# CS3110 Fall 2013 Lecture 1: Introduction, OCaml syntax, Evaluation (8/29)

Robert Constable

## 1 Lecture Outline



## 2 Course Mechanics

The course web site has been available since mid August. You will find a great deal of useful information there. We will use Piazza to communicate as a group. Two instructors are listed: Robert Constable, a long term Cornell CS professor, and Michael George, about to graduate with a Cornell PhD in programming languages and security.

The large course staff of undergraduates, masters students and PhD students are highly qualified, e.g. all of the undergraduates on the staff have taken an offering of this course in the past and done very well. The most senior course staff have all done the job before. The current staff is listed with photographs. We hope to add a few more members.

Compared to last semester the enrollment has essentially doubled causing us to alter the delivery of the material a bit. Here are the key elements of the week to week operation.

- We require **two recitations per week** for most weeks. We might cancel a recitation now and then in favor of more office hours.
- Only **two prelims** will be given, the dates are listed on the web site.
- There will be **six programming assignments**/problem sets. We will not accept late programming assignments and problem sets. We will deal with medical issues as they arise and may create alternative assignments.
- There *may be* **short quizzes**, true/false, multiple choice, and short answer. If so they will be given in lecture and recitations and announced in advance.

There are previous course notes for about 75% of the material in this course, starting with fall 2009 notes. There is an on line resource book, *Introduction to OCaml*, written by a Cornell CS PhD, Jason Hickey, one of my former PhD students. You can freely download it from the course web site. There are other books on OCaml available, some in French.[1]

The OCaml reference manual is also on the web site. It covers the entire language and is a bit dense and terse.

The web page for the course describes more fully the course mechanics, e.g. programming assignments and problem sets, prelims, quizzes, and a final exam. Please read that material. It discusses the role of recitations, consulting, office hours and so forth. Note, we have added extra office hours to help deal with the large enrollment. The recitation leaders will also go over elements of course mechanics. We all meet together on a regular basis to discuss the course.

The programming assignments are the core of the course. They bring the concepts to life, they teach advanced programming skills, and they allow dedicated students to stand out. We tap the very best students in this course to join the course staff for future offerings. Moreover, the CS faculty recruit student help for their research projects from this course. In particular I lead a large research project in computer security, and our group is always looking for experienced dedicated students to join us.

[1] Jason used OCaml to create the *MetaPRL proof assistant* used to produce verified computer programs and to produce formalized mathematics.

**Academic Integrity** We remind you of the academic integrity policies. If you cheat we will find out, and the consequences will be severe.

# 3 Course Content

Our course will teach the *OCaml programming language* and how to program in the functional style. You will learn not only a new programming language but some new ways to think about the programming process and how to think rigorously about programming languages. These skills will help you understand computer science better, and they might help you get a job in the information technology industry where the ideas we teach are highly valued.

OCaml is a member of the *ML family* of programming languages which includes Standard ML (SML) [13] which we previously used in this course. The family also includes Microsoft's $F^\sharp$, and the original language in this family, Classic ML [4], a very small compact language from 1979 still used in some research projects including mine.

Knowing a language in the ML family is an indication that you were exposed to certain modern computer science ideas that have proven very valuable in writing clean reliable programs and in designing software systems.

The ML family is also an excellent basis for presenting the topics in data structures and the analysis of algorithms because *the semantics of the language can be given in a simple mathematically rigorous way as we will illustrate* [6, 11, 9, 5]. This mathematical foundation will allow us to study in some depth the following ideas.

1. Functions as data objects that can be inputs and outputs of other functions, called *higher-order* functions.

2. Recursion as the main control construct and induction as the means of proving properties of programs and data types. Indeed, we will see that *induction and recursion are two sides of the same concept*, an idea connecting proofs and programs in a mathematically strong way [1].

3. We will study *recursive data types* and see that these data types have natural *inductive properties*. We will examine the concept of *co-recursion* and look at co-recursive types such as *streams* and possibly *spreads* as well (trees that can grow indefinitely, also called co-trees).

4. The ML family of functional programming languages is especially appropriate for rigorous thinking about computational mathematics. We

will illustrate this by developing some aspects of the real numbers, $\mathbb{R}$ in OCaml. These will be *infinite precision computable real numbers*, and they have been used to present in a computational manner most of the calculus you learn in mathematics, science, and engineering [2, 3].

