL, Neo4J cypher, HBase, HiveQL, Pig Latin, Jaql, etc.) brings new constraints to data flow languages and querying compilers and interpreter in this context of polyglot data store querying and persistence [Sellami et al., 2014].

A software engineer who resolved a compiler building problematics, when confronted to multi-platform complex architecture, is able to (i) **build structured mind representation meta-models of every as-is and to-be architecture** components, data formats, and languages, (ii) to make better classifications of evolving tools, and (iii) to learn rapidly a new programming environment since he/she will have a heavy background in terms of language dimensions and layers, and **deeper understanding of what is being the black box**. An software engineer without compiling is a simply a technology user who will be limited in his/her capabilities of software development or integration problem diagnosis, and solving. *Compiling is a considerable pre-requisite for Model Driven Engineering discipline*. Moreover, during his/her career the software engineer will manipulate languages for data storage, data visualisation, languages in programming, and even more natural languages (non/semi structured databases, text analytics), etc. In a word, *compiling teaching is a perfect introduction to the Science of Text Algorithms.*

B. Real world education experience

First author experience in teaching compiler engineering in ENSIAS engineering school evolves since September 2004. Many experiments were tested before using Hortensias [Baïna, 2020] in its first integrated version in December 2010. This paper aims to present and evaluate Hortensias compiling based pedagogy, a pedagogical language coming with two components : hensiasc a pedagogical compiler of Hortensias language towards a simple one-address code, and and hensiasi a pedagogical interpreter of the generated one-address code. hensiasc and hensiasi are all written in C language. Pedagogical mean three things : Hortensias is (i) a simple imperative, not case sensitive, language for beginner programmers inspired from C, Ada, and other languages, (ii) Hortensias provides multilingual (for the moment : English, Spanish, French, and German) error messages for international students, and (iii) Hortensias code is a complete pedagogical platform for teaching and learning concepts and techniques for designing and building a LL(1)[1] top down compiler front-end (scaner, parser, pragmatic analyser, and intermediary representation -IR- generator), IR, and back-end (pseudo-code generator, optimiser and pseudo-code interpreter). Hortensias name is inspired from flowers name, but also contains ENSIAS suffix which stands for Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes, Mohammed V University in Rabat, Morocco where Hortensias was built. Hortensias framework is composed of many software components : a graphical user interface for young programmers, a compiler, and an interpreter as shown in figure 2. In a Java-like style, Hortensias compiler (hensiasc) takes an Hortensias program, compiles it to an abstract pseudo-code (one address bytecode), then Hortensias interpreter (hensiasi) reads this generated pseudo-code to interpret it in a virtual/abstract Hortensias machine. Hortensias is written in C language based on flex scaner generator. A Hortensias user may be (i) a *beginner programmer* aiming to learn programming without the barrier of syntactical instructions coding, (ii) a *programmer* aiming to discover programming through a simple imperative language, and a virtual machine interpreter through its simple pseudo-code (bytecode) artifact, (iii) a *compiling course student* aiming to learn compiling techniques, their concepts, and implementations, understand exercises, resolve labworks, and prepare exams, or (iv) a *compiling course lecturer* aiming to teach compiling techniques, design exercises, labworks, exams, and correct exams. Figure 1 shows Hortensias front-end, IR generation API, and back-end API Management.

III. Research Method : Design Science Methodology

This paper study is based on real world education experience. To capitalise this education experience, this paper research methodology follows Design Science Research Approach consisting of a rigorous end-to-end scientific process aiming constructing a Design Artifact (section IV details the construction of a pedagogical compiler & interpreter Hortensias as a Design Artifact in our case), and evaluating this design artifact (section V presents the evaluation of Hortensias and Hortensias based pedagogy experience through a satisfaction survey approach and a literature review approach in our case). The design science research approach main purpose, according to Hevner [Hevner et al., 2008], is to achieve knowledge understanding of a problem domain by building and applying a designed artifact following seven guidelines. **Guideline 1:** *Design as an Artifact* – producing a viable artifact in the form of a construct, a model, a method, or an instanciation. **Guideline 2:** *Problem Relevance* – developing technology-based solutions to important and relevant business problems. **Guideline 3:** *Design Evaluation* – demonstrating the utility, quality, and efficacy of a design artifact via well-executed evaluation methods. **Guideline 4:** *Research Contributions* – providing clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. **Guideline 5:** *Research Rigor* – applying rigorous methods in both the construction and evaluation of the design artifact. **Guideline 6:** *Design as a Search Process* – utilizing available means to reach desired ends while satisfying laws in the problem environment. **Guideline 7:** *Communication of Research* – presenting to both technology-oriented as well as management-oriented audiences.

Hortensias compiling pedagogy evaluation has been conducted through two surveys involving in a voluntarily basis more than sixty engineering students and alumni during one week. Through 15 questions of 5 different evaluation levels, these surveys targeted two main null hypotheses : whether compiling teaching is becoming outdated with regards to current curricula evolution ? and whether Hortensias compiling based pedagogy has no impact neither on understanding nor on implementing compilers and interpreters ?

IV. Designed Artifact Construction – Hortensias Language Compiler, and Pseudo-code interpretation Virtual Machine

We implement the design artifact construction main step of Design Science Approach through modeling, and applying Hortensias pedagogical platform front-end, middle-end, and back-end artifact components.

[1] Note that Hortensias analyser has a LALR formalisation and implementation version too that is not detailed in this paper. This version uses the same Hortensias APIs.

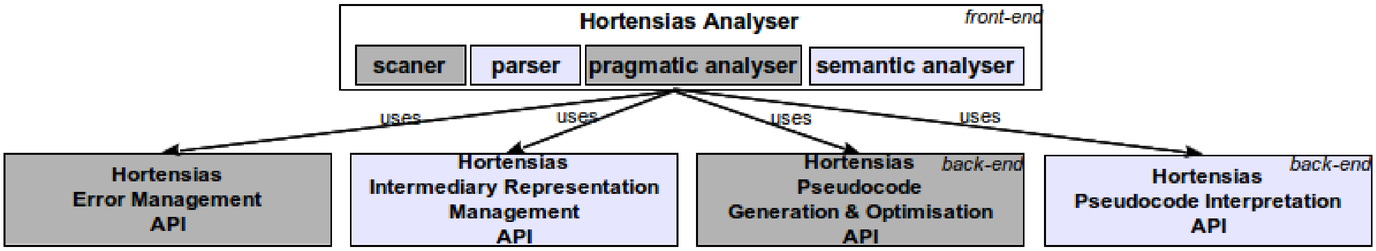

Hortensias Analyser
front-end
scaner
parser
pragmatic analyser
semantic analyser
uses
uses
uses
uses
Hortensias Error Management API
Hortensias Intermediary Representation Management API
Hortensias Pseudocode Generation & Optimisation API
back-end
Hortensias Pseudocode Interpretation API
back-end

Figure 1: Hortensias API Management

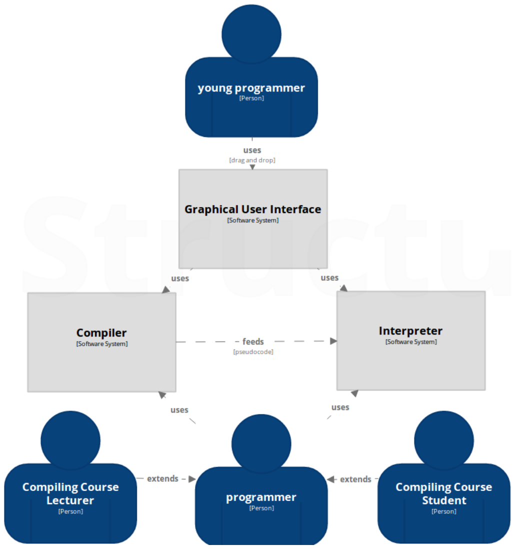


Figure 2: Hortensias Framework Architecture

A. *Deriving Design Artifact Requirements*

Requirements will be formalised as user stories focused on the main Hortensias platform user : "Compiling Course Student" (illustrated in figure 2).

User Story 1 : The user must be able to customize Hortensias code to program and test his/her own compiler front-end :

1) scaner.
2) LL(1) parser
3) semantic analyser
4) pragmatic analyser

User Story 2 : The user must be able to customize Hortensias code to program and test his/her own compiler middle-end :

1) intermediary representation generator
2) intermediary representation optimiser

User Story 3 : The user must be able to customize Hortensias code to program and test his/her own compiler back-end :

1) pseudo-code generator
2) pseudo-code interpreter

B. *Hortensias Syntax Analyser (Parser)*

Hortensias language parser is a top-down LL(1) analyser based on Hortensias syntax partially represented graphically[2] by the rail road diagrams of table I. The complete formalised LL(1) grammar is detailed in appendix VII.

Here are two Hortensias language programs examples : a for and a switch-case examples.

Listing 1: Hortensias for example

```
#spanish
#leftassoc
#staticoptimiser

REM this program computes 120!
n int 10;
facto int 1;
i int;

begin

  for i = 1 to 120 do
    facto = facto * i;
  endfor

print facto;

end
```

C. *Hortensias Semantic Analyser*

1) Semantic Error management: Hortensias handles eleven semantic errors as follows:

1) Not Declared Variable : Each variable should not be used in instruction body without pre-declaration in declaration part.
2) Already Declared Variable : Each variable should be declared once.
3) Not Initialised Variable : Each variable should be initialised.
4) Badly Initialised Variable : Each typed variable should be initialised with a value of a type compatible with declaration type (implicit casting may be possible).
5) Incompatible Assign Type : Each left and right Assignment expressions should be of the compatible type (implicit casting may be possible).
6) Incompatible Comparison Type : Each Left and Right comparison expressions should be of compatible types (implicit casting may be possible).
7) Incompatible Operation Type : Each Left and Right operation expressions should be of compatible types (implicit casting may be possible).

[2] The syntax visualisation has been done thanks to Railroad Diagram generator [Rademacher, 2019] clever tool.

