NOVA
SCIENCE PUBLISHERS, INC.

THE CTM APPROACH FOR TEACHING AND LEARNING PROGRAMMING

Peter Van Roy

In: "Horizons in Computer Science Research. Volume 2"

Editor: Thomas S. Clary

ISBN: 978-1-61761-439-2 2011

400 Oser Avenue, Suite 1600
Hauppauge, N. Y. 11788-3619
Phone (631) 231-7269
Fax (631) 231-8175
E-mail: main@novapublishers.com
http://www.novapublishers.com

In: Horizons in Computer Science Research. Volume 2 ISBN: 978-1-61761-439-2
Editor: Thomas S. Clary, pp. 101-126

Chapter 5

THE CTM APPROACH FOR TEACHING AND LEARNING PROGRAMMING

Peter Van Roy*
Dept. of Computing Science and Engineering
Université catholique de Louvain (UCL), Belgium

Abstract

Since 2004 we have been teaching a second-year programming course to all engineering students at UCL. The course uses a uniform framework and introduces programming concepts, techniques, and models one by one, to overcome limitations in expressiveness as they appear (hence the acronym "CTM"). The course gives a complete formal semantics for the uniform framework in terms of a simple kernel language and abstract machine. The semantics is important since it gives a precise understanding of what the language does with no handwaving. For most students at UCL it is the only programming language semantics they see in all their studies. We teach the three main programming models or paradigms, namely functional, objected-oriented, and dataflow concurrent programming. We explain how object orientation and dataflow each extend functional programming with just one concept, respectively state and concurrency. We give a thorough presentation of data abstraction that includes objects, abstract data types, polymorphism, and inheritance. For the practical coding we use the Oz multiparadigm language, which has a uniform syntax for programming in each paradigm and for combining paradigms when needed. Oz has a high-quality open-source implementation, the Mozart Programming System. The course is the second part of the two-course sequence on programming in the UCL core engineering curriculum, where the first course is an introduction to programming based on Java. We find that this combination of two courses works well. This chapter gives our experience with this course: how it originated, what it contains, how we teach it and what the learning objectives are, and how it has become accepted in the computing science department and engineering school of UCL.

*E-mail address: peter.vanroy@uclouvain.be

1. Introduction

Computer programming is a unique mixture of practice and theory. A computer program is a long narrative written in a notation that resembles a formalized subset of a natural language and has a precise meaning in a clearly delimited domain. The main purpose of the narrative is to create an artifact that performs a series of actions to realize a desired specification.[1] Other purposes are that the artifact must be readable, maintainable, extensible, sufficiently fast, and parsimonious in its use of resources. Writing a program requires a mastery of algorithmic thinking, which is the ability to think fluently in terms of specifications and actions as two sides of the same coin. There is typically a large gap between the specification of what is required and its implementation as a program, and an important part of the art of programming is to bridge this gap by inventing abstractions.

Since the emergence of the programmable computer as a practical tool in the 1950s, half a century of research and development has gone into designing programming languages. These languages now support programs that can be quite complex, reaching sizes measured in millions of lines of code, written by large teams of human programmers over many years. Such languages are successful in part because they model some essential aspects of how to construct large systems. The languages are not just arbitrary constructions of the human mind, but they have a fundamental relationship to complex systems. This is why it is important for a programming course to teach languages in a fundamental way, where programming is seen as a way to construct complex systems.

These motivations underlie the CTM approach, which teaches programming by using a uniform framework and by introducing concepts one by one to overcome limitations in expressiveness. The approach is not built around a single concept, such as functions or objects, but rather on how different concepts are used together. We extend the language progressively with new concepts and we explain the new techniques that are made possible by each concept.

We have elaborated this approach since 1999 into a course and a textbook. The course is designed for the second year in a university or college setting, for all engineering students and not just students in computer science. The course teaches programming as an engineering discipline, that is, as a set of practical techniques based on a sound scientific foundation. It attempts to cover all the most important concepts, techniques, and paradigms within the limitations of twelve two-hour lectures. It reduces bias toward any particular paradigm or language by using a research language, Oz, that was designed to support all paradigms equally. The textbook was published in March 2004 and the course has been taught at UCL since 2003 up to the present day [13, 14].[2] The current course is called FSAB1402 and is part of the core curriculum taught to all engineering students at UCL [4]. This curriculum has two programming courses, FSAB1401 in the first year and FSAB1402 in the second year.

[1] It is curious that music has a deep relationship to computer programming, where the score corresponds to the program and the performance to its execution [2].

[2] The course materials are available for free download at `ctm.info.ucl.ac.be/fr`.

Structure of the Chapter

This chapter is organized as three parts:

- Section 2. explains the CTM approach: a uniform framework in which we introduce concepts one by one, a simple formal semantics for the framework, and a practical implementation that respects this semantics. Section 2.5. gives the measurable learning objectives of the course that are made possible by the CTM approach.
- Section 3. gives the course's historical background and explains how it has come to be accepted at the Louvain Engineering School of UCL (*École Polytechnique de Louvain* or EPL).
- Section 4. presents three course lectures in detail to show the distinctive flavor of the approach: named state and modularity, concurrency and multi-agent systems, and formal semantics.

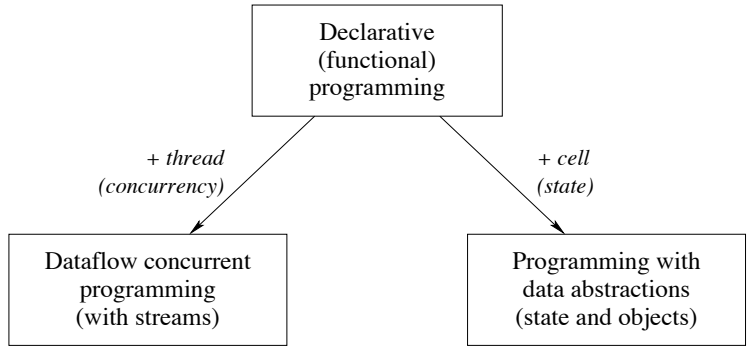


Figure 1. The three main programming paradigms taught in the course.

2. Course Principles

For most engineering students at UCL, this course is the deepest study of computer programming they will see in all their studies. This makes the course's foundation especially important. We base the course on a uniform framework that contains all the concepts taught in the course, that is defined precisely with a formal semantics, and that is supported by a practical software system.

2.1. Uniform Framework

The course can teach a large number of programming concepts in a short period because no information is repeated. We teach functional, object-oriented, and dataflow concurrent programming but we do not need three sets of concepts. The object-oriented and dataflow paradigms are extensions of the functional paradigm, with just one extra concept for each (cells and threads, respectively). Concepts used in functional programming remain

useful for object-oriented programming and dataflow concurrent programming. Nevertheless, each paradigm is complete and supports all programming techniques specific to the paradigm. Figure 1 shows the relationships between the three paradigms. We note that the course is relatively free from temporary fashions; it is focused on deep concepts that remain useful. For example, the recent popularity of multi-core processors highlights the importance of dataflow concurrency, which is well-suited for that architecture.