Two or three topics in this course are "cutting edge" in the sense that they are at the frontier of computing theory and type theory. So you will encounter a few ideas that are not typically seen until graduate school in computer science at other universities. These are topics that are especially interesting to the Cornell faculty in programming languages who are known for work in *language based security*.[2]

In particular, we will look briefly at how programs can be *formally specified in logic* and how systems called *proof assistants* can help us prove rigorously that programs meet their specifications. We will discuss a particular French proof assistant that is widely used for this purpose, called *Coq*, and its close relative the *Nuprl* proof assistant built and maintained at Cornell. These and other proof assistants (Agda, HOL, Isabelle HOL) have contributed to research in programming languages that are related to OCaml. Coq can generate OCaml code from proofs, and Nuprl can be thought of as a combined programming language and logic, *Nupr*ogramming *l*anguage.

The Coq proof assistant is being used to create a book, *Software Foundations* [15] which formalizes the semantics of programming languages using ideas from the textbook on programming language theory by Pierce [14] and the textbook by Harper [5]. All of the mathematical results in the *Software Foundations* book have been developed with the Coq proof assistant and are correct to the highest standards of mathematics yet achieved because they are mechanically checked by the proof assistant. It is not only that there are no "typos" in the proofs from this book, it is that there are no mistakes in reasoning, and the programs written are completely type correct and also meet their specifications.

**OCaml Theory of Computation and Types** Every programming language embodies a "mathematical theory of computation". OCaml relies on a *theory of types* to organize its theory of computation. This computation theory is grounded in sophisticated mathematical concepts originating in *Principia Mathematica* [19] and adapted to programming. For example, you

[2]The current PL faculty at Cornell are R. Constable, J. Foster, D. Kozen, A. Myers, R. Tate, and F. Schneider (mainly a faculty member in distributed systems).

might enjoy reading an extremely influential article by Tony Hoare [7, 12] on data types. After this course you will understand it well.

Our version of the course will stress these mathematical ideas a bit more than in the past because of the background and beliefs of the two faculty instructors. We believe that the mathematical ideas underlying OCaml have enduring theoretical and practical value and will become progressively more important in computer science and in computing practice. These ideas are especially important in an age when US cyber infrastructure is increasingly under attack.

This course adds to the functional programming and data structures core other important concepts from computer science theory, namely an understanding of performance, e.g. *asymptotic computational complexity* and an understanding of *program correctness*, how to define it and how to achieve it. We will study methods and tools for organizing large programs and computing systems. We also take up the issue of concurrency and asynchronous distributed computing, a key topic in the study of modern software systems.

**Course Topics** You can see the sweep of the course and how its main topics are approached from the section headings for our lectures and the accompanying recitations in this offering. They are:

1. Introduction to OCaml functional programming – 4 lectures, 4 recitations
2. Data types and structures – 6 lectures, 6 recitations
3. Verification and testing – 4 lectures, 3 recitations
4. Analysis of algorithms and data structures – 4 lectures, 4 recitations
5. Modularity and code libraries – 3 lectures, 3 recitations
6. Concurrency and distribution – 4 lectures, 4 recitations

These topics account for 25 lectures and there is room for a review lecture and another enrichment lecture. The content is covered in about 25 lectures and 24 recitations.

**Lecture Notes and Readings** Many of the lecture notes will be from previous offerings of CS3110, however several lectures including this one will be new material and will be posted on the web around the time of the lecture. Some new lecture notes will include the material from previous offerings, perhaps with additions or modifications.

# 4 OCaml Syntax

**The OCaml alphabet** The first step in defining any language precisely, including natural languages, programming languages, and formal logics, is to present its *syntax*. The syntax determines precisely what strings of characters are *programs* and what strings are *data*. The first step is to specify the *alphabet* of symbols used, the "letters of the alphabet of the programming language." Let $\Sigma_{OCaml}$ be this alphabet. In this section we use $\Sigma$ for short. We use exactly 94 symbols (tokens, characters) which are the 52 letters of the English alphabet, 26 lower case and 26 upper case, and 32 special symbols from the standard key board, and ten digits, 0 to 9. These are available on standard key boards.