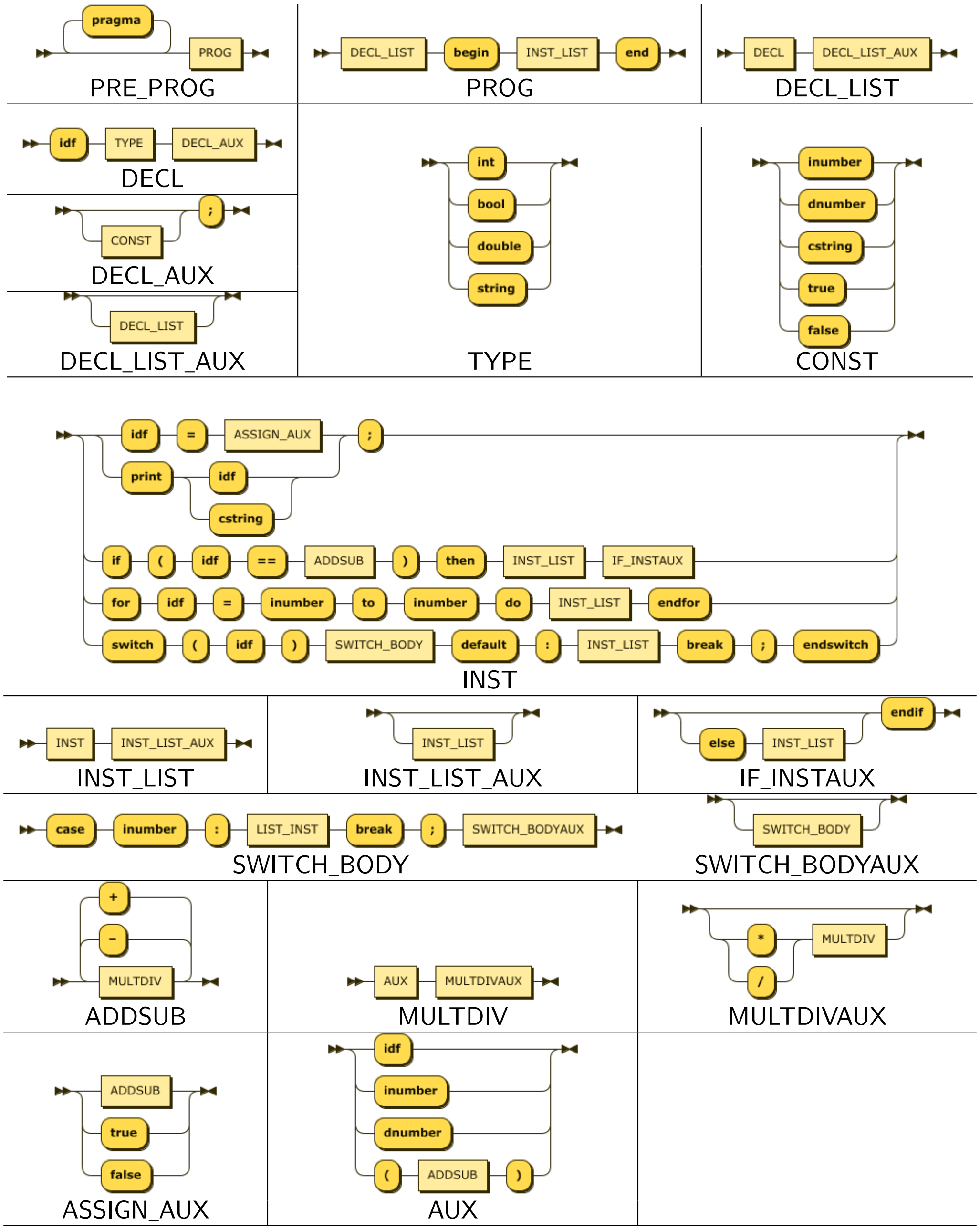


Table I: Hortensias visual syntax

8) Incompatible For Index Type : Each for loop index should be integer.
9) Incompatible Switch Index Type : Each switch case index should be integer.
10) Switch Multiple Values : Each switch case value should appear one time at most.
11) Division by Zero[3] : Each denominator arithmetic expression evaluation cannot produce a zero value.

2) Type implicit casting: Implicit casting may be possible in three cases : (1) a typed variable is initialised with a value of a type different but compatible with declaration type, (2) a left and a right assignment expressions are not of the same type but belong to compatible types, and (3) a left and a right comparison expressions are not of the same type but belong to compatible types.

1) declaration case : Let $[x \ \ double \ \ v;]$ a declaration, with x a variable, and v a numeric value, $\frac{[x\ double\ v;] \wedge type(v)=int}{[x\ double\ (double)v;]}$
   All other different types for x and v will produce Badly Initialised Variable error for declaration case.
2) assignment case : Let $[x \ \leftarrow \ e;]$ an assignment, with x a variable, and e an arithmetic expression, $\frac{[x \leftarrow e;] \wedge type(x)=double \wedge type(e)=int}{[x \leftarrow (double)e;]}$
   All other different types for x and v will produce

[3] To detect division by zero, Hortensias pseudo-code generator pre-generates pseudo-code and simulates an execution across the produced pseudo-code in order to verify safety, and discover null denominators in compile-time.

Incompatible Assign Type error for assignment case.

3) comparison cases :
Let $[if\ (x\ ==\ e)\ then\ inst\_list\ endif]$ a conditional if instruction,

   a) with x a variable, and e an arithmetic expression, $\frac{[if\ (x\ ==\ e)\ then\ inst\_list\ endif] \wedge type(x)=double \wedge type(e)=int}{[if\ (x\ ==\ (int)e)\ then\ inst\_list\ endif]}$
   b) with x a variable, and e an arithmetic expression, $\frac{[if\ (x\ ==\ e)\ then\ inst\_list\ endif] \wedge type(x)=int \wedge type(e)=double}{[if\ ((double)x\ ==\ e)\ then\ inst\_list\ endif]}$

All other different types for x and e will produce Incompatible Comparison Type error for comparison case.

3) Type inference: Let e_1, and e_2 be two arithmetic expressions, and let $e_1\ op\ e_2$ be an arithmetic abstract syntactic tree (ast), if e_1 and e_2 are of two different numeric types, the type of every operations gathering e_1, and e_2 will be inferred as the most general type by the semantic analyser. Resulting AST types are induced by the following inference rules :

- $\frac{type(e_1)=int \wedge type(e_2)=double}{type(e_1\ op\ e_2) \leftarrow double}$ where $op \in \{+, -, *, /\}$
- $\frac{type(e_1)=double \wedge type(e_2)=int}{type(e_1\ op\ e_2) \leftarrow double}$ where $op \in \{+, -, *, /\}$

All other different types for e_1 and e_2 will produce Incompatible Operation Type error.

D. Hortensias Pragmatic Analyser

Hortensias pragmatic analyser enables the programmer to customise at compile time four types of pragmatic aspects : (i) associativity orientation customisation : either left (default) or right for all arithmetic operations, (ii) compiler optimisations : either static, dynamic or default (generate code with no optimisation), (iii) intermediary representations visualisation (control flow graph - CFG, and embedded abstract syntactic trees - ASTs) and (iv) finally error management language : either Spanish, French, English or German. Figure 3 shows Hortensias pragmatic parser fundamental decisions and interactions with intermediary representation generator, and pseudo-code optimiser[4]. Note that for simple pedagogy, the figure highlights a fixed order of detecting pragma directives (optimisation then associativity). In reality, pragma primitives are parsed by syntactic analyser in a free order (see PRAGMA_LIST syntactic rules), even each specific pragma detection details are let to lexical scaner responsibility (see pragma terminal). Also, a pragma may cancel the effect a previous pragma. That means, that if many pragma from the same type appear in the code, the latest mentioned in the code is the one that the compiler will choose. A pragma may also occur multiple times without verification. Moreover, neither intermediary representation visualisation, nor error management pragmatic customisation pragmas have been mentioned in this figure without loosing in generality.

Listing 2: Hortensias switch-case example

```
#controlflowgraph
#french
#dynamicoptimiser

Saison int ;
Automne int 1; Hiver int 2; Printemps int 3;
    Ete int 4;
Pays int ; Maroc int 1; France int 2;
Temperature double;

begin

Saison = Hiver; Pays = France;

switch ( Saison )
case 1 : switch ( Pays )
  case 1 : Temperature = 10.0 ;   break;
  default : Temperature = 3.0 ;   break;
  endswitch break;
case 2 : switch ( Pays )
  case 1 : Temperature = 4 ;      break;
  default : Temperature = 0 ;     break;
  endswitch break;
case 3 : switch ( Pays )
  case 1 : Temperature = 35 ;     break;
  default : Temperature = 28 ;    break;
  endswitch break;
default : switch ( Pays )
  case 1 : Temperature = 35 ;     break;
  default : Temperature = 28 ;    break;
  endswitch break;
endswitch
Print Temperature;
end
```

1) Pragmas for associativity : Tuning associativity at compile time: Hortensias offers the possibility to the beginner programmer to customise the compiler associativity management as follows :

- #rightassoc : selects a semantic analyser with right AST. This mode assigns a right associativity to all binary operators : +, -, /, * and enables generation of AST, and pseudo code related to it.
- #lefttassoc : selects a semantic analyser with left AST. This mode assigns a left associativity to all binary operators : +, -, /, * and enables generation of AST, and pseudo code related to it. This associativity mode is by default.

2) Pragmas for optimisation management : Tuning compiler language at compile time: Hortensias offers the possibility to the programmer to customise the compiler optimisation management as follows :

- #staticoptimiser : activates the static optimiser (by default no optimiser is active). The static optimisation targets the non generation of unused variables, the transformation of non modified variables to constants, the computing in the code, the non generation of dead path in pseudo code, the static optimisation of stack use (left-right-root (post-order) generation for AST when left associativity, and right-left-root (reverse post-order) generation for AST when right associativity), and the non generation of obvious operations.
- #dynamicoptimiser : insures all #staticoptimiser optimisations and enhances them with generating the deeper AST

[4] The visualisation of pragma directives consequences on the whole compiling workflow has been done thanks to Bonita tool [Baïna and Baïna, 2013].