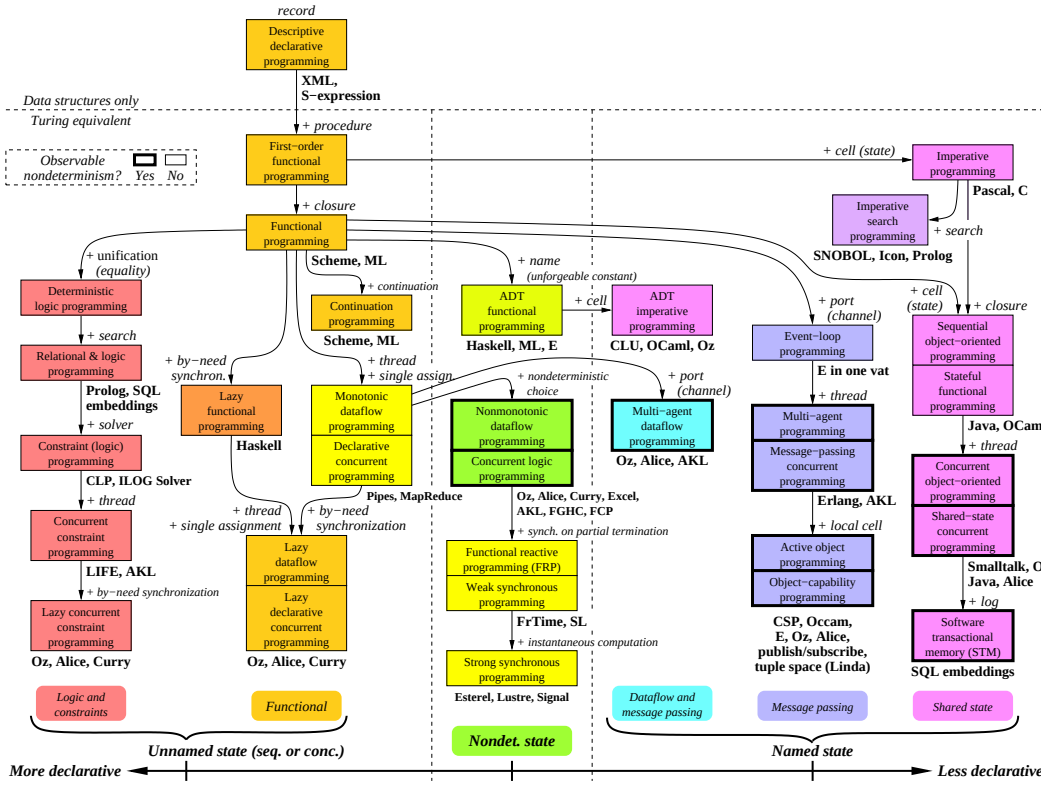


Figure 2. Programming paradigms organized according to the creative extension principle.

The uniform framework is applicable to many more paradigms than just these three. Figure 2 shows a more complete diagram of programming paradigms that subsumes Figure 1 [15]. Programming paradigms appear as a kind of epiphenomenon, depending on which concepts one uses. This leads to multiparadigm programming in a completely natural way. The diagram in Figure 2 is not shown to the students in our course. However, other universities have organized courses based on CTM that cover a bigger part of this diagram. Some of these courses are listed on the CTM website [13]. The diagram is organized according to the *creative extension principle*: a new concept is added when programs start getting complicated for reasons unrelated to the problem being solved. More precisely, a new concept is added to the language when programs require nonlocal transformations to encode the concept in the language. If the new concept is added to the language, then only local transformations are needed. This principle was first defined by Matthias Felleisen [5].

Table 1. Programming concepts taught in the course

Lecture	Concepts
1. Basic concepts	Programming paradigm, instruction, value, interactive interface, declaration, identifier, single-assignment variable, environment, assignment, lexical scope, function, integer, floating point, atom, conditional, induction, recursion, kernel language, procedure
2. Integer recursion	Contextual environment, kernel language, declarative model, recursion, invariant, accumulator, tail recursion, tail call optimization, free identifier, higher-order programming, specification, list, type, type variable
3. List recursion	Loop, nested loop, invariant, contextual environment, kernel language, list, grammar rule, EBNF notation, type variable, syntactic sugar, tuple, pattern matching
4. Computational complexity	Tuple, record, tree, execution time, active memory, memory consumption, garbage collection, best/worst/average case, asymptotic analysis, elementary operation, kernel language, big $O/\Omega/\Theta$ notation, recurrence equation, Moore's law, P and NP problems, traveling salesman problem, satisfiability, NP-completeness
5. Records and trees	Tuple, record, equality, kernel language, ordered binary tree, search tree, global condition, goal-oriented programming
6. State and data abstraction	State, cell, multiple-assignment memory, kernel language, structure equality, name equality, cell semantics, modularity, encapsulation, data abstraction, abstract data type, object
7. Programming with abstract data types	Indexed collection, array, dictionary, hash table, matrix, specification, formal semantics, mathematical induction, abstract machine, environment
8. Formal semantics	Kernel language, abstract machine, operational semantics, semantic instruction, semantic stack, initial configuration, execution, sequential composition, adjunction, restriction, instruction semantics, procedure definition semantics, procedure value (lexically scoped closure), procedure call semantics
9. Objects, classes, polymorphism, inheritance	Object, procedural dispatch, class, polymorphism, responsibility principle, inheritance, interface, composition, substitution principle, static link, overriding, dynamic link, self, multiple inheritance
10. Java and exceptions	Tail call semantics, kernel language, Java, primitive type, reference type, parameter passing, abstract class, final class, Java interface, exception, error containment principle, execution context
11. Concurrency and multi-agent systems	Concurrency, declarative concurrency, message passing, shared state, free (unbound) variable, dataflow execution, thread, execution order, nondeterminism, race condition, thread scheduling, thread priority, thread suspension, fair scheduling, stream, agent, pipeline, transformer, producer/consumer
12. Declarative concurrency	Deterministic concurrency, transparency, dataflow, multi-agent programming, programming paradigms, kernel language

2.2. Important Concepts

Table 1 lists the most important programming concepts taught in the course. These concepts complete and generalize the concepts learned in the first-year course: object-oriented

programming (by giving a more general view of data abstraction including abstract data types), the Java language (by explaining class hierarchies and parameter passing), recursive data types and algorithms (by defining and using lists and trees), and reasoning about programs (through computational complexity and formal semantics).

Of all the course's concepts, higher-order programming, data abstraction, and concurrency are particularly important. The course introduces higher-order programming early on, as part of a programming example in which a general iterator is specialized to calculate square roots using Newton's method. The general iterator is parameterized with functions for termination detection and state transformation. Higher-order programming with lexical scoping appears naturally when these functions are defined.

Data abstraction is a broad concept that is often taught too narrowly. For example, pure object-oriented languages such as Smalltalk use one kind of data abstraction, but many other kinds are possible. There are two orthogonal axes: whether the abstraction uses state (destructive assignment) or not, and whether the abstraction is bundled (object) or not (abstract data type). This gives four possibilities in all. Each has its own trade-offs and real-world metaphors. We explain the trade-offs between objects and abstract data types, and show how to use polymorphism with both. For example, we show how Java objects combine characteristics of both objects and abstract data types to get some of the advantages of both. It is important for the students to understand the issues involved and why languages make the choices they do.

Concurrency is often considered a difficult concept, and for this reason it is usually left to third or fourth-year courses. The difficulty is apparent for the most widespread form of concurrency, namely the shared-state concurrency as used in Java's threads and monitors. This form is hard to program in and subject to race conditions. But shared-state concurrency is not the only form of concurrency. There exist much simpler, deterministic forms of concurrency. We teach declarative dataflow concurrency, which has no race conditions. It is expressive enough to program parallel tasks, streams, and pipelines. We consider it a good preparation for the more sophisticated forms of concurrency that are used in Internet applications and multicore programming. We note that Gary Leavens has also used this approach with CTM to teach concurrent programming [10].