In words the thirty two special symbols are these: exclamation point (!), at-sign (@), pound sign (♯), dollar sign ($), percent sign (asterisk (*), right parenthesis, left parenthesis, underscore, hyphen (-), plus (+), equal (=), right curly bracket, left curly bracket, right brace (]), left brace ([), vertical line (|), colon (:), semicolon (;), quote ("), apostrophe ('), less (¡), greater (¿), comma, period, question mark (?), tilde (∼), backslash, front slash (/), reverse apostrophe (‘).

Latex uses some of these characters to control the type setting, but the names are quite standard. Some have nicknames, such as "squiggle" for tilde. Hyphen is also a minus sign. The pound sign is sometimes called a "hash", and it is not the sign for the UK currency. Here is a use of square brackets, [...], and here is a use of curly brackets {...}.

The full set of OCaml symbols are from the ISO8859-1 character set with 128 standard characters and 127 others, many are English letters with diacritical marks to spell words in Western European languages, e.g ö, umlaut o. OCaml implementations typically support the standard 94 symbols plus 51 accented letters such as, ö.

Latex shows the need for many many more special symbols as does *unicode*. There a many hundreds of special characters that can be printed with

Latex and with unicode, and in the future such symbols will be included in the atomic symbols of programming languages. So we might have an alphabet $\Sigma$ with thousands of letters. Languages like Chinese could show us the limits of comprehension for such rich symbol sets.

**OCaml words and expressions** Finite strings of the basic symbols we will call *expressions* or *terms*. They are an analogue of words in English, even though many are nonsense words, like *abkajeky* in English. The set of words is denoted $\Sigma^+_{OCaml}$, all finite strings of symbols, even nonsense ones such as **1Ab–!. The space (character 0020) is not part of any word in the language nor is a line break or carriage return.

Unlike with natural languages, there is no dictionary of all known OCaml words as there seems to be for (almost) all English or French words. However, there is a dictionary of *reserved words* such as *fun, if, then, else, int, float, char, string*, and so forth. There is a largest reserve word (what is it?) but no"largest word" such as "supercalifragilisticexpialidocious" in an OCaml "dictionary", though memory requirements on machines place a practical limit on word length, and in any particular application program there is a list of names of important functions and data types. One can imagine that each project has a dictionary.

**OCaml programs and data** An OCaml program is simply an OCaml expression that reduces under the computation rules when applied to a value or given input. *Running a program is evaluating an expression.* A *value* is an OCaml expression that is *irreducible* under the computation rules. We will next look carefully at how to organize the explanation of programs and data. First a word about the scope of this task.

OCaml is a large industrial strength programming language meant to help people do serious work in science, education, business, government and so forth. Like all such languages *it is large, complex, and evolving.* We aim to study a subset that is good for teaching important ideas in computer science. *Thus there are many features of OCaml that we will not cover.* On the other hand, we will present a good framework for learning the entire language as it evolves from year to year – as all living languages do.

There is no official OCaml subset for education as has been the case in the past with large commercial programming languages, e.g. at the time when PL1 was a widely used language supported by IBM, there was a Cornell

subset called PLC that was widely taught in universities and made Cornell well known in programming languages.[3] The PLC work had an influence on Milner's thinking about ML, see the references in *Edinburgh LCF* [4], the first book on ML.

# 5 Mathematical Semantics for Programming Languages

A rigorous mathematical method has been developed for precisely defining how programs execute [18, 16, 8, 17]. The concepts are covered in most modern textbooks on programming languages [13, 20, 10, 14, 5]. We will use these ideas to give an account of OCaml semantics. Here is the first key idea of that semantics.

**Definition**: We divide the OCaml expressions into two classes, the *canonical expressions* and the *non-canonical expressions*. The canonical expressions are the *values* of the language. They are defined as expressions which are *irreducible under the computation rules*. This is a concept that you need to know for exams and discussions.

Sometimes we call these values *constants*. This is common terminology for *numerical values* of which OCaml has two types, the *integers* and the *floating point* numbers which are approximations of the infinitary real numbers of mathematics. It is not a word typically used for all of the constants of OCaml, some of which are functions and types.

**Exercise**: Give five examples of canonical OCaml expressions and five non-canonical ones not mentioned in this lecture.