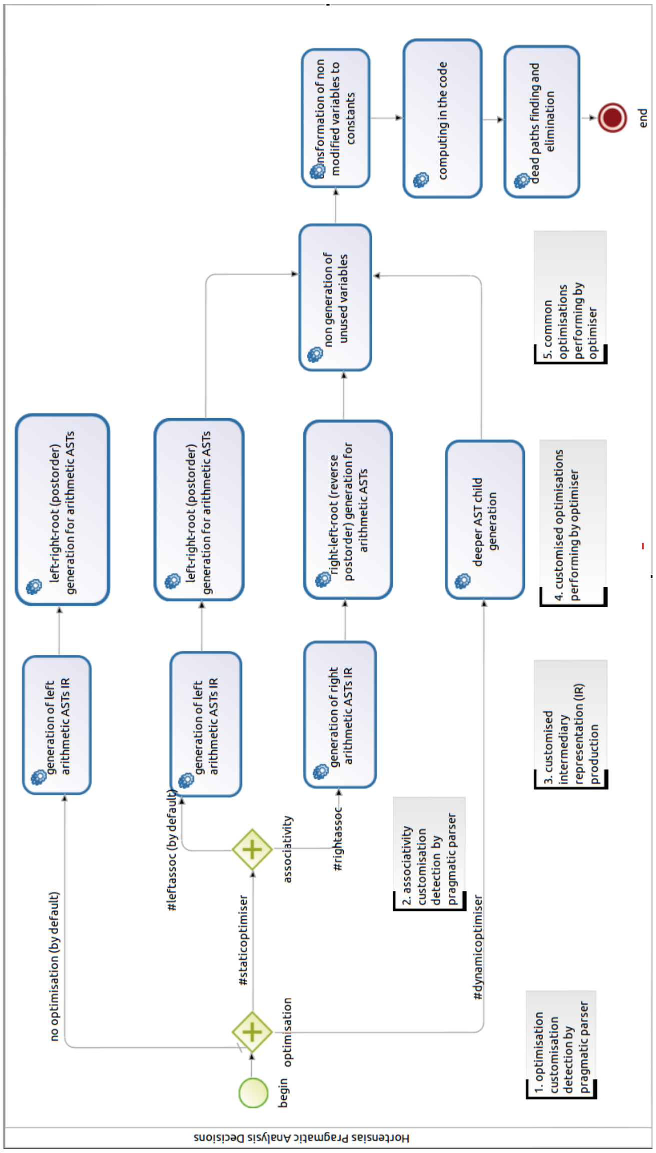

Hortensias Pragmatic Analysis Decisions
begin
optimisation
no optimisation (by default)
#staticoptimiser
#dynamicoptimiser
associativity
#leftassoc (by default)
#rightassoc
generation of left arithmetic ASTs IR
generation of left arithmetic ASTs IR
generation of right arithmetic ASTs IR
left-right-root (postorder) generation for arithmetic ASTs
left-right-root (postorder) generation for arithmetic ASTs
right-left-root (reverse postorder) generation for arithmetic ASTs
deeper AST child generation
non generation of unused variables
nsformation of non modified variables to constants
computing in the code
dead paths finding and elimination
end
1. optimisation customisation detection by pragmatic parser
2. associativity customisation detection by pragmatic parser
3. customised intermediary representation (IR) production
4. customised optimisations performing by optimiser
5. common optimisations performing by optimiser

Figure 3: Hortensias pragmatic parser decisions and interactions

first in arithmetic expressions in all kind of associativity.

By default, no optimisation is activated.

3) Pragmas for multilingual error management : Tuning compiler language at compile time: Error management is among functionalities that enhance the learning curve of a beginner programmer especially when he/she has to decrypt both programming language, and meta-language used by the compiler error manager. Hortensias offers the possibility to the beginner programmer to customise the compiler error management language as follows :

- #English : switch to English error handler (default one)
- #French : switch to French error handler
- #Spanish : switch to Spanish error handler
- #German : switch to German error handler

The latest pragma mentioned in the code is the one the compiler will choose for error management. The following example illustrates error management customisation.

Listing 3: Hortensias error management internationalisation

```
1 #spanish
2 x bool 0;
3 x int;
4 f bool;
5 t bool;
6 begin
7 print f;
8 if (x==f) then
9 if (y==7) then
10 x = 90;
11 else
12 t = 120 + f + t;
13 endif
14 endif
```

Here are errors found by the semantic analyser. The user has specified #spanish for error management messages, #english is by default if no language is specified.

```
Bienvenido al comilador del lingua Hortensias 2.2
l 2: error semantico: x variable mal iniciada
l 3: error semantico: x variable ya declarada
l 8: ..: f variable incompatible con la operacion
l 8: ..: x incompatible con el valor de comparacion
l 9: error semantico: y variable no declarada
l 10: ..: x incompatible con el valor de asignacion
l 12: ..: f variable incompatible con la operacion
l 12: ..: t variable incompatible con la operacion
l 12: ..: t incompatible con el valor de asignacion
l 15: error sintactico: instruccion esperada
```

4) Pragmas for intermediary representation visualisation: Hortensias offers the possibility to the programmer to visualise the produced intermediary representations of the code instead of generating pseudo-code by using #controlflowgraph pragma which bypasses both Hortensias optimiser and pseudo-code generator to print textually the whole structure of the compiler code abstraction built in memory during syntactical, pragmatic, and semantic analysis phases. The two examples of table II illustrate multilingual intermediary representation visualisation.

E. Hortensias One-address pseudo-code generator

Hortensias language is compiled to a portable one-address pseudo-code (bytecode) by a compiler (hensiasc). This pseudo-code contains 16 operation codes (opcode). Appendix VIII details Hortensias pseudo-code language grammar. This pseudo-code is interpreted by Hortensias interpreter (hensiasi) through a simple abstract machine. The pseudo-code is constituted of (i) a DATA section which presents compiled variables to the interpreter, as a set of key-value lines, then (ii) a LIST_INST body which lists compiled pseudo-code one-address[5] instructions as follows : (i) binary four arithmetic operators are handled by add (addition), mult (multiplication), idiv (integer division), ddiv (decimal division), sub (subtraction), (ii) four branching operators : jmp (unconditional jump), jne (jump if not equal), jg (jump if greater), jeq (jump if equal), (iii) variable operators : load (evaluates a variable by copying its value form the virtual machine static memory and pushes it

[5] One-address operators always suppose the stack containing implicit operands pushed in the corresponding commutative operation order.

Hortensias **sample code**	**CFG visualisation** #controlflowgraph	
	#english #rightassoc	#german #leftassoc

value double;
a double;
b double;
c double;
d double;
e double;
f double;
g double;
begin
a = 2;
b = 7;
c = 9;
d = 6;
e = 1;
f = 5;
g = 3;
value =
a * b - c + d
/ e + f * g ;
PRINT value;
end
	Visualisation of Control Flow
	Graph :
	AssignArith a = 2.000000;
	AssignArith b = 7.000000;
	AssignArith c = 9.000000;
	AssignArith d = 6.000000;
	AssignArith e = 1.000000;
	AssignArith f = 5.000000;
	AssignArith g = 3.000000;
	AssignArith value =
	(- (* a b)
	    (+ c (+ (/ d e) (* f g)))) ;
	PrintIdf value ;
		Visualisierung des
		Kontrollflussgraphen :
		AssignArith a = 2.000000;
		AssignArith b = 7.000000;
		AssignArith c = 9.000000;
		AssignArith d = 6.000000;
		AssignArith e = 1.000000;
		AssignArith f = 5.000000;
		AssignArith g = 3.000000;
		AssignArith value =
		(+
		 (+
		  (-
		    (* a b)
		    c
		  )
		  (/ d e)
		)
		 (* f g)
		) ;
		PrintIdf value ;

Table II: Multilingual intermediary representation visualisation

into the virtual machine stack), store (opposite of load operator : pops the top value of virtual machine stack and copy it as new value of a name variable in the the virtual machine static memory), (iv) stack operators : dupl (duplicate the top value of virtual machine stack), swap (interchange the two top values of virtual machine stack when the generator encounters a non commutative operation e.g. $/$ or $-$), (v) constant evaluation operator : push (evaluates a constant value by pushing it into the virtual machine stack), and (vi) printing operators : printi (print identifier), prints (prints a string constant).

Hortensias pseudo-code generation follows a classical recursive intermediary representation driven generation algorithm. This algorithm is not detailed in this paper without loosing in generality. However, its pedagogical particularity is that it operates a simulation phase to enhance a posteriori semantic analysis, and also it prepares code optimisation phase. In fact, the simulation phase consists in pre-generating pseudo-code and running an anticipated pseudo-code interpretation (invisible for the end programmer) in order to (1) detect Division by zero semantic error (by discovering zero value denominators), (2) mark live execution path to distinguish it from dead paths useful for the optimisation part.

The example of table III illustrates the pseudo-code resulting from compiling a relatively simple Hortensias code (a factorial code).

F. Hortensias *optimiser*

Hortensias optimiser operates optimisations with regards to many aspects : (1) Dead path elimination, (2) Ignoring not used variables, (3) Transforming non modified variables to constants, (4) Computing in the code, (5) Stack use optimisation, and (6) More basic optimisations : bypassing code generation for obvious operation. In the following sections every optimisation aspects will be detailed.

Hortensias **sample code**	**Pseudo-code**

#spanish
#leftassoc
 	
REM this program
REM computes 120!
n int 10;
facto int 1;
i int;
 	
begin
 	
for i = 1 to 120 do
facto = facto * i;
endfor
 	
print facto;
 	
end
	n 10.000000
	facto 1.000000
	i 0.000000
	begin:
	PUSH 1.000000
	STORE i
	for0:
	PUSH 120.000000
	LOAD i
	JG endfor0
	LOAD facto
	LOAD i
	MULT
	STORE facto
	PUSH 1.000000
	LOAD i
	ADD
	STORE i
	JMP for0
	endfor0:
	LOAD facto
	PRINTI
	end:

Table III: Factorial loop resulting pseudo-code

1) Dead path elimination: The control flow graph abstracted from the code during syntactic/pragmatic/semantic analysis pass contains the whole workflow execution paths that the program may take during runtime. Only one path among all those workflow paths is the executable path. To detect this executable path, and thus to eliminate all dead paths pseudo-code generation, Hortensias optimiser pre-generates pseudo-code and simulates an execution across the produced pseudo-code in order to mark/distinguish executable pseudo-code instruction from dead instructions. Hence, Hortensias optimiser can produce an optimised pseudo-code gathering

Table IV: Dead path elimination optimisation

Hortensias **sample code**	**Pseudo-code** with	
	#staticoptimiser	no optimisation

x int;
pi double;
begin
pi = 3.14;
print pi;
end
	pi 0.000000
	begin:
	PUSH 3.140000
	STORE pi
	LOAD pi
	PRINTI
	end:
		x 0.000000
		pi 0.000000
		begin:
		PUSH 3.140000
		STORE pi
		LOAD pi
		PRINTI
		end:

Table V: Ignoring a not used variable optimisation

only executables pseudo-code instructions.

The example of table IV illustrates the pseudo-code resulting from dead path elimination optimisation during code generation of a simple nested if-the-else Hortensias code.

Hortensias **sample code**	**Pseudo-code** with	
	no optimisation	#staticoptimiser

a int 1;
b int 2;
c int 3;
 	
begin
 	
if (a==a) then
if (a==b) then
if (a==c) then
print "case 1";
else
print "case 2";
endif
else
print "case 3";
endif
endif
 	
end
	a 1.000000
	b 2.000000
	c 3.000000
	begin:
	LOAD a
	LOAD a
	JNE endif0
	LOAD b
	LOAD a
	JNE else1
	LOAD c
	LOAD a
	JNE else2
	PRINTS "case 1"
	JMP endif2
	else2:
	PRINTS "case 2"
	endif2:
	JMP endif1
	else1:
	PRINTS "case 3"
	endif1:
	endif0:
	end:
		a 1.000000
		b 2.000000
		c 3.000000
		begin:
		PUSH 1.000000
		LOAD a
		JNE endif0
		PUSH 2.000000
		LOAD a
		JNE else1
		else1:
		PRINTS "case 3"
		endif1:
		endif0:
		end:

2) Ignoring not used variables: Many times, programmers declare variables than never use in their program body. Those variables cause space wasting both in static, and code memory. The pseudo-code optimiser detects variables that are declared and never referenced in any instruction in the code, and ignores those variables in pseudo-code generation pass.