2.3. Formal Semantics

We give a formal semantics for the uniform framework of the course. The semantics covers all the concepts taught in the course. It is an essential part of the course. Without it, programming becomes a set of recipes in a cookbook instead of an engineering discipline. We introduce the semantics gradually from the beginning of the course. In the first lecture, we explain the concepts of lexical scope, variable identifier (textual name), variable in memory, and environment (frame) which connects identifiers and variables in memory.

We give the semantics of a practical language, i.e., a language with a rich syntax that supports many programming idioms, by translating it into a kernel language. The kernel language consists of a small number of programmer-significant concepts. This makes it easy to understand by practicing programmers. Nevertheless, the kernel language is formally defined as a process calculus; we give its operational semantics in terms of an abstract machine. The abstract machine explains all concepts and techniques used in the course and

allows to reason about program behavior, execution time, and execution space. With the abstract machine we can derive asymptotic complexity, explain garbage collection, and show how tail call optimization keeps stack size constant.

The kernel language is factored: it gives the semantics of each programming concept separately. This lets us show exactly what each concept adds and lets us give the semantics of each paradigm separately, uncontaminated by other paradigms. The semantics supports whatever degree of formality best suits the problem: from the most rigorous formal methods to the most intuitive craftsmanship. Even if programmers do not use the semantics directly, its mere existence ensures that there are no unpleasant surprises.

The two most similar approaches to our kernel language approach are the foundational calculus and the virtual machine. A foundational calculus, like the λ-calculus or π-calculus, reduces programming to a minimal number of primitive concepts. This is especially useful for the theoretical study of computation. A virtual machine defines a language in terms of its implementation on an idealized machine. This is especially useful for language implementors and compiler writers. The problem with both approaches is that any realistic program written in them will be cluttered with technical details about language mechanisms. The kernel language approach avoids this clutter by choosing concepts wisely. The kernel languages are designed for human beings. Students can use them to execute programs with pencil and paper.

2.4. Software Support

The course uses the Oz language throughout. Oz is a fully developed research language that is supported by a complete implementation, the Mozart Programming System [11]. Oz is designed to support multiple paradigms in a factored way: concepts can be used separately without having to mix them. Because of its multiparadigm design that combines a large number of concepts, Oz has an unusual syntax compared to more widespread syntaxes such as those inspired by C. We do not consider this a serious problem. Syntax is just a thin surface layer over what really matters, namely the semantics. Furthermore, not depending on a C-like syntax makes the course less sensitive to current fashions. The Oz syntax is simple and factored, and not difficult to learn. In the course we organize the first two practical sessions in the engineering school's computer labs, to help the students quickly master the Oz syntax and the Mozart programming environment.

Once the students understand the Oz syntax, many advantages are gained. Using a single language instead of several (e.g., Java, Prolog, Haskell, and Erlang, which are often used in courses on programming paradigms) makes it easier to show the deep relationships between the paradigms as well as reducing the administrative burden for students and teachers (since only one system needs to be installed and learned instead of many). The students understand this: they consistently give the course high marks and they accept the use of Oz. They understand that using Oz lets the course go much farther than Java. The course presents higher-order functional programming, object-oriented programming and general data abstractions, dataflow concurrency, and a simple formal semantics for all these concepts. To my knowledge, this coverage is not possible with any other language than Oz.

Java is not a good language for teaching programming, for the following reasons. Java does not support functional or dataflow concurrent programming in a simple way. Java's

support for concurrent programming and its semantics are unnecessarily complex for students. Furthermore, Java is strongly biased toward object-oriented programming, which is only one narrow segment of a wide spectrum of programming concepts. Many important industrial applications, such as distributed and telecommunications applications, require very different concepts including dataflow concurrency and functional programming.

Mozart is a production-quality implementation that supports all the code examples in CTM. Mozart is available without charge under an Open Source license. It exists for various flavors of Unix and Windows and for Mac OS X. Mozart is actively developed and maintained by the Mozart community. Since 2007 there is a second tool, the iLabo, that is specifically targeted to support the course. The iLabo is a front end to Mozart that contains all the examples of the French version of CTM and has the same structure as the book [12].

2.5. Learning Objectives

We have defined a set of measurable learning objectives that are attainable within the length of a one-semester course. The learning objectives were determined based on the following meta-objectives:

- **Thinking algorithmically** Continued advances in computing technology make it clear that computer usage, which is already high, will continue to increase for the foreseeable future. Engineers of all disciplines therefore need to understand how to design and understand programs that satisfy specifications while being readable, maintainable, extensible, efficient, and parsimonious in resource usage.

- **Thinking with abstractions** Good design in all disciplines depends on the ability to think clearly with abstractions: the ability to reason correctly about a system that consists of several layers of abstraction and the ability to define new abstractions to simplify a problem's solution [6]. This is especially important for computer programming because of the large size and complexity of programs.

- **Continued study** The course should be a good foundation for advanced study of programming. For example, this covers the ability to continue with courses on programming language theory and with advanced courses in symbolic programming systems (such as Mathlab, Mathematica, Maple) and in programming languages that implements one of the course's paradigms (including Java, Erlang, Scheme, and Haskell, but not Prolog which is out of scope for the course).

To determine the actual learning objectives, the above meta-objectives were confronted with the discipline of programming as presented in CTM [13]. The objectives were then refined over several years as the course has matured. Given the time constraints of the course, there is a tension between the desire to broaden the learning objectives and the desire to keep the course simple. We find that strengthening the unified framework has allowed us to improve both simultaneously. This gives us the following set of learning objectives for the current version of the course:

- **Programming concepts** The ability to define with precision and to use appropriately in small programs (up to one page) the principal programming concepts. The abil-

ity to use the right programming techniques for each concept. The list of concepts covered by the course is given in Table 1.

- **Programming paradigms** The ability to define with precision the three principal programming paradigms, namely functional programming, object-oriented programming, and concurrent dataflow programming, with the concepts they contain and the properties they give to programs. The ability to choose the right paradigm to solve a given problem and to write a program in that paradigm to solve the problem (up to several pages). Finally, the ability to combine paradigms inside a program when necessary, with a justification of the design.

- **Syntax and recursion** The ability to define the syntax of a recursive data structure (both linear and arborescent) using EBNF (Extended Backus-Naur Formalism) grammar rules. The ability to write a program to traverse a recursive data structure to do a calculation on the structure or to create another structure.

- **Semantics** The ability to define a formal operational semantics of a simple programming language in one of the paradigms of the course, with the right mathematical concepts. The ability to execute a program manually (on paper) according to this semantics. The ability to justify certain programming rules of thumb (such as tail call optimization) based on the semantics. The ability to calculate the asymptotic complexity, both temporal and spatial, in big $O/\Omega/\Theta$ notation, for the best case, worse case, and average case for simple programs (except if the calculation requires solving nontrivial recurrence equations or more than an elementary knowledge of probability theory).