## 5.1 expressions and values

**Simple values as constants** The integer constants are 0, 1, -1, 2, -2, .... These are constants in decimal notation. They are canonical values because no computation rules reduce them. There is a limit to their size on either 32 bit machines or 64 bit machines. OCaml supports both sizes. Thus these numbers are not like the mathematical integers whose value is unbounded and

[3]Many old languages such as COBOL and PL1 are still in use supporting large industrial operations. For mysterious reasons certain languages like C tend to become nearly "immortal". Others like FORTRAN continue to evolve and are immortal in that way. Java, $C^{++}$, and Lisp might be like that.

which thus form an *unbounded type*. OCaml does support an implementation of mathematical integers which in Lisp are called "Bignums."

We may discuss Bignums later, but we will not go into much detail on the limits of OCaml-integers and OCaml-floats. Later in the course we will show how to define *infinite precision* real numbers and thus model the type of mathematical reals $\mathbb{R}$ exactly.

The type *bool* is simpler having only two canonical values, the two Booleans, $true$ and $false$; simpler still is the *unit type* with one value, ().

**Structured values – tuples and records** Other canonical forms have structure. For example, $(1, 2)$ is the *ordered pair* of two integers. This is a value, and we call it a constant as well, although unlike the boolean *true* pairs have structure. OCaml also has n-tuples of values – here is a quadruple or four-tuple, $(1, 3, 5, 7)$. OCaml also has values called *records* which are like tuples, but the components are named as in $\{yr = 2020; mth = 1; day = 20\}$.

**Structured values – functions** One of the significant distinguishing features of OCaml is that functions are values. They can be supplied as inputs to other functions and produced as output results of computation. Functions have the syntactic form $fun\ x\ \rightarrow body(x)$, where $x$ is an identifier denoting the input value, and $body(x)$ is an OCaml expression that usually includes $x$ as a subterm, but need not, e.g., $fun\ x\ ->\ 0$ is the constant function with value integer 0. The *identity function* on any data type is $fun\ x\ ->\ x$.

These function expressions are *irreducible*, and thus are canonical expressions. When applied to a value, as in $(fun\ x\ ->\ x)0$ we create a reducible term. In this case it reduces to 0. We see that function values can have considerable internal structure. There is the operator name, *fun*, an abbreviation of the word function. The identifier $x$ is the *local name* of the input to the function, and $body(x)$ is its "program" or operation on the potential data $x$.

During computation after an input value $v$ is supplied, this value is substituted for the input variable $x$ resulting in the term $body(v)$. This expression can be canonical or non-canonical. A value is required to initiate the evaluation of a function, but the computation of $body(v)$ might not ever use the value, as in the case of a constant function such as $fun\ x\ \rightarrow\ 0$ or $fun\ x\ \rightarrow (fun\ y\ \rightarrow\ y)$.

In the original ML language, now called Classic ML, the function con-

stants have the form $\backslash\ x.body(x)$ which is close to the mathematical notation derived from *Principia Mathematica* and made popular by the American logician Alonzo Church who defined the *lambda calculus* where functions are denoted $\lambda\ x.body(x)$.

There are many notations for functions used in mathematics. In some textbooks we see functions written as in $sine(x)$ or $log(x)$ or even $x^2$. This notation is ambiguous because we might also use the same expression to denote "the value of the sine function applied to a variable $x$."

The programming languages Lisp and Scheme also allow functions as values. Lisp uses the key word *lambda* instead of *fun*. So $fun\ x\ \rightarrow x+1$ is written $(lambda(x)(x+1))$.

As mentioned above one of the other basic syntactic forms of OCaml is the *application* of a function to an argument. This is written as $fa$ where $f$ is a function expression and $a$ is another expression. The application operator is implicit in this notation whereas in some programming languages we see application written as $ap(f;a)$ where the operator is explicit.

## 5.2 evaluation and reduction rules

The OCaml run time system executes programs that have been compiled into assembly language. This is in a sense the *machine semantics* of OCaml evaluation, but it is too detailed to serve as a mathematical model of computation that we can reason about at a high level. The ML languages have a semantics at a higher level of *reduction rules*. These rules are used in textbooks such as *The Definition of Standard ML* [9].

Evaluation is defined using *reduction rules*. These rules tell us how to take a single step of computation. We use a computation system called *small step* reduction.

Here is an example of a very simple reduction rule. We first note that there are two primitive canonical functions, $fst$ and $snd$, that operate on ordered pairs, that is on expressions of the form $(e1, e2)$. They are (built-in) primitive operations.