The example of table V illustrates the pseudo-code resulting from ignoring a not used variable optimisation during code generation of a simple Hortensias code.

Hortensias **sample code**	**Pseudo-code** with	
	#staticoptimiser	no optimisation

pi double 3.14;
piprime double;
begin
piprime = pi;
print piprime;
end
	pi 3.140000
	piprime 0.0000
	begin:
	PUSH 3.140000
	STORE piprime
	LOAD piprime
	PRINTI
	end:
		pi 3.140000
		piprime 0.0000
		begin:
		LOAD pi
		STORE piprime
		LOAD piprime
		PRINTI
		end:

Table VI: Transforming a non modified variable to a constant optimisation

Hortensias **sample code**	**Pseudo-code** with #rightassoc & #dynamicoptimiser	**Pseudo-code** with #rightassoc with no optimisation

value double;
begin
value =
 (
  (
   (2 * 7)
   - 9)
 + 6)
/ (1 + 5) * 3;
PRINT value;
end
	value 0.000000
	begin:
	PUSH 0.611111
	STORE value
	LOAD value
	PRINTI
	end:
		value 0.000000
		begin:
		PUSH 2.000000
		PUSH 7.000000
		MULT
		PUSH 9.000000
		SWAP
		SUB
		PUSH 6.000000
		ADD
		PUSH 1.000000
		PUSH 5.000000
		ADD
		PUSH 3.000000
		MULT
		SWAP
		DDIV
		STORE value
		LOAD value
		PRINTI
		end:

Table VII: Computing in the code optimisation impact on pseudo-code compiling

3) Transforming non modified variables to constants: During the programming phase, we often declare several variables to store different data that will be needed during the program. However, it is possible in many cases that variables are only declared to store certain values without modification neither by assignment (as left expression), nor for loop (as loop index), nor switch statements (as variable). Thus those variables can be seen as simple constants in the program. So to overcome this problem, the static optimiser intervenes during the compilation phase to replace all the variables that are never modified by their real value in the pseudo-code. This allows us not only to get rid of the expensive loading of variables, but also to be able to lighten the stack that no longer needs to store variables unnecessarily.

The example of table VI illustrates the pseudo-code resulting from transforming a non modified variable to a constant optimisation during code generation of a simple Hortensias code.

4) Computing in the code: When Hortensias pseudo-code generator parses AST and encounters constants and operations between constants, instead of generating pseudo-code that achieve computing operations between those constants, the pseudo-code optimiser chooses to achieve the computing itself and to generate the resulting output constant.

The example of table VII illustrates the pseudo-code resulting from computing in the code optimisation during code generation of a constant computing Hortensias code.

5) Stack use optimisation: When activated, Hortensias optimiser adopts three tactics with regards to stack use optimi-

Hortensias **sample code**	**Pseudo-code** with **#rightassoc**	**Pseudo-code** with **#leftassoc**
	left-right-root (post-order) code generation (no stack optimisation)	
	`begin:`	`begin:`
	`..`	`..`
	`LOAD a`	`LOAD a`
	`LOAD b`	`LOAD b`
`..`	`MULT`	`MULT`
`begin`	`LOAD c`	`LOAD c`
`..`	`LOAD d`	`SWAP`
`value =`	`LOAD e`	`SUB`
`a`	`SWAP`	`LOAD d`
`* b`	`DDIV`	`LOAD e`
`− c`	`LOAD f`	`SWAP`
`+ d`	`LOAD g`	`DDIV`
`/ e`	`MULT`	`ADD`
`+ f`	`ADD`	`LOAD f`
`* g ;`	`ADD`	`LOAD g`
`print value;`	`SWAP`	`MULT`
`end`	`SUB`	`ADD`
	`STORE value`	`STORE value`
	`LOAD value`	`LOAD value`
	`PRINTI`	`PRINTI`
	`end:`	`end:`

Table VIII: Arithmetic association impact on compiling pseudo-code of a simple Hortensias program with no stack-optimisation

sation (1) *static optimisation for left associativity* : left-right-root (post-order) code generation for left degenerated ASTs since they are the deeper trees by construction, (2) *static optimisation for right associativity* : right-left-root (reverse post-order) code generation for right degenerated ASTs since they are the deeper trees by construction, and finally finally (3) *dynamic optimisation for all associativities* : generate always deeper AST child first.

Examples of table VIII illustrates the pseudo-code resulting from of a simple Hortensias code compiling with no stack optimisation (left-right-root (post-order) generation) with regards to with association orientation.

Examples of table IX illustrate the comparison between different pseudo-codes resulting from of a simple Hortensias code compiling with static and dynamic stack optimisations.

6) More basic optimisations : bypassing code generation for obvious operation: Hortensias offers since its static optimiser mode many basic optimisations like short-cutting code generation for obvious operations : addition of zero, subtraction of zero, variable subtraction of itself, multiplication by zero, multiplication by one, division by one, variable division by itself, , etc.

G. Hortensias Pseudo-code interpretation Virtual Machine

Hortensias pseudo-code interpretation is supported by a Hortensias Virtual Machine composed of three types of memories (i) a *code memory* to store the linear pseudo-code intermediary representation produced by the pseudo-code generator, (ii) a *static memory* to store the global variables, and their values extracted for DATA section of the pseudo-code , and (iii) a *stack memory* (VM_STACK) to support the runtime semantic during the one-address pseudo-code interpretation. Note that no heap memory is supported by Hortensias virtual machine due to the fact that memory management is basically static in the current version of Hortensias (no pointer, nor dynamic allocation, nor tables are proposed, but the community may extend Hortensias for this). Hortensias pseudo-code interpreter is independent of previous Hortensias compiler components, since a programmer (respectively a lecturer), for a pedagogical purpose of learning (respectively teaching) machine language programming, may produce manually a pseudo-code program and use the interpreter to validate and interpret his/her code.

The following algorithm presents the way Hortensias virtual machine uses its *static* and *stack memory* to interpret each pseudo-code instructions stored linearly in its *code memory*.

Algorithm IV.1: INTERPRET(pseudo-code_inst)

```
codop ← pseudo-code_inst.codop
switch codop
  case DATA: create new variable data by name
             and value in VM static memory
  case $OP \in \{$ADD, SUB, MULT, IDIV, DDIV$\}$ :
             op1 ← pop(VM_STACK);
             op2 ← pop(VM_STACK);
             push(VM_STACK,
                eval(operator(OP),
                   eval(op1), eval(op2));
  case LOAD:
             data ← read variable value from
             static memory by name
             push(VM_STACK, data);
  case STORE:
             op ← pop(VM_STACK);
             update VM static memory with
             popped data to specified variable
             name location
  case DUPL:
             op ← pop(VM_STACK);
             push(VM_STACK, eval(op1));
             push(VM_STACK, eval(op1));
  case SWAP:
              op1 ← pop(VM_STACK);
              op2 ← pop(VM_STACK);
             push(VM_STACK, eval(op1));
             push(VM_STACK, eval(op2));
  case PUSH: data ← read constant data value
             push(VM_STACK, data);
  case $JMPOP \in \{$JNE, JG, JEQ$\}$ :
             op1 ← pop(VM_STACK);
             op2 ← pop(VM_STACK);
             if $eval(operator(JMPOP,$
             $eval(op1), eval(op2)) ==$ true
   then {       branching to specified label
  case JMP :
             branching to specified label
  case $PRNT \in \{PRINTS, PRINTI\}$ :
             display specified data
  else pass. ex. LABEL, etc.
```

The example of table X illustrates the pseudo-code resulting from of a simple Hortensias code compiling and its final interpretation output.

V. Designed Artifact Evaluation – Survey & Literature review Approach

We implement the evaluation main step of Design Science Approach through evaluating Hortensias and Hortensias based pedagogy experience towards a satisfaction survey approach enriched with and a literature review approach.

A. *Student satisfaction surveys*

This student satisfaction surveys aim to assess the satisfaction of students and alumni about (i) compiling lecture in general (survey 1), and (ii) Hortensias platform in particular (survey 2), regarding promoting desired pedagogical outcomes.

1) Surveys' Approach: The student satisfaction surveys were based on a selection of 15 questions divided into two separate surveys one general survey about compiling lecture (survey 1 - 5 questions), and one specific survey about Hortensias pedagogy supporting platform (survey 2 - 10 questions). Some questions have been inspired directly (question Q1.1) or adapted to Hortensias from LISA nine questions context [Mernik and Zumer, 2003] as a compiling educational survey reference even if in this latter the survey participants segments and distribution are unknown. In our surveys, each question targets one or more Kirkpatrick training outcome 5 levels [Kirkpatrick, 1975] : **level 1 - reactions** assessing hot reactions of trainees to the training curriculum and training process, their engagement, and contribution, **level 2 - learning** measuring what trainees have and have not learned, and how training has developed their skills, attitudes and knowledge, as well as their confidence and commitment, **level 3 - behaviour** measuring application of learning, ability to teach the new knowledge, skills or attitudes to other people, and the trainees awareness of their changed behaviour after the training, **level 4 - Results** evaluating long term outcomes, benefits, or final results linked to the training, and an extra level **level 5 - ROI** measuring economic return on investment of the training [Kirkpatrick, 1996].

Levels 1 and 2 can be assessed during or directly after the training (*hot evaluation*), while levels 3-5 are generally conducted a long period after the training to really measure effectiveness of results (*cold evaluation*). For this reliable evaluation purpose, both conducted surveys targeted not only fresh engineering students in last engineering year (*participants segment 1*), but also experimented alumni engineers with many years'experience (*participants segment 2*). The common point between both participants segments : they all followed compiling lecture or Hortensias pedagogy based approach with the first author in the past 15 years

Surveys were published online during one week, and participants answered anonymously to the survey questionnaires, and/or to free comments forms.

2) Survey 1 - General compiling lecture student satisfaction survey:

a) Survey 1 Hypothesis:
The general compiling lecture student satisfaction survey has the following null hypothesis :

H_{O_1} : *In 2020, it becomes not worthful to teach compiling in computer engineering majors.* .

which is tested against the alternative hypothesis:

H_{A_1} : *Being a relevant introduction to advanced topics (like model driven engineering, [semi/non/]structured data parsing, text analytics, natural language processing, chatbots), teaching compiling is still worth the cost in 2020 in computer engineering majors.* .

b) Survey 1 Participants:
63 engineers (21 engineering students -in last engineering year, and 42 graduated engineers working in industry) were recruited for the survey in a non-paid voluntarily basis. They are aged between 23 and 34 years old, and they have all followed the compiling module between 2008 and 2019 in their second engineering year when they were 22 years old so that permit to evaluate their opinion about compiling lecture experience. Survey 1 participants distribution is shwon in table XI [6], while figure 4 presents five questions of survey 1, their Kirkpatrick levels, and results.