- **Concurrent programming** The ability to define the basic concepts of concurrent programming, in particular, the concepts of thread, dataflow synchronization, nondeterminism, scheduler, and fair scheduling. The ability to explain the three principal paradigms of concurrent programming (shared state, message passing, and declarative concurrency) with their main properties. The ability to write small programs in the declarative (dataflow) concurrent paradigm.

These learning objectives are student-centered: they indicate clearly to students what the course expects from them and they can be evaluated (for example, in an examination).

3. Historical Background and Acceptance

3.1. Origins of the Course

The course's foundation is based on a large collaborative research effort in programming languages that started in the late 1980s undertaken by three research groups. The three groups were located at Saarland University, Germany (headed by Gert Smolka), the Royal Institute of Technology, Sweden (headed by Seif Haridi), and the DEC Paris Research Laboratory (headed by Hassan Aït-Kaci). These partners and others initiated the European ACCLAIM project (1991–1994), which had a major effect on this work. The

goal was to study programming languages by factorizing them into their independent concepts. The research started with logic and constraint programming, as an outgrowth of work on concurrent logic programming, but quickly branched out to embrace all important programming paradigms. Several languages were designed, among which Oz, AKL, and LIFE were the most prominent. The ideas of AKL and LIFE were eventually merged into Oz, which became the most visible result of this research.

In the summer of 1999, Seif Haridi and Peter Van Roy had the insight that the clean and factored design of Oz could be used as the basis of a programming course. They started writing a book on programming. The book was published in March 2004 by MIT Press as *Concepts, Techniques, and Models of Computer Programming* [13]. This book has become known as CTM.

During the writing of CTM (mid 1999 to mid 2003), Seif Haridi, Christian Schulte, and Peter Van Roy all taught programming courses based on this material to students at the Royal Institute of Technology (KTH), the National University of Singapore (NUS), and UCL. Seif Haridi taught the course Datalogi II at KTH in Fall 2001 and Christian Schulte taught it in Fall 2002 and Fall 2003. Seif Haridi taught a similar course during his sabbatical at NUS in Fall 2003. Peter Van Roy has continued to teach and refine a programming course at UCL since the Spring 2003 semester up to the present day. The course, currently called FSAB1402, is taught in French to all engineering students at UCL as part of the core curriculum in the three-year Bachelor's degree in engineering [4]. There is a choice of two textbooks: CTM or its French translation which was published in Sep. 2007 together with its custom software support, the iLabo environment [14, 12]. The French translation covers exactly the material of the course, which is about one third of CTM. There is another course at UCL, currently called INGI1131, which is a successor to FSAB1402 for computer science majors. It is focused on concurrent programming and covers another third of CTM.

3.2. Acceptance at the Louvain Engineering School (EPL)

An early version of this course, named LINF1251, was taught to computer science students in the second year of the four-year licentiate degree during the Spring 2003 and Spring 2004 semesters. Subsequently, there was a course opening in the Fall 2004 semester for all second-year students at EPL. When the course was first taught to all engineering students, there was skepticism in the computing science department whether it would be successful. The course was provisionally accepted for two years, to be followed by an evaluation to see whether it should be continued. After two years, the course was accepted without any problems and it has been taught each fall semester ever since (six times so far including the Fall 2009 semester) [4].

One reason for the initial skepticism was a problem of perceived scientific credibility of computer science with respect to other engineering disciplines. Since all engineering disciplines use computers as tools, the non-computer science disciplines at UCL tended to consider computer science as a technical field but not a scientific one. This perception was initially strengthened by the computing science department itself, a young department which taught programming using mainstream languages such as C++ and Java. Paradoxically, this was done to improve the usefulness of computer science for the other engineering disciplines. The CTM approach has partially changed this mindset, although a lingering

unease remains about the use of a non-mainstream language, Oz, to illustrate the concepts. This was done to support the focus on programming concepts, since no single mainstream language has the broad coverage and simple semantics of Oz (see also Section 2.4.).

In the current version of the course, there are around three hundred students each year. The didactic team consists of the author, four teaching assistants, and eight or nine student monitors. There are many ways to teach the course. We currently organize the course as a two-hour lecture and a two-hour practical session each week for thirteen weeks. The lecture combines transparencies, interactive programming using the Mozart system, and explanations on the blackboard. The practical sessions are organized in groups of 24 students. The students have take-home exercises that are graded each week, an optional midterm, a programming project, and a final exam. The project counts for one fourth of the final grade and the final exam counts for three fourths. The take-home exercises, if completed successfully, add a small bonus to the final grade. The midterm can improve the final score through a weighted average in which it counts for one third. The project is done in groups of two students during the last third of the course and cannot be redone. Part of the project grade comes from an interview of each group.

3.3. Two Essential Insights

Since the initial teaching of this course, we have gained two important insights about how to teach programming [8]:

- *Importance of the second year*. We find that the second year is a good time to teach a programming course based on programming concepts. In the first year, students are not mature enough (the course is too abstract for them). In the third and later years, students get conservative (they get too attached to the languages they have learned so far). In the second year, they have enough maturity to understand the concepts and enough openness to appreciate them. At UCL, the core curriculum for all engineering students has a first-year course based on Java, FSAB1401, that gives an introduction to programming and object-orientation. The second-year course is a natural continuation of FSAB1401 [4].

- *Importance of formal semantics*. We recommend that students be taught programming language semantics in the second year. This can succeed if: (1) the semantics requires very little mathematical baggage, for example just sets, sequences, and functions, (2) the semantics is factored so it can be taught incrementally for all paradigms, and (3) the semantics is simple and uncluttered so that students can work out a program's execution with paper and pencil. The abstract machine semantics of CTM is one example that fits these conditions. Despite its simplicity, the students consider the semantics the hardest part of the course. They are used to highly technical courses in continuous mathematics such as the differential and integral calculus, but they have never before in their studies been exposed to the style of discrete mathematics used in a programming language semantics. The semantics is important because it allows the students to think precisely about how programs execute. Without a formal semantics, the execution of programs remains vague and mysterious. For this reason, we consider that students in any technical field should be taught a language semantics at

least once in their careers. Considering the omnipresence of computers and programming, it needs to be a part of engineering culture as much as traditional mathematical subjects such as algebra and calculus.

3.4. Comparison with Other Approaches

The ACM computing curriculum gives several approaches to teach introductory programming [7]. It has approaches focused on different concepts such as objects and functions. Since object-oriented programming can be explained with functional programming extended with named state, we find it natural to start with functional programming and then extend it to object-oriented programming. We show the techniques of functional programming and then we show how adding named state allows to add more techniques. For example, object-oriented programming allows richer forms of data abstraction than functional programming.

MIT has recently switched from Scheme to Python for their first-year course. This was much discussed on Internet forums, including the programming languages forum Lambda the Ultimate [9]. In our experience it is not necessarily a bad thing to put Scheme beyond the first year. First-year students are not yet ready for the abstract thinking that such a course requires. The book used in MIT's first-year course, namely *Structure and Interpretation of Computer Programs* (SICP) [1], is dense with concepts and programming techniques in a similar style to CTM. SICP uses an interpreter-based approach as opposed to CTM's kernel language approach which is based on translation. CTM has a greater coverage of concurrency whereas SICP covers meta-interpreters.