We want a rule format to tell us that $fst(a,b)$ reduces in one step to $a$ and $snd(a,b)$ reduces in one step to $b$. The rules tell us that we can think of $fst$ as picking out the first element of an ordered pair while $snd$ picks out the second.

**Rule-fst** $fst(a, b) \downarrow a$.
**Rule-snd** $snd(a, b) \downarrow b$.

Here are rules for the Boolean operators.

**Rule Boolean-and** $true \;\&\&\; false \downarrow false$
**Rule Boolean-or** $true \;||\; false \downarrow true$

The general rule for the Boolean operators should take arbitrary expressions, say $exp1$ and $exp2$ and reveal how those values are computed before the principal Boolean operator is computed. To express such rules, we need to state hypotheses about how these expressions are evaluated. Here is the way OCaml performs the reduction.

**Rule Boolean-or-1** $exp1 \downarrow true \vdash exp1 \;||\; exp2 \downarrow true$

**Rule Boolean-or-2** $exp1 \downarrow false, exp2 \downarrow true \vdash exp1 \;||\; exp2 \downarrow true$

**Rule Boolean-or-3** $exp1 \downarrow false, exp2 \downarrow false \vdash exp1 \;||\; exp2 \downarrow false$

These Boolean values are used to evaluate conditional expressions.
**Rule Conditional-true**

$$bexp \downarrow true, exp_1 \downarrow v_1 \vdash (if\ bexp\ then\ exp_1\ else\ exp_2) \downarrow v_1.$$

**Exercise**: Write the other rule for evaluating the conditional expression.

Here is the rule for evaluating function application.
**Function Application**

$$exp_2 \downarrow v_2, exp_1 \downarrow fun\ x\ ->\ body(x), body(v_2/x) \downarrow v_3 \vdash (exp_1\ exp_2) \downarrow v_3.$$

Notice the *order of evaluation*, we evaluate the argument, $exp_2$ first. If that expression has a value, then we evaluate $exp_1$ and if that evaluates to a function $fun\ x\ ->\ body(x)$, then we substitute the value $v_2$ for the variable $x$ in *body* and evaluate that expression. This is called eager evaluation or call by value reduction because we eagerly look for the input to the function, even before we really know that $exp_1$ evaluates to a function.

There is another order of evaluation in programming languages where we first evaluate $exp_1$ to make sure it is a function, then we substitute $exp_2$ in

for the variable in the body and only evaluate it if that is required by the body. For example, if the body is just $fun\ x\ ->\ x$ then we do not have to evaluate the input first since the body does not "need it yet." This is called *lazy evaluation*. OCaml supports this style of evaluation as well, but we will discuss that later.

These simple rules might seem tedious, but they are the basis for a precise semantics of the language that both people and machines can use to understand programs. By writing down all these rules formally, we create a *shared knowledge base with proof assistants*. It would be very good if OCaml had a complete formal definition of this kind to which we had access. I don't know of one. We could probably crowdsource its creation if we had the ambition and the time.

**divergence** In all of the evaluation rules for OCaml it is entirely possible that the expression we try to evaluate will *diverge*, meaning "fail to terminate". That is, the computation runs on forever until memory is exhausted or until you get tired of waiting and stop the evaluation process which is "in a loop." We can write very simple programs that will loop forever without using up memory.

**exceptions** Expressions might also just "get stuck" as when we try to apply a number to another number, as in 5 7 or take the first element of a function value, e.g. $fstfun\ x\ ->\ (x, x)$. Such attempts to evaluate an expression do not make sense and would get stuck if we tried to evaluate them.

We will see that the type system helps us avoid expressions whose attempted evaluation would get stuck, but we cannot avoid all such situations, and later we will discuss computations that cause exceptions.

**Exercise**: Write a *diverging computation*, a short non-canonical expression that diverges. This will be discussed in recitation where you will try to find the simplest such expression in OCaml. More subtle question, can there be such an expression that does not consume an unbounded amount of memory?

# References


[1] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[2] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[3] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

[4] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[5] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, 2013.

[6] Robert Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report TR ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[7] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[8] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[9] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[10] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[11] John C. Mitchell and Robert Harper. The essence of ML. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 28–46. ACM, 1988.

[12] E. W. Dijkstra O. J. Dahl and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[13] L. C. Paulson