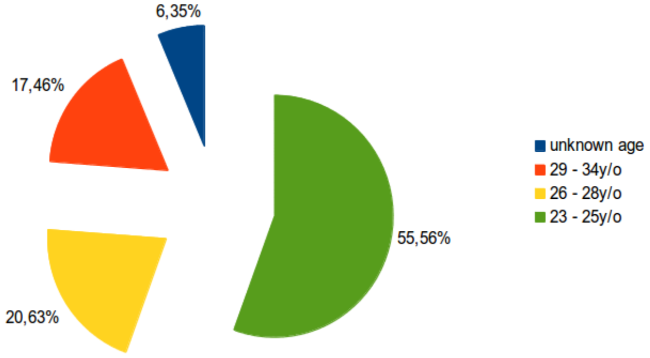


Figure 4: Survey 1 agregated participants distribution

c) Survey 1 Questionnaire and Results:
Table XII highlights survey 1 questions and results.

One may remark that 63.49% of compilers interest in the past (question Q1.1) is close enough to [Mernik and Zumer, 2003] result (69%) with a slight drop in interest. Compiling has many opportunities (like those stated in section II) where compiling lecture side effects 68% (question Q1.2) and return on investment can be measured (see question survey 2 - question Q2.10). Notice that compiling is still seen as return on investment lecture for all majors (88.89% for software engineering, and 68.25% for other computer engineering including embedded systems), however, compiling is seen to be matching more software engineering majors than other majors (6.35% No against 17.46% No for embedded systems, and 25.40% No for all computer engineering majors).

d) Survey 1 Interviews:
Among survey 1 questionnaires, participants were asked to

[6] grad, exp, and distrib stands for year of graduation, years'experience in industry, and distribution

give anonymously free comments or constructive proposals about compiling lecture. In the following, the result of these interviews will be given as pro and cons testimonials in favour and against formally expressed null hypothesis H_{O_1} : *In 2020, it becomes not worthful to teach compiling in computer engineering majors.*

H_{O_1} *Cons testimonials:*
1. *"My fascination with compiling, which certainly existed before the course, was greatly fuelled by this experience. Its effect has been mostly to open my mind to programming as an expression, alongside programming as a calculation. Since then, doing syntactic abstraction, or just finding more expressive ways of representing a domain, has become a pleasure that has often paid off. In a recent industrial project, I developed an application to express syntactic rules of accounting calculation where syntactic abstraction is limited to notation. This notation is chosen because it is already used by the accountants, thus allowing the professional accounting expert to contribute executable documents to the project, thus eliminating the errors that can slip into the comings and goings with the computer scientists. This use of syntactic abstraction as a scoring tool, however, tends to be lost, or to be limited to niche cases in companies, because the ability of the compiler to output actionable error messages is still limited, especially when the user is a beginner. This is partly explained by the great freedom that the medium of the text offers in the composition of a program, which represents as many opportunities to produce a wrong program. Companies tend to move towards specialized graphical data entry interfaces, which can make contextual validation more extensive, with more actionable error messages, precisely because they severely limit expressive freedom, by report to text. Of course, this power of graphical interfaces also removes the power of composition, which is one of the greatest strengths of the text. Representing the same possible combinations in a graphical interface, as textual expressivity allows, is already more complicated. However, more the composition has power, and more possibilities in the field of general use language (loops, procedural abstraction, etc.) become necessary, which explains that the DSLs (Domain Specific Languages) I encountered in business are often internal . At my current employer, these DSLs are often in Scala, which lends itself well enough with its flexible syntax. Companies, however, generally remain suspicious of external DSLs, because of the cost of maintenance.".*

2. *"Understanding Compilers is an essential part of understanding how the machine works and what computer scientists deal with. It opens our eyes to how to make a good compiler that works hard to optimize the machine code it generates".*

3. *"The idea of programming a compiler for a modern language is good. It allows the student to understand the mechanism of a compiler/interpreter which remains, in my opinion, a fundamental component in a computer system. At the same time the compilation course offers the student the opportunity to discover a new interesting language (it was mongo db for my case)".*

4. *"I think that compilers are very necessary for any software engineer".*

5. *"One of the most interesting fundamental lecture in the computer engineering".*

6. *"If there any project in ENSIAS that interested me, it would be the compiling project, and there is no doubt that it should remain at the curricula because of the huge impact it has on the understanding of many domains of computer science".*

H_{O_1} *Pro testimonials:*
1. *"Compiling is among technologies of 70s, and we need to change the way of teaching people and what we are teaching".*

H_{O_1} *Interviews Conclusions:*
Interviews argumentations are in majority against H_{O_1}, affirming thus that teaching compiling is still worth the cost in 2020 in computer engineering majors.

3) Survey 2 - Hortensias based pedagogy specific student satisfaction survey:

a) Study Hypothesis:
The Hortensias based pedagogy specific student satisfaction survey has the following null hypothesis :

> H_{O_2} : *As a result of the Hortensias platform compiling based pedagogy, there will be no significant impact neither on (i) motivating students to understand compiling nor on (ii) their learning curve in designing and implementing their own compiler/pseudo-code interpreter.*

.

which is tested against the alternative hypothesis:

> H_{A_2} : Hortensias *platform compiling based pedagogy, will significantly impact (i) students motivation and understanding of compiling concepts, and (ii) their learning curve in designing and implementing their own compiler/pseudo-code interpreter.*

.

b) Survey 2 Participants:
46 engineers (19 engineering students -in last engineering year, and 27 graduated engineers working in industry) were recruited for the survey in a non-paid voluntarily basis. They are aged between 23 and 31 years old, and they have all followed the Hortensias based pedagogy beside compiling module between 2011 and 2019 in their second engineering year when they were 22 years old so that permit to evaluate their opinion about Hortensias pedagogy experience. Survey 2 participants distribution is shwon in table XIV, while figure 5 shows survey 2 participants distribution by age interval.

c) Survey 2 Questionnaire and Results:
Table XIII presents ten questions of survey 2, their Kirkpatrick levels, and results.

One may remark that for 54.35% of participants, inner working of compilers was difficult before the training (question Q2.1), which is lower than 72% of [Mernik and Zumer, 2003] measure since students become more technology savvy. Understanding and programming compilers, pseudo code generator/interpreters was impacted positively by Hortensias based pedagogy for [76.26%, 82.61%] participants (questions Q2.2 – Q2.5). Notice that only [13.04%, 23.91%] think that Hortensias based pedagogy was not helpful to understand compiling, to change their design & programming behaviour, to develop their own compiler/generator/interpreter, and that learning Hortensias is not better that without it

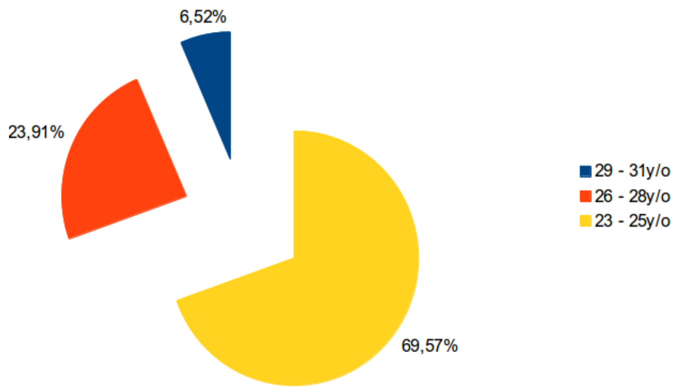


Figure 5: Survey 2 agregated participants distribution

(questions Q2.2 - Q2.9). 17.39% No (programming generator/interpreter) against 21.74% No (programming compiler) respectively for questions Q2.4 and Q2.5 can be explained by the fact that once Hortensias virtual machine provides one-address pseudo-code generator and interpreter, the student may neither reprogram nor enrich the VM generator/interpreter and rather use provided APIs as they are, while the student will be focused on reading/enriching/adapting more Hortensias compiler codebase and thus reducing compiler programming by analogy efforts. Finally, compiling lecture with Hortensias based pedagogy had good positive impact and return on investment in understanding and discovering new technology opportunities according to 71.74% of participants (question Q2.10) which is aligned with questions Q1.3 – Q1.5.

d) Survey 2 Interviews:

Among survey 2 questionnaires, participants were asked to give anonymously free comments or constructive proposals about compiling labworks and assignment based on Hortensias platform. In the following, the result of these interviews will be given as pro and cons testimonials in favour and against formally expressed null hypothesis H_{O_2} : Hortensias *platform compiling based pedagogy will be of no significant impact neither on motivating students to understand compiling nor on their compiling programming learning curve*.

H_{O_2} *Cons testimonials:*

1. *"With* Hortensias, *I liked the fact that we didn't have to start from scratch, this helped me focus on the core concepts of compiling".*

2. *"Honestly,* Hortensias *codebase is the most properly writing code in the C language that I have ever read. Because it is really hard to write a such a proper code with an imperative paradigm. Keep it going !"*

3. *"At this very moment, nearly everything we learned in* Hortensias *based compiling project is being used in our model driven engineering project. Moreover, the compiling lecture course and the project helped me so much to analyse how the code is being processed in my daily basis".*

4. *"Programming a compiler with* Hortensias *platform helped me a lot to understand compiler concepts".*

5. *"The compiling assignment based on* Hortensias *was one of the projects that I had the most fun developing".*

6. *"It was a pleasure for me to achieve the compiling project as I have understood how compilers work internally. My vision of programming has totally changed after this course. I highly encourage the teaching of* Hortensias *project to software engineering students".*

H_{O_2} *Pro testimonials:*

1. *"The* Hortensias *compiler project can be more beneficial to students if the design patterns are used to implement compiler concepts are exposed and explained to students like the visitor, and interpreter pattern".*

2. *"I think understanding compiling programming of a known language such* C *or* Java *would be much easier than a new language* Hortensias*".*

3. *"A deep dive into the actual inner workings of modern languages such as* Java, Kotlin, Swift *or* Scala *to develop* Hortensias, *instead of* C *language, would be helpful and bridge the gap between theory and industry practices to program compilers.".*

4. *"I suggest to add advanced compiling towards high performance computing hardware".*

H_{O_2} *Interviews Conclusions:*

Interviews argumentations are in majority against H_{O_2} hypothesis, affirming thus that Hortensias compiling pedagogy has impacted not only (i) students motivation and understanding of compiling concepts, but also (ii) their learning curve in designing and implementing their own compiler/pseudo-code interpreter.

4) Surveys' Conclusion: From the 15 questions and testimonials of both surveys 1 & 2, involving 63 ENSIAS alumni and students, conducted during one week, results are clearly in favour of the two alternative hypotheses : (i) *being a relevant introduction to advanced topics (like model driven engineering, [semi/non/]structured data parsing, text analytics, natural language processing, chatbots), teaching compiling is still worth the cost in 2020 in computer engineering majors general and software engineering and embedded systems in particular*, and (ii) Hortensias *platform compiling based pedagogy, had during 15 years significantly impacted (i) students motivation and understanding of compiling concepts, and (ii) their learning curve in designing and implementing their own compiler/pseudo-code interpreter* which is encouraging Hortensias compiling based pedagogy experience to be pursued and continuously improved in the future.