At EPL there are two computer science courses in the core curriculum: a first-year course that uses Java, called FSAB1401, and a second-year course explained in this chapter that uses Oz, called FSAB1402. Java is a mainstream language that the students must learn at some point. Putting it in the first year is a reasonable solution since the first-year course does not go far enough to show its limitations. Using Oz in the second-year course allows the course to cover a much wider set of concepts. The second year also has a project course based on Java (currently only for students taking computer science as a major or a minor). With these two courses, students get both sides of the coin in the second year: an improved understanding of programming concepts (a long-term educational goal) and an improved understanding of practical programming with Java (a short-term educational goal).

4. Example Lectures and Highlights

To give the flavor of our approach and how it differs from more standard approaches, we give three example lectures from the course. We give them with text, code, and figures in similar fashion to an actual lecture. We have chosen three lectures:

- Named state and modularity (part of lecture 6). This lecture introduces *cells*, which are variables that can be assigned many times (destructive assignment variables) and shows why they are good for modularity. It makes the bridge between the functional paradigm and the object-oriented paradigm.

- Concurrency and multi-agent systems (part of lecture 11). This lecture introduces a particularly simple and well-behaved form of concurrent programming, namely declarative dataflow concurrency. This form of concurrency has no race conditions and is as simple to program as the functional paradigm.

- Formal semantics (part of lecture 8). This lecture gives the formal semantics of the complete language in terms of a kernel language and an abstract machine. This model is general enough to cover all the paradigms of the course and simple enough that the students can work out semantic examples with pencil and paper.

Introduction to the Oz language The three lectures all use the Oz language. We give a brief introduction to Oz, explaining just enough to understand the lectures. Variable identifiers in Oz start with capital letters and must be declared before use. Function and procedure calls are delimited with braces { and }, because the parentheses (and) are used for records. Here is a simple way to define the factorial function:

```
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

All compound instructions, such as function definition (**fun**) and conditional (**if**), start with a keyword and terminate with **end**. The basic data structures are lists, tuples, and records. These are values; once created they cannot be modified. Lists are written as dotted pairs with a notation derived from Prolog. The empty list is denoted by `nil` and `H|T` denotes a list with head `H` and tail `T`. Pattern matching is done with a **case** statement. We can define the list append function as follows:

```
fun {Append Xs Ys}
   case Xs
   of X|Xr then X|{Append Xr Ys}
   [] nil then Ys end
end
```

Because Oz has single-assignment variables, the above append function is tail-recursive. We can see this by translating `Append` into the simple core language of Oz, the *kernel language*:

```
proc {Append Xs Ys Zs}
   case Xs of X|Xr then
      local Zr in
         Zs=X|Zr
         {Append Xr Ys Zr}
      end
   else
      case Xs of nil then
         Zs=Ys
      else
         raise typeError end
      end
   end
end
```

The variable `Zr` is created unbound and passed to the recursive call `{Append Xr Ys Zr}`. This allows us to place the recursive call last. The kernel language has only procedures; functions are encoded as procedures with an additional output argument. All variables used during calculations are visible in the kernel language, including function outputs and local variables. The formal semantics of Oz explained in the course is given for this kernel language.

4.1. Named State and Modularity (part of lecture 6)

Our first lecture introduces explicit state. This extends the declarative model with destructive assignment variables. We will show that explicit state is essential for modularity: the ability to change part of a system without changing the rest.

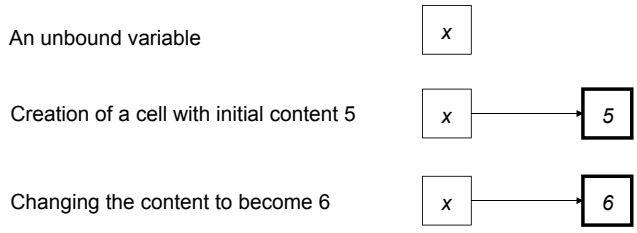
Explicit state added to the language In the declarative model, there is no notion of time. Functions are mathematical functions: for the same arguments the results are always the same. This is not true for entities in the real world. In the real world, there is time and time is accompanied by change. Organisms change their behavior with time, they grow and learn. How can we model these changes in a program? We can add a kind of *abstract time* in a program. Abstract time is simply a sequence of values, which we call a *state*. More precisely, a state is a sequence of values calculated successively, which contain the intermediate results of a computation. According to this definition, the declarative model can also have state! For example, consider this definition:

```
fun {Sum Xs A}
   case Xs of nil then A
   [] X|Xr then {Sum Xr A+X}
   end
end
{Browse {Sum [1 2 3 4] 0}}
```

(`Browse` is a one-argument procedure that displays its argument.) At each recursive call to `Sum`, the two arguments `Xs` and `A` have the following values:

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

This gives a sequence of values, which is exactly a state. We call this an *implicit state* because we did not change the programming language. The state exists in the programmer's mind. State can also be explicit, in other words, we extend the language. This extension allows us to express directly a sequence of values in time. We will call this extension a *cell*. A cell has a content that can be changed. A state is a cell's sequence of contents.



A cell is a container that has two parts: an identity and a content. The identity is constant (the "name" or "address" of the cell). The content is a variable in the single-assignment memory. The cell's content can be changed. This replaces the variable by another variable. It does not change the variable's value!

```
X={NewCell 5}
{Browse @X}
X:=6
{Browse @X}
```

We add the cell as a new concept to the kernel language. A cell has three operations:

- `X={NewCell I}` creates a new cell with initial content `I` and binds `X` to the cell's name.
- `X:=J` gives the cell a new content `J`, if `X` is bound to the cell's name.
- `Y=@X` binds `Y` to the content of the cell, if `X` is bound to the cell's name.

Explicit state is needed for modularity We say that a system (or a program) is *modular* if it is possible to update part of the system without updating the rest. Here, "part" can have many meanings: function, procedure, component, module, class, library, package, and so forth. We give an example to show how explicit state can be used to make a modular system. We will see that this is not possible in the declarative model.

Consider the following scenario. There are three software developers, let us call them P, U1, and U2. P has developed a module M that provides two functions F and G:

```
fun {MF}
   fun {F ...}
      % Definition of F
   end
   fun {G ...}
      % Definition of G
   end
in
  moduleRec(f:F g:G)
end
M={MF}
```

In this code, `MF` is a component that creates module `M` when it is instantiated. The module `M` has two functions `F` and `G` accessed through the module's interface, which is the record `moduleRec(f:F g:G)`. U1 and U2 are both users of the module M.

Now, U2 has an application that takes a lot of computation time. U2 would like to extend the module `M` to count the number of times that function `F` is called by its application. U2

asks P to do this without changing M's interface. Surprise! This is impossible to do in the declarative model because F has no memory of its previous calls. The only possible solution in this model is to change F's interface by adding two arguments Fin and Fout:

```
fun {F ... Fin Fout} Fout=Fin+1 ... end
```

Here Fin is the number of times F is called before this call, and Fout is the number of times F is called after. The rest of the program has to make sure that the output Fout of one call to F is the input Fin of the next call to F. This means that the interface to M has changed. All users of M, even U1 (who did not ask for any change), must change their programs. Neither U1 nor U2 are happy.

The solution to this problem is to use explicit state, namely a cell.