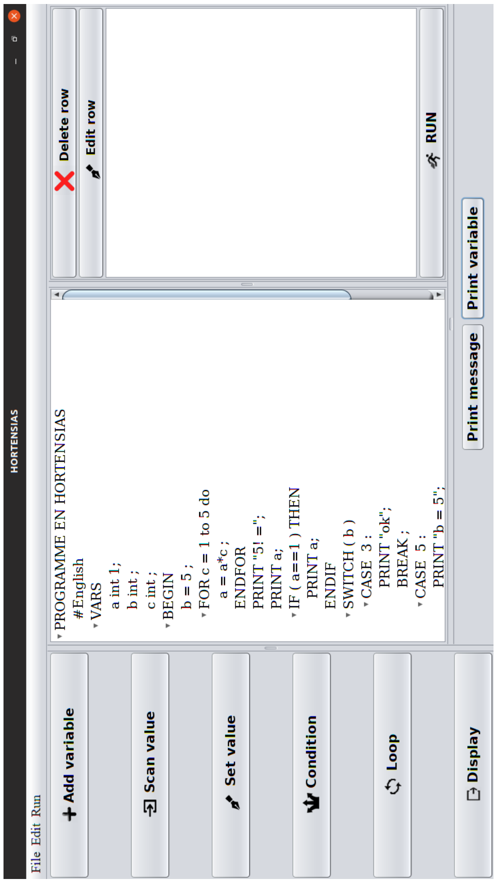
B. Contribution positioning – Literature review

Many works have highlighted the importance and the difficulty of compiling courses pedagogy [Neto et al., 1999], [Waite, 2006], [Aho, 2008], [Shehane and Sherman, 2014], [Kundra and Sureka, 2016], [Subramanian and Natarajan, 2019]. In this section, Hortensias will be positioned both with pedagogical compiling programming frameworks, and with pedagogical programming languages. Compiler design is a beautiful marriage of theory and practice – it is one of the first major areas of systems programming for which a strong theoretical foundation has developed that is now routinely used in practice [Aho, 2008].

If [Mernik and Zumer, 2003], [Xu and Martin, 2006], [Henry, 2005], [Sangal et al., 2018] propose a framework to teach compiling, [Mallozzi, 2005], [Demaille et al., 2008] advise to use a set of tools without proposing an integrated framework or core compiler.

What characterises Hortensias in tackling compiling engineering pedagogy is not its used techniques novelty, but its aptitude to cover concretely all concepts and compiler phases from scaning to code generation, and interpretation. Table XV compares Hortensias with languages and frameworks for teaching compiling.

Moreover, for beginner programmers, Hortensias offers the possibility to generate the program code by clicking on buttons instead of writing syntactically and semantically well-formed instructions. Figure 6 shows a view of Hortensias GUI, and table XVI compares Hortensias with languages and frameworks for teaching languages to beginner programmers.



```
PROGRAMME EN HORTENSIAS
  #English
  VARS
    a int 1;
    b int ;
    c int ;
  BEGIN
    b = 5 ;
    FOR c = 1 to 5 do
      a = a*c ;
    ENDFOR
    PRINT "5! =";
    PRINT a;
    IF ( a==1 ) THEN
      PRINT a;
    ENDIF
    SWITCH ( b )
    CASE  3 :
      PRINT "ok";
      BREAK ;
    CASE  5 :
      PRINT "b = 5";
```

Figure 6: Hortensias Framework GUI

VI. Conclusion and Discussions

Through Design Science Approach, this paper presents real world education experience in compiling pedagogy based on Hortensias language, and a pedagogical compiling laboratory platform as a design artifact. The evaluation of this design artifact were conducted through two surveys, consisting of 15 questions & testimonials, involving ENSIAS alumni and students, and conducted during one week, are clearly in favor of two hypotheses : (i) *teaching compiling being a relevant introduction to advanced topics (like model driven engineering, [semi/non/]structured data parsing, text analytics, natural language processing, chatbots), teaching compiling is still worth the cost in 2020 in computer engineering majors*, and (ii) Hortensias *platform compiling based pedagogy, had during 15 years significantly impacted (ii.1) students motivation and understanding of compiling concepts, and (ii.2) their learning curve in designing and implementing their own compiler/pseudo-code interpreter*. Moreover, a literature review has completed the surveys with positioning Hortensias platform with languages and frameworks for teaching compiling on one side, and on the other side with languages and frameworks for teaching languages to beginner programmers, before detailing formalised Hortensias defined grammars.

What are the implications of this study on both research and practice ? In the paper, authors show that teaching compiling is still worthful in 2020 when many world wide software engineering majors remove compiling from their curricula. This Omission is often justified from a mistaken perception that the study of compilers is now irrelevant to modern software engineering practice.

The paper states that teaching compiling in 2020 is a prerequisite for each software engineers to have a good abstraction capability and to be technology independent architect, to build structured mind representation meta-models of every as-is and to-be architecture, to have deeper understanding of what is behind black boxes. Compiling is also presented as a considerable pre-requisite for Model Driven Engineering discipline, and a perfect introduction to the Science of Text Algorithms.

This discussion is significant because with computer science transition to Global IT transitions : Big Data, Data sciences, Artificial Intelligence, Machine & Deep Learning, Computer Vision, Internet of Things, etc., many pedagogic instabilities and uncertainties are born in computer science in general and software engineering faculties and engineering schools.

The paper presents and evaluates real world education experience in compiling using Hortensias language, a pedagogical compiling laboratory platform based on a language compiler and a virtual machine. Hortensias code is open source and lecturers and students are all invited to contribute to improve it, live and enrich the Hortensias lecturer and student experience.

The paper presents an evaluation study based on two surveys targeting more sixty engineering students and alumni to evaluate impact on compiling course in their day to day needs and to assess the effectiveness of the tool in promoting desired pedagogical outcomes.

In fact, IT evolution accompanying the new digital/smart world requirements (e.g. computing scale-in/scale-out capabilities and architectures, new programming execution environments paradigm shifts, birth of many frameworks and programming languages, etc.) create a "healthy biodiversity IT environment". However, bridging the gap between layers evolving in different directions is a heavy task that should involve many research, development, and engineering efforts. Compiling teaching contribute to provide engineers with a good abstraction capability and technology independence within this continuously global evolving context, and thus insures a good global career evolution without getting stuck and being dependent to some specific language, framework, execution environment, hardware, or IT architecture.

What kind of work other researchers may do based on this study ? This paper states by arguing that teaching compiling in 2020 is a pre-requisite for each software engineers to have a good abstraction capability and to be technology independent architect, to build structured mind representation meta-models of every as-is and to-be architecture, to have deeper understanding of what is behind black boxes. Compiling is also presented as a considerable pre-requisite for Model Driven Engineering discipline, and a perfect introduction to the Science of Text Algorithms. The conducted surveys of this paper, stay at a high evaluation level. For curricula pre-requisites assessment, other researchers may be interested in *conducting empirical studies to confirm the positive impact of teaching compiling on :*

- *Computer Sciences skills*;
- *Programming languages skills*;
- *Model Driven Engineering skills*;
- *Text Analytics skills* (Science of Text Algorithms, Bag of Words Techniques, Natural Language Processing, etc.);
- *Big Data engineering, and analytics skills.*

What kind of work open source community may do based on this paper design artifact ? Hortensias code is open source. Lecturers and students are invited to contribute to improve it, live and enrich the Hortensias lecturer and student experience.

- *customise* Hortensias *compiler (front-end parser, semantic analyser, pragmatic analyser, IR generation, and back-end pseudo-code generator, optimiser, and virtual machine supporting interpretation) and use it as compiling pedagogy platform*;
- *customise* Hortensias *compiler error management internationalisation layer with a specific community language preserving languages diversity, and breaking language barriers for world wide young programmers* (some initiatives created even programming language with community specific vocabulary - see Irish programming language [Davey, 2020]).

What are limitations of this paper ? There are some limitations to this paper : (i) Conducting a detailed empirical evaluation of the effectiveness of each feature of Hortensias would enrich the pedagogical experience evaluation study, (ii) Conducting systematic literature review would provide an important support to the design artifact literature review evaluation approach, (iii) To address hot and cold evaluation, the evaluation surveys targeted a huge participants population from different generations, however, the evaluation survey approach still suffers from two biases : non-response bias, and voluntary response bias which may be improved with a systematic online obligatory periodic survey, (iv) Targeting several types of design artifact users (1) *beginner programmers*, (2) *programmers*, (3) *compiling course students*, and (4) *compiling course lecturers* create many other scientific and positioning problems not yet resolved. The author is aware of all those limitations, and will address them as interesting perspectives.

Acknowledgment

Acknowledgement must go to The Ministry of National Education, Higher Education, Staff Training, and Scientific Research, Morocco for accepting and supporting first author sabbatical leave to do research, and return to ENSIAS refreshed. First author also acknowledges his colleagues at ENSIAS maintaining the superb teaching and learning culture in the school in his absence.

Acknowledgement must also go to Prof. Khalid Benali, Université de Lorraine, France, and Prof. Salah Baïna, ENSIAS, Mohammed V University in Rabat, Morocco for they valuable remarks on this paper.

First author would also like to thank ENSIAS students Badr El Abbassi, and Ayoub Errahmi for their graphical programming on Hortensias, and the students Hamza Tamenaoul, Imane Merghichi, and Achraf Saddiki for their programming investigations on Hortensias artificial intelligence component by augmenting Hortensias AST and pseudo-code with genetic algorithms not detailed in this paper.

Acknowledgement must also go to all ENSIAS alumni and students who participated professionally and in a non-paid voluntarily basis to the evaluation surveys.

References

[Aho, 2008] Aho, A. V. (2008). Teaching the compilers course. *ACM SIGCSE bulletin*, 40(4):6–8.

[Baïna, 2020] Baïna, K. (2020). Hortensias programming language, https://gitlab.com/kbaina/hortensias.

[Baïna and Baïna, 2013] Baïna, K. and Baïna, S. (2013). User experience-based evaluation of open source workflow systems: The cases of Bonita, Activiti, jBPM, and Intalio. In *2013 3rd International Symposium ISKO-Maghreb*, pages 1–8. IEEE.

[Brachet, 2019] Brachet, P. (2019). Algobox, https://www.xm1math.net/algobox/.

[Burdick et al., 2013] Burdick, D. R., Ghoting, A., Krishnamurthy, R., Pednault, E. P. D., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., and Vaithyanathan, S. (2013). Systems and methods for processing machine learning algorithms in a MapReduce environment. US Patent 8,612,368.

[Chong et al., 2017] Chong, F., Franklin, D., and Martonosi, M. (2017). Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187.

[CMU, 2019] CMU (2019). Alice, https://www.alice.org/.

[Davey, 2020] Davey, E. (2020). Setanta - Teanga Rómhchlárúcháin as Gaeilge - Irish Programming Language https://docs.try-setanta.ie/.

[Demaille et al., 2008] Demaille, A., Levillain, R., and Perrot, B. (2008). A set of tools to teach compiler construction. In *ACM SIGCSE Bulletin*, volume 40, pages 68–72. ACM.

[Doerfert et al., 2019] Doerfert, J., Diaz, J. M. M., and Finkel, H. (2019). The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *International Workshop on OpenMP*, pages 153–167. Springer.

[Google, 2019] Google (2019). Blockly : A JavaScript library for building visual programming editors, https://developers.google.com/blockly/.

[Henry, 2005] Henry, T. R. (2005). Teaching compiler construction using a domain specific language. In *ACM SIGCSE Bulletin*, volume 37, pages 7–11. ACM.

[Hevner et al., 2008] Hevner, A. R., March, S. T., Park, J., and Ram, S. (2008). Design science in information systems research. *Management Information Systems Quarterly*, 28(1):6.

[Hibti et al., 2019] Hibti, M., Baïna, K., and Benatallah, B. (2019). Towards Swarm Intelligence Architectural Patterns – an IoT-Big Data-AI-Blockchain convergence perspective. In *The 4th International Conference On Big Data and Internet of Things (BDIoT'19), Tangier-Tetuan, Morocco*.

[Kirkpatrick, 1996] Kirkpatrick, D. (1996). Revisiting kirkpatrick's four-level model. *Training & Development*, 50(1):54–57.

[Kirkpatrick, 1975] Kirkpatrick, D. L. (1975). *Evaluating training programs*. Tata McGraw-Hill Education.

[Kundra and Sureka, 2016] Kundra, D. and Sureka, A. (2016). Application of Case-Based Teaching and Learning in Compiler Design Course. *arXiv preprint arXiv:1611.00271*.

[Lachaux et al., 2020] Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. (2020). Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

[Mallozzi, 2005] Mallozzi, J. S. (2005). Thoughts on and tools for teaching compiler design. *Journal of Computing Sciences in Colleges*, 21(2):177–184.

[Mernik and Zumer, 2003] Mernik, M. and Zumer, V. (2003). An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68.

[Milne and McAdam, 2011] Milne, A. C. and McAdam, E. V. (2011). Compilers, The Forgotten Subject ? *Innovation in Teaching and Learning in Information and Computer Sciences*, 10(2):32–40.

[MIT, 2019] MIT (2019). Scratch : Create stories, games, and animations, share with others around the world, https://scratch.mit.edu/research.

[Neto et al., 1999] Neto, J. J., Pariente, C. B., and Leonardi, F. (1999). Compiler construction-a pedagogical approach. In *Proceedings of the V International Congress on Informatic Engineering-ICIE*, volume 99, pages 1–12.

[Rademacher, 2019] Rademacher, G. (2019). Railroad Diagram Generator, https://www.bottlecaps.de/rr/ui.

[Sangal et al., 2018] Sangal, S., Kataria, S., Tyagi, T., Gupta, N., Kirtani, Y., Agrawal, S., and Chakraborty, P. (2018). PAVT: a tool to visualize and teach parsing algorithms. *Education and Information Technologies*, 23(6):2737–2764.

[Sellami et al., 2014] Sellami, R., Bhiri, S., and Defude, B. (2014). ODBAPI: a unified REST API for relational and NoSQL data stores. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 653–660. IEEE.

[Shehane and Sherman, 2014] Shehane, R. and Sherman, S. (2014). Visual Teaching Model for Introducing Programming Languages. *Journal of Instructional Pedagogies*, 14.

[Subramanian and Natarajan, 2019] Subramanian, V. and Natarajan, K. (2019). Simplification of compiler design course teaching using concept maps. In *Proceedings of Computer Science & Information Technology-CSIT*, volume 9, pages 91–102.

[Tillet et al., 2019] Tillet, P., Kung, H., and Cox, D. (2019). Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19. ACM.

[Waite, 2006] Waite, W. M. (2006). The compiler course in today's curriculum: three strategies. In *ACM SIGCSE Bulletin*, volume 38, pages 87–91. ACM.

[Xu and Martin, 2006] Xu, L. and Martin, F. G. (2006). Chirp on crickets: teaching compilers using an embedded robot controller. In *ACM SIGCSE Bulletin*, volume 38, pages 82–86. ACM.

VII. Appendix : Hortensias language formal basis – LL(1) grammar

A. Hortensias language Terminals

Let H be Hortensias grammar, T is the set of H grammar terminals constituted of :

- pragma : associativity orientation (#rightassoc or #leftassoc (default)), optimisation modes (#staticoptimiser or #dynamicoptimiser or no optimisation (default)), intermediary representation visualisation (#controlflowgraph), and error management language customisation primitives (#english (default), #french, #spanish, #german)
- program delimiters : begin, end
- idf : identifier
- constant terminals : inumber (integer), dnumber (double), cstring (character array), true, and false
- types terminals : int, bool, double, string
- other : ';', '(', ')', ':', '+', '-', '*', '/'
- '=' : initialisation, assignment, and for initial index value initialisation
- "==" : equality operator
- conditional control structure : if, then, endif, else
- switch control structure : switch, case, break, default, endswitch
- for control structure : for, to, do, endfor
- basic standard output printing : print.

B. Hortensias language Non Terminals

Let NT be the set of H grammar non-terminals : PRE_PROG, $PRAGMA\_LIST$, $PROG$, $TYPE$, $CONST$, $DECL$, $DECL\_AUX$, $DECL\_LIST$, $DECL\_LIST\_AUX$, $INST, INST\_LIST$, $INST\_LIST\_AUX$, $ASSIGN\_AUX$, $IF\_INSTAUX$, $SWITCH\_BODY$, $SWITCH\_BODYAUX$, $ADDSUB$, $ADDSUBAUX$, $MULTDIV$, $MULTDIVAUX$, AUX.

C. Hortensias language production rules

The following rewriting rules detail Hortensias H LL(1) grammar :

PRE_PROG → PRAGMA_LIST PROG
PRAGMA_LIST → **pragma** PRAGMA_LIST | ϵ
PROG → DECL_LIST **begin** INST_LIST **end**
INST_LIST → INST INST_LIST_AUX
INST_LIST_AUX → INST_LIST | ϵ
DECL_LIST → DECL DECL_LIST_AUX
DECL_LIST_AUX → DECL_LIST | ϵ
DECL → idf TYPE DECL_AUX
DECL_AUX → CONST **’;’** | **’;’**
TYPE → **int** | **bool** | **double** | **string**
CONST → **inumber** | **dnumber** | **cstring** | **true** | **false**
INST → **idf =** ASSIGN_AUX **’;’**
| **if ’(’ idf ==** ADDSUB **’)’ then** INST_LIST IF_INSTAUX
| **print idf ’;’** | **print cstring ’;’**
| **for idf = inumber to inumber do** INST_LIST **endfor**
| **switch ’(’ idf ’)’** SWITCH_BODY **default ’:’** INST_LIST **break ’;’ endswitch**
ASSIGN_AUX → ADDSUB | **true** | **false**
SWITCH_BODY → **case inumber ’:’** INST_LIST **break ’;’** SWITCH_BODYAUX
SWITCH_BODYAUX → SWITCH_BODY | ϵ
IF_INSTAUX → **endif** | **else** INST_LIST **endif**
ADDSUB → MULTDIV ADDSUBAUX
ADDSUBAUX → **’-’** MULTDIV ADDSUBAUX
ADDSUBAUX → **’+’** MULTDIV ADDSUBAUX
ADDSUBAUX → ϵ
MULTDIV → AUX MULTDIVAUX
MULTDIVAUX → **’*’** MULTDIV
MULTDIVAUX → **’/’** MULTDIV
MULTDIVAUX → ϵ
AUX → **idf**
AUX → **inumber** | **dnumber**

AUX → **'('** ADDSUB **')'**

Hortensias H grammar was kept incomplete to let the community complete it for pedagogical purpose. Hortensias framework developing has preferred rather to focus on delivering the whole compiling stack architecture compiler-optimiser-interpreter. In fact, many features can be easily added in the near future : at syntax and semantic level (e.g. functions, tables, while statement, more complete boolean expressions, rich for statement, etc.).

D. Hortensias language LL(1) grammar proof elements

Hortensias H grammar is LL(1) predictive. In fact, all not nullable non terminals of Hortensias H grammar are LL(1) predictive since all their right rules have distinct firsts. We only discuss in this section nullable non terminals : $LISTE\_PRAGMA$, $LISTE\_INSTAUX$, $SWITCH\_BODYAUX$, $LISTE\_DECLAUX$, $LISTE\_DECLAUX$, $ADDSUBAUX$, $MULTDIVAUX$

- $follow(LISTE\_PRAGMA) = \{ \text{idf} \}$, $first(LISTE\_PRAGMA) = \{\text{pragma}\}$
- $follow(LISTE\_INSTAUX) = \{$ end, endif, else, endfor, break $\}$, $first(LISTE\_INSTAUX) = \{ \text{idf} \}$
- $follow(SWITCH\_BODYAUX) = follow(SWITCH\_BODYAUX) = \{ \text{default} \}$, $first(SWITCH\_BODYAUX) = \{ \text{idf} \}$
- $follow(LISTE\_DECLAUX) = \{ \text{begin} \}$, $first(LISTE\_DECLAUX) = \{ \text{idf} \}$
- $follow(ADDSUBAUX) = \{';', ')'\}$, $first(ADDSUBAUX) = \{'-', '+'\}$
- $follow(MULTDIVAUX) = \{'+', '-', ';', ')'\}$, $first(MULTDIVAUX) = \{'*', '/'\}$
- $\forall X \in \{LISTE\_PRAGMA, LISTE\_INSTAUX, SWITCH\_BODYAUX, LISTE\_DECLAUX, LISTE\_DECLAUX, ADDSUBAUX, MULTDIVAUX\}$, $follow(X) \cap first(X) = \emptyset \Rightarrow H$ grammar is predictive LL(1).

VIII. Appendix : Hortensias pseudo-code language theoretical basis – LL(1) grammar

A. Hortensias pseudo-code language Terminals

Let P be Hortensias pseudo-code language grammar, T' is the set of P grammar terminals constituted of : delimiters (begin, end, ':'), idf (identifier), constant values (inumber, dnumber, cstring), binary arithmetic operators (add, mult, idiv, ddiv, sub), branching operators (jmp, jne, jg, jeq), branching label, variable operators (load, store), stack operators (dupl, swap), constant evaluation operator (push), printing operators (printi, prints).

B. Hortensias pseudo-code language Non Terminals

Let NT' be the set of P grammar non terminals : $PSEUDOCODE$, $DATA$, $DATA\_ITEM$, $DATA\_AUX$, $PSEUDOCODE\_INST$, $PSEUDOCODE\_LISTINST$, $PSEUDOCODE\_LISTINSTAUX$, and $CONST$.

C. Hortensias pseudo-code language production rules

The following rewriting rules detail Hortensias pseudo-code language LL(1) P grammar :

PSEUDOCODE → DATA **begin** **':'** PSEUDOCODE_LISTINST **end** **':'**
PSEUDOCODE_LISTINST → PSEUDOCODE_INST PSEUDOCODE_LISTINSTAUX
PSEUDOCODE_LISTINSTAUX → PSEUDOCODE_LISTINST | ϵ
DATA → DATA_ITEM DATA_AUX
DATA_AUX → DATA | ϵ
DATA_ITEM → **idf** CONST
CONST → **inumber** | **dnumber**| **cstring**
PSEUDOCODE_INST →
add | **sub** | **idiv** | **ddiv** | **mult**
| **jmp label** | **jne label** | **jg label** | **jeq label**
| **label** **':'**
| **load idf** | **store idf** | **push** CONST
| **dupl** | **swap**
| **printi** | **prints**

D. Hortensias pseudo-code language LL(1) grammar proof elements

Hortensias pseudo-code language P grammar is LL(1) predictive. In fact, all not nullable non terminals of Hortensias P grammar are LL(1) predictive since all their right rules have distinct firsts. We only discuss in this section nullable non terminals : $PSEUDOCODE\_LISTINSTAUX$, and $DATA\_AUX$.

- $follow(PSEUDOCODE\_LISTINSTAUX) = \{$ end $\}$, $first(PSEUDOCODE\_LISTINSTAUX) = first(PSEUDOCODE\_LISTINST) = first(PSEUDOCODE\_INST) =$ $\{$ add, idiv, ddviv, dupl, label, mult, printi, prints, cstring, sub, swap, store, jmp, jne, jg, jeq, load, push $\}$
- $follow(DATA\_AUX) = \{begin\}$, $first(DATA\_AUX) = first(DATA) = first(DATA\_ITEM) = \{ \text{idf} \}$
- $\forall X \in \{PSEUDOCODE\_LISTINSTAUX, DATA\_AUX\}, follow(X) \cap first(X) = \emptyset \Rightarrow P$ grammar is LL(1) predictive.

Hortensias **sample code**	**#rightassoc** **#staticoptimiser** (degenerated right AST) (with reverse post-order)	**#rightassoc** **#dynamicoptimiser** (degenerated right AST) (with deeper AST child first)	**#lefttassoc** **#staticoptimiser** (degenerated left AST) (with post-order)	**#lefttassoc** **#dynamicoptimiser** (degenerated left AST) (with deeper AST child first)
	(- (* a b) (+ c (- (/ d (* e f)) (/ g h))))		(- (+ (- (* a b) c) (* (/ d e) f)) (/ g h))	
.. begin .. value = a * b - c + d / e * f - g / h ; print value; end	begin: .. LOAD h LOAD g DDIV LOAD f LOAD e MULT LOAD d DDIV SUB LOAD c ADD LOAD b LOAD a MULT SUB STORE value LOAD value PRINTI end:	begin: .. LOAD f LOAD e MULT LOAD d DDIV LOAD h LOAD g DDIV SWAP SUB LOAD c ADD LOAD b LOAD a MULT SUB STORE value LOAD value PRINTI end:	begin: .. LOAD a LOAD b MULT LOAD c SWAP SUB LOAD d LOAD e SWAP DDIV LOAD f MULT ADD LOAD g LOAD h SWAP DDIV SWAP SUB STORE value LOAD value PRINTI end:	begin: .. LOAD e LOAD d DDIV LOAD f MULT LOAD b LOAD a MULT LOAD c SWAP SUB ADD LOAD h LOAD g DDIV SWAP SUB STORE value LOAD value PRINTI end:

Table IX: Static versus dynamic stack optimised pseudo-codes of a simple Hortensias program

Hortensias **sample code**	**Pseudo-code** generation	**Pseudo-code** interpretation output

REM Fibonacci
 	
REM grand pere Fibo(i=0) = 1
gp int 1;
 	
REM pere Fibo(i=1) = 1
p int 1;
 	
REM petit fils
pf int 0;
 	
i int;
 	
begin
 	
REM calcul de Fibo(i=1000)
for i = 2 to 1000 do
 	
pf = p + gp;
 	
gp = p;
 	
p = pf;
 	
rem print pf;
 	
endfor
 	
print pf;
 	
end
	gp 1.000000
	p 1.000000
	pf 0.000000
	i 0.000000
	begin:
	PUSH 2.000000
	STORE i
	for0:
	PUSH 1000.000000
	LOAD i
	JG endfor0
	LOAD p
	LOAD gp
	ADD
	STORE pf
	LOAD p
	STORE gp
	LOAD pf
	STORE p
	PUSH 1.000000
	LOAD i
	ADD
	STORE i
	JMP for0
	endfor0:
	LOAD pf
	PRINTI
	end:
		7033036771142276532204
		8724758141642699282707
		5657920012118404210516345
		9120474321865589459726
		7839624957874326566415
		6108317257606824774039
		2863294845000514502639
		8286231156065912982412
		5166666796383563765944
		827248
		64.000000

Table X: Compiled pseudo-code and interpretation output of a simple Hortensias program

grad. 20-	NA	09	10	11	12	13	14	15	16	17	18	19	20
age (y/o)	NA	34	33	32	31	30	29	28	27	26	25	24	23
exp. (y)	NA	11	10	9	8	7	6	5	4	3	2	1	**fresh**
distrib %	6.35	4.76	1.59	4.76	1.59	3.17	1.59	9.52	4.76	6.35	11.11	11.11	33.33

Table XI: Survey 1 detailed participants distribution

Level	Question	Yes	No	NA
-	**Q1.1.** Did the working of compilers interest you in the past ?	**63.49**%	36.51%	0.00%
-	**Q1.2.** Does language compiling/interpreting (or related domains : MDE, text analytics algorithms, data parsing scripting, NLP, or others) interest you now in your current industry activity ?	**68.25**%	30.16%	1.59%
4/5	**Q1.3.** Do you think that it is still worth the cost to teach compiling in 2020 in **Software Engineering** majors ?	88.89%	**6.35**%	4.76%
4/5	**Q1.4.** Do you think that it is still worth the cost to teach compiling in 2020 in **Embedded System Engineering** majors ?	68.25%	**17.46**%	14.29%
4/5	**Q1.5.** Do you think that it is still worth the cost to teach compiling in 2020 in **all Computer Engineering** majors ?	68.25%	**25.40**%	6.35%

Table XII: Survey 1 : questions, Kirkpatrick levels, and results

Level	Question	Y	N	NA
-	**Q2.1.** Was it difficult to understand the inner working of a compiler before Hortensias based compiling teaching methodology ?	**54.35%**	41.30%	4.35%
1	**Q2.2.** Was Hortensias of any help to a better understanding of compiler concepts ?	**82.61%**	13.04%	4.35%
1	**Q2.3.** Was Hortensias important for a better understanding of compiler concepts ?	**78.26%**	17.39%	4.35%
2	**Q2.4.** Was Hortensias important for helping you programming your own compiler ?	**76.09%**	21.74%	2.17%
2	**Q2.5.** Was Hortensias important for helping you programming your own pseudo code generator/interpreter ?	**78.26%**	17.39%	4.35%
3	**Q2.6.** Do you think that Hortensias has given you necessary concepts & methodology to move easily to other tools for compiling programming ?	76.09%	**21.74%**	2.17%
3	**Q2.7.** Do you think that your knowledge acquired thanks to Hortensias will be long lasting ?	73.91%	**19.57%**	6.52%
3	**Q2.8.** Do you think that learning compiling with Hortensias is better than without it ?	73.91%	**23.91%**	2.17%
3	**Q2.9.** Do you think that programming a compiler with Hortensias core support helped you improving your abstract/virtual machine understanding, computer science problem decomposition, data structure design, solution architecture, or programming skills ?	71.74%	**21.74%**	6.52%
4/5	**Q2.10.** Do you think that Hortensias helped you understanding advanced topics like model driven engineering, [semi/non/]structured data parsing, text analytics, natural language processing, chatbots, etc. ?	**71.74%**	26.09%	2.17%

Table XIII: Survey 2 : questions, Kirkpatrick levels, and results

grad. 20-	12	13	14	15	16	17	18	19	20
age (y/o)	31	30	29	28	27	26	25	24	23
exp. (y)	8	7	6	5	4	3	2	1	**fresh**
distrib %	2.17	2.17	2.17	10.87	6.52	6.52	15.22	13.04	41.30

Table XIV: Survey 2 detailed participants distribution

	Criteria	**Lisa** [Mernik and Zumer, 2003]	**Chirp** [Xu and Martin, 2006]	**GPL** [Henry, 2005]	**PAVT** [Sangal et al., 2018]	Hortensias [Baïna, 2020]
	principle	language specification based	compiling for robots	domain specific language	visual grammar simulator	extensible core compiler and API provider
	formal specification language based	syntax & semantic	syntax & semantic	syntax & semantic	syntax	
	compiler code	simulation	generation	generation	simulation	programming
	compiler programming based		x x			x x
	engine dependent	LISA tool	ANTLR	DSL	PAVT tool	flex
	runtime associativity customisation					x
	runtime optimisation customisation					static & dynamic
	pseudo-code generator	specification based	robot specific			virtual portable 1-@ bytecode
	parsing algorithm customisation				x x	
	eventual IR based interpretation	x x				x x
	pseudo-code interpreter		robot			x
API for	IR production		x			x
	semantic analysis		x			x
	error management					x
	code generation		x			x
	code optimisation					x
	code interpretation		x			x

Table XV: Hortensias *versus* languages and frameworks for teaching compiling

Criteria	**Scratch** [MIT, 2019]	**Blockly** [Google, 2019]	**Alice** [CMU, 2019]	**Algobox** [Brachet, 2019]	Hortensias [Baïna, 2020]
principle	coding for all	programming the web for all	virtual reality	maths	compiling learn/teach(ing)
Visual	x	x	x	x	in progress
Drag and drop	x	x	x		
use buttons				x	x
dynamic	x				
block-based	x	x	x	x	
graph-based					
imperative	x	x	x	x	x
concurrent	x	x	x		
web-based	-	x			
popular	x	x	x	French web	in progress
portable	x	x	x	x	x
compiler	x		x	x	extensible core
interpreter	x		x	x	x
machine language	x		-	-	x
object-oriented	x		x		
FOSS	x	x	x	x	x
event management	x	x	x		
exception handling	x				
web programming		x			
synchronous	x	x	x	x	x
pedagogy as a target	x	-	x	x	x
security	x	x	x	-	

Table XVI: Hortensias *versus* languages and frameworks for beginner programmers