```
fun {MF}
   X={NewCell 0}
   fun {F ...}
      X := @X+1
      % Definition of F
   end
   fun {G ...}
      % Definition of G
   end
   fun {Count} @X end
in
  moduleRec(f:F g:G c:Count)
end
M={MF}
```

When the module M is created, this creates a cell referenced by X inside the module. Because of lexical scoping, the cell is hidden and is only visible from inside the module. The function F (called as M.f from the outside) has not changed its interface. A new function M.c is available to get the value of the count, but this can be ignored.

We conclude this example by comparing the declarative model with the imperative model (which has cells). In the declarative model, a component never changes its behavior (if it is correct, it stays correct). But updating a component is difficult: it means that often the interface has to be changed too. In the imperative model, a component can be updated without changing its interface (the program is modular). But a component can change its behavior because of previous calls (it may break, for example). It is sometimes possible to combine the advantages of both models: use explicit state to help with updating, but be careful never to change the behavior of a component.

4.2. Concurrency and Multi-Agent Systems (part of lecture 11)

Our second lecture introduces concurrent programming. We will see how to do dataflow execution with threads and streams. We will build simple multi-agent systems including producer/consumer systems and pipelines (similar to Unix pipes).

The real world is concurrent: it consists of activities that evolve independently. The computing world is concurrent too. There are three levels of concurrency in a computing system:

- A distributed system consists of computers linked by a network. A concurrent activity in a distributed system is called a computer.
- An operating system of a computer consists of the software program that provides the basic environment available to all other programs. A concurrent activity in an operating system is called a process. Processes have independent memory spaces.
- A process consists of an executing program with its memory space. A concurrent activity in a process is called a thread. Threads share the same memory space.

In Web browsers each window typically corresponds to one concurrent activity, which can be a thread or a process depending on how the Web browser is built.

There is a strong link between independence and concurrency: any two activities that are independent are concurrent. How can we write a program with two independent activities? It needs support from the programming language. In a concurrent program, there can be several activities executing at the same time. These activities can communicate (one activity passes information to another) and synchronize (one activity waits for another).

Declarative concurrency There are three main approaches to program with concurrency. By far the simplest of the three is *declarative concurrency*. This is the one we will see today. The two other approaches are outside the scope of this course. We mention them briefly:

- Message-passing concurrency. Activities send each other messages, similar to packets on a network or postal mail. This approach is also relatively simple. It is used in the Erlang language.
- Shared-state concurrency. Activities share data and try to work together without interfering with each other. Whenever an activity uses some data, it locks the data to keep other activities from interfering. This is by far the most complicated approach. Unfortunately, for historical reasons this is the most widely used one. It is implemented in Java via the *monitor* concept, which allows multiple activities to coordinate access to shared data.

We study these approaches in other courses, such as INGI1131 in the third year bachelor's program of computer science at UCL. Today we will focus on declarative concurrency. It is based on a simple idea: using single-assignment variables to synchronize data. Assume we have a variable `X` that is not bound to a value. Let us try to use it in an operation:

```
local X Y in
   Y=X+1
   {Browse Y}
end
```

The addition `X+1` has an unbound argument `X`. What does it do? Nothing! Execution waits just before the addition operation until `X` is bound. As we shall see, this behavior is at the heart of declarative concurrency.

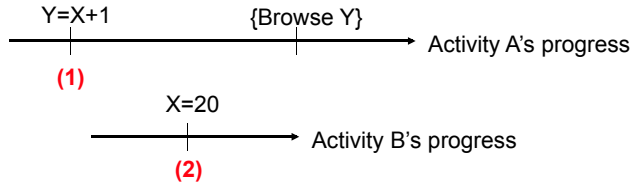
We can compare this behavior to other languages. Different programming languages do different things with variables that are not initialized. In early versions of C, the addition

continues but X has a "garbage" value (content of machine memory at that instant). This is very bad since the result can vary from one execution to the next. In Java, the addition continues with X bound to 0 (if X is the attribute of an object and has type integer). This is better since the result is always the same. In Prolog, execution stops with an error message. In Java, the compiler detects an error if X is a local variable. In Oz (our language), the execution waits just before the addition and can continue when X is bound (dataflow execution). In constraint programming, the equation "Y=X+1" is added to the set of constraints and execution continues by solving the constraints. This is the most sophisticated form of declarative programming, which is outside the scope of this course. If you are curious about this, you can take the course INGI2365 on constraint programming at UCL.

Now, let us see how execution can continue with our example:

```
declare X
local Y in
   Y=X+1
   {Browse Y}
end
```

(The **declare** X instruction declares X as a global variable in the interactive Mozart interface.) If someone could bind X, then execution would continue. But who can bind X? Answer: another concurrent activity! If the activity does X=20 then the addition will continue and 21 will be displayed. This behavior is called *dataflow execution*.



Activity A waits patiently at point (1) just before the addition. When activity B does X=20 at point (2), then activity A can continue. If activity B does X=20 before activity A arrives at point (1), then activity A will continue without waiting. In both cases, the same result (21) is displayed.

Threads In the previous examples, what we called an "activity" is a sequence of instructions in execution. Technically this is called a *thread*. Each thread is an independent sequential execution. There is no order defined between two threads, that is, we cannot tell which thread will execute first. The system is free to choose this. The system guarantees that each thread will receive a fair share of the processor's calculating capacity. Two threads can communicate if they have a shared variable, such as the variable X in the previous example.

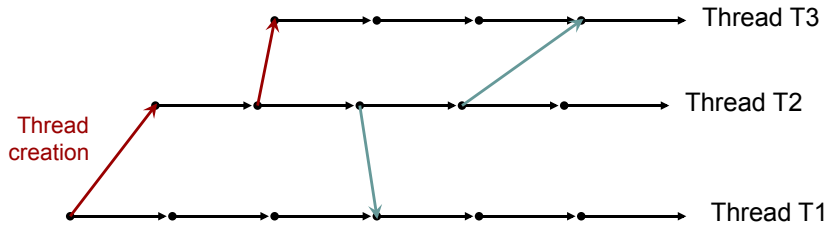
In Oz it is easy to create a thread. Any instruction <s> can run in a new thread by executing **thread** <s> **end**. For example:

```
declare X
thread {Browse X+1} end
thread X=1 end
```

What does this program do? There are several possible executions, but they will all eventually display the same result: 2!

Total order, partial order, and nondeterminism Concurrent activities can execute "at the same time". We can imagine that all the threads are really executing in parallel, each with its own processor but sharing the same memory. This is not really the case, but it is a good way to picture what is going on. In reality, the threads will share the processor that is executing the process containing the threads. In a sequential program, which executes in one thread, all execution states are in a *total order*: there is an order between each pair of execution states.

In a concurrent program, all execution states of the same thread are in a total order. The execution states of the whole program are in a *partial order*: some pairs of execution states are not ordered.



What does the following program do:

```
declare X
thread X=1 end
thread X=2 end
```

The execution order of the two threads is not determined. `X` will be bound to 1 or 2, but it is not specified which. The other thread will have an error (an exception will be raised) since a variable cannot be bound to two values. This uncertainty about what is done is called *nondeterminism*. The system is free to choose either possibility. Let us try the same example with cells:

```
declare X={NewCell 0}
thread X:=1 end
thread X:=2 end
```

Again, the execution order is not determined. The cell `X` will first be assigned one value and then the other. When both threads have finished, `X` will have content 1 or 2, but it is not specified which. This time there is no error. Again, the uncertainty about what is done is called nondeterminism. The programmer does not know what will happen, because the two activities are independent. The system is free to choose either possibility.

Nondeterminism is bad if it can affect the outcome of a program, i.e., different executions of the same program can give different outputs. We call this *observable nondeterminism*. This is possible in both of the examples given above. Avoiding this is not always easy. It is especially difficult for programs that use both threads and cells, which is called

shared-state concurrency. Unfortunately, many popular languages such as Java and C++ use this form of concurrency. The usual approach to control the nondeterminism is to use a programming concept called a *monitor*. This is outside the scope of this course. It is explained in the course INGI1131. The declarative model has a major advantage over shared-state models: it has no observable nondeterminism. A declarative concurrent program that does not end in error always gives the same result. This is a remarkable property that deserves to be widely known.

Streams and agents Let us write some programs using declarative concurrency. We introduce two programming techniques, streams and agents. We define a *stream* as a list whose tail is an unbound variable:

```
S=a|b|c|d|S2
```

`S` is a stream. A stream can be extended indefinitely with new elements by binding its tail and closed if desired by binding the tail to `nil`. A stream can be used as a communication channel between two threads. The first thread adds elements to the stream and the second thread reads the stream. Here is a program that displays all the elements of a stream:

```
proc {Disp S}
   case S of X|S2 then {Browse X} {Disp S2} end
end
declare S
thread {Disp S} end
```

All elements added to the stream will be displayed. For example, this fragment will display `a`, `b`, `c`, and `d`:

```
declare S2 in S=a|b|S2
declare S3 in S2=c|d|S3
```

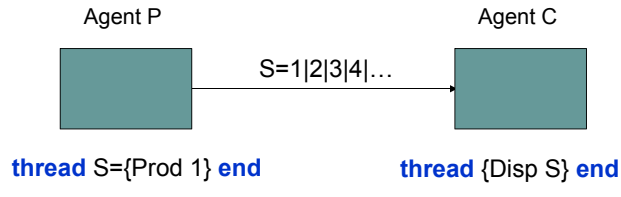
With streams we can build a simple concurrent producer/consumer system. A *producer* generates a stream of data:

```
fun {Prod N} {Delay 1000} N|{Prod N+1} end
```

Here the `{Delay 1000}` causes the thread to wait for 1000 ms to slow down execution enough so we can see what happens. A *consumer* reads the stream and uses it. Here is a simple producer/consumer system:

```
declare S
thread S={Prod 1} end
thread {Disp S} end
```

The structure of this system can be shown nicely as a diagram:

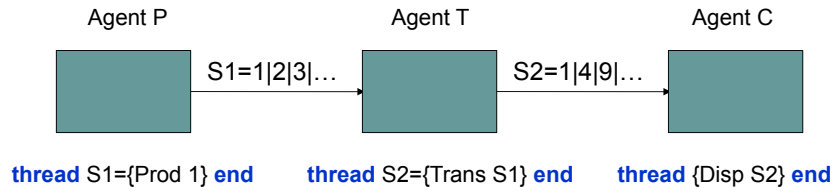


Each rectangle is an *agent*, which we define as a concurrent activity with one (or more) communication channels. The two agents P and C communicate using the stream S. We can add other agents in between P and C. For example, here is a transformer that reads a stream and outputs a modified stream:

```
fun {Trans S}
   case S of X|S2 then X*X|{Trans S2} end
end
```

Here is a producer/transformer/consumer program with three agents:

```
declare S1 S2
thread S1={Prod 1} end
thread S2={Trans S1} end
thread {Disp S2} end
```



The pipeline technique illustrated by this example is very useful. It has become a ubiquitous tool in operating systems since its invention in Unix.

4.3. Formal Semantics (part of lecture 8)

Our third lecture is the most mathematical of the whole course. In previous lectures we have set the stage by introducing already many of the concepts that we need for the semantics: instructions, the kernel language, identifiers, variables, the single-assignment memory, and environments. As you will see, just two additional concepts are needed to define a complete semantics: the semantic stack and the execution state.

The formal semantics of a language gives a mathematical explanation of how any program executes. With this explanation we can understand why a program does what it does, we can reason about correctness, computational complexity (both in time and space), and memory management (how the stack and heap are managed, including garbage collection). For example, we can use the semantics to understand how tail call optimization reduces the stack space used by a program. We will define the semantics in terms of an idealized computer called an abstract machine. Starting from a program, there are two steps to take:

- First translate the program into the kernel language. The kernel language is a simplified form of the language that contains all the essential concepts but no syntactic short-cuts for the programmer. Technically, the kernel language makes explicit all operations and variables used in the execution.

- Then execute the kernel language program with the abstract machine.

The kernel language of the declarative model has the following syntax definition:

```
<s>             ::=  skip
                   | <s>1 <s>2
                   | local <x> in <s> end
                   | <x>1 = <x>2
                   | <x>1 = <v>
                   | if <x> then <s>1 else <s>2 end
                   | {<x> <y>1 ··· <y>n}
                   | case <x> of <p> then <s>1 else <s>2 end
<v>             ::=  <number> | <procedure> | <record>
<number>        ::=  <int> | <float>
<procedure>     ::=  proc {$ <x>1 ··· <x>n} <s> end
<record>, <p>   ::=  <lit>(<f>1:<x>1 ··· <f>n:<x>n)
```

This is an EBNF grammar notation where <s> designates an instruction, <x> and <y> designate identifiers, and <v> designates a value, which can be a number, procedure, or record. Patterns <p> in a case statement have the same syntax as records.

To execute an instruction <s> written in kernel language, we need to combine it with an environment E and a single-assignment memory σ that contains variables and the values they are bound to. The instruction <s> is combined with an environment E to make a *semantic instruction* $S = (\texttt{<s>}, E)$. An *execution state* (ST, σ) combines a stack $ST = [S_{n-1}, \cdots, S_2, S_1, S_0]$ of semantic instructions together with a memory σ. Each instruction on the stack has its own environment and all instructions share the same memory (why?). An *execution* is a sequence of execution states $(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \cdots$.

To execute a program <s> we form the initial execution state $(ST_0, \sigma_0) = ([(\texttt{<s>}, \emptyset)], \emptyset)$. This combines the instruction <s> with an empty environment $\emptyset$ to give the first semantic instruction (<s>, $\emptyset$). The semantic stack containing this instruction is then paired with an empty memory $\emptyset$.

Execution in the abstract machine Each execution step removes the top of the stack and executes the instruction according to a rule that defines the semantics of the instruction. When the stack is empty then execution stops. To show how this works we give the execution of the following instruction:

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```

The initial execution state is:

```
([(local X in local B in B=true if B then X=1 else skip
end end end, ∅)], ∅)
```

The instruction on the top of the stack is **local** <x> **in** <s> **end**, where <x>=X and <s>=**local** B **in** B=**true** **if** B **then** X=1 **else skip end end**. The rule for executing this instruction creates a new variable x in memory and an environment $\{\texttt{X} \rightarrow x\}$ to refer to it. We can define this rule as follows:

Rule for **`local`** instruction
Replace: ([(**`local`** <X> **`in`** <S> **`end`**,E), (rest of stack)], σ) by: ([(<S>,$E + \{$<X>$ \rightarrow x\}$), (rest of stack)], $\sigma \cup \{x\}$)

where x is a fresh variable (which does not occur in σ). Applying this rule to the initial execution state gives:

([(**`local`** B **`in`** B=**`true`** **`if`** B **`then`** X=1 **`else skip end end`**, $\{$X $\rightarrow$ $x\}$)], $\{x\}$)

The top instruction on the stack is again a **`local`**, so we apply the rule again:

([(B=**`true`** **`if`** B **`then`** X=1 **`else skip end`**, $\{$X $\rightarrow x$, B $\rightarrow b\}$)], $\{x, b\}$)

The top instruction is now a sequential composition <S>$_1$ <S>$_2$. The rule for a sequential composition splits it into two stack entries:

([(B=**`true`**, $\{$X $\rightarrow x$, B $\rightarrow b\}$),
(**`if`** B **`then`** X=1 **`else skip end`**, $\{$X $\rightarrow x$, B $\rightarrow b\}$)], $\{x, b\}$)

The top instruction is now the binding B=**`true`**. The rule for a binding simply binds B's variable to **`true`**:

([(**`if`** B **`then`** X=1 **`else skip end`**, $\{$X $\rightarrow x$, B $\rightarrow b\}$)], $\{x, b =$ **`true`**$\}$)

Continuing in this way, we eventually end up with an empty stack. This gives the final execution state:

([], $\{x = 1, b =$ **`true`**$\}$)

Semantics of each instruction For each instruction of the kernel language, there is a rule that specifies how it executes in the semantics. Each instruction takes an execution state (semantic stack and memory) and returns another execution state. In the above example we showed how to use the rules for the **`local`** instruction, the sequential composition instruction, and the binding instruction. All the other instructions have similar rules. The most complicated rules are for the procedure definition and procedure call.

We have given the semantics of the functional paradigm. To support the object-oriented and dataflow concurrent paradigms, we extend the declarative model with cells, exceptions, and threads. We give a semantics to each of these language concepts by extending the abstract machine. Cells are defined in terms a new memory, a *multiple-assignment store*, that sits beside the single-assignment store. Exceptions are defined in terms of an *execution context*, which is a contiguous part of the semantic stack from the top down to the beginning of the context. Execution contexts can be nested. When an exception is raised, an execution context is removed by emptying the stack down to the beginning of the context and replaced by an instruction that does the exception handling. Threads are defined by giving each thread one semantic stack and by allowing individual semantic stacks to suspend. A semantic stack *suspends* if the instruction on the top cannot continue because a variable is not bound. Binding this variable on another semantic stack will allow the original one to continue.

Mastering the semantics The formal semantics is the most mathematical part of the course. To understand how it works, you need to do exercises using pencil and paper. Be especially careful with procedure definitions and procedure calls, since they are a bit tricky. A procedure definition stores a contextual environment together with the procedure's code in memory. A procedure call creates an environment by combining the contextual environment with the procedure's arguments.

5. Conclusion

We have presented a mature second-year course that teaches computer programming based on a uniform framework, introducing concepts one by one within the framework as they are needed. The course teaches programming as an engineering discipline, combining a theoretical foundation with practical techniques. It is part of the core curriculum for all engineering students at the Université catholique de Louvain since 2004. The course is accepted at the Louvain Engineering School and the computing science department and by the students, who give it high marks in course evaluations. Course materials and a course book are available in French and English [14, 13]. Student-centered learning objectives for the course are given in Section 2.5. and are formulated to allow straightforward student evaluation.

The course is based on a unified framework that covers all concepts taught in the course. We give the framework a formal operational semantics based on a kernel language and an abstract machine. This operational semantics is used to reason about program correctness and to understand how programs execute. It justifies the programming techniques used in the course (such as tail call optimization) and the techniques used for computational complexity. The programming semantics and techniques cover the three most important programming paradigms, namely functional programming, object-oriented programming, and declarative dataflow programming. The course uses a multiparadigm language, Oz, which allows presenting all the paradigms using a single syntax in a natural way. Oz has a simple formal semantics and a high-quality implementation, Mozart [11], which both contribute to the success of the course.

5.1. Applying the CTM Approach to a First-Year Course

We have experimented with applying the CTM approach of introducing concepts one by one in a uniform framework to a first-year programming course. For the first year we focus less on semantics and more on concurrency. The idea is to start immediately with multi-agent programming in a small microworld. Each time the limits of a microworld are reached, we add a new concept to create a richer microworld. We have developed a complete set of lecture notes for this approach and tested it on several small groups of students [3].

The progressive multi-agent microworld approach introduces a series of microworlds that rapidly advance toward concepts that are typically only seen in much more advanced courses. The students learn multi-agent programming, asynchronous message passing, graphic interfaces, higher-order programming, software components, fault tolerance, and distributed systems. We hope to eventually redo the complete programming curriculum based on these ideas.

Acknowledgments

We thank Elie Milgrom and Pierre Dupont for their insightful comments that substantially improved this chapter. We thank Olivier Bonaventure and Charles Pecheur, who teach FSAB1401, for their insights and support for teaching computer science in the core curriculum. We thank the Dept. of Computing Science and Engineering at UCL for their support in the development of the course FSAB1402 in the UCL Engineering School.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*, 2nd edition. The MIT Press, Cambridge, MA, 1996.

[2] Gérard Assayag, Andrew Gerzso, and Peter Van Roy. "Foreword." *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.

[3] Isabelle Cambron, Mathieu Cuvelier, Gregory de le Vingne, Maxime Romain, Cécile Toint, and Peter Van Roy. *La Programmation en Première Année Basée sur l'Enrichissement Progressif de Micromondes Multi-Agents* (*First-Year Programming Based on Progressive Enrichment of Multi-Agent Microworlds*) (in French). Master's thesis. See www.info.ucl.ac.be/ pvr/micromondes.html.

[4] École Polytechnique de Louvain (*Louvain Engineering School*) (EPL). Bachelier en Sciences de l'Ingénieur, Orientation Ingénieur Civil (*Bachelor in Engineering Science, Orientation Graduate Engineer*). Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2009. See www.uclouvain.be/prog-2009-fsa1ba.html.

[5] Matthias Felleisen. On the Expressive Power of Programming Languages. In *3rd European Symposium on Programming (ESOP 1990)*, pages 134–151, May 1990.

[6] Michael French. *Invention and Evolution: Design in Nature and Engineering*, 2nd Edition. Cambridge University Press, 1994.

[7] Joint Task Force on Computing Curricula, IEEE Computer Society and ACM. "Computing Curricula 2001: Computer Science", Final Report, Dec. 15, 2001.

[8] Lambda the Ultimate. "Insights on Teaching Computer Programming." LtU discussion 1195. See lambda-the-ultimate.org/node/1195. Dec. 2005.

[9] Lambda the Ultimate. "Why Did MIT Switch from Scheme to Python?" LtU discussion 3312. See lambda-the-ultimate.org/node/3312. May 2009.

[10] Gary T. Leavens. Use Concurrent Programming Models to Motivate Teaching of Programming Languages. In *Programming Languages Curriculum Workshop*, May 2008, Cambridge, MA. Also report CS-TR-08-04a, University of Central Florida.

[11] Mozart Programming System version 1.4.0. See www.mozart-oz.org. July 2008.

[12] ScienceActive, iLabo, 2007. See www.scienceactive.com.

[13] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. Course website: ctm.info.ucl.ac.be/en. The MIT Press, Cambridge, MA, 2004.

[14] Peter Van Roy and Seif Haridi. *Programmation: Concepts, Techniques et Modèles* (in French). Course website: ctm.info.ucl.ac.be/fr. Dunod Éditeur, Paris, France, 2007.

[15] Peter Van Roy. "Programming Paradigms for Dummies: What Every Programmer Should Know." *New Computational Paradigms for Computer Music* , Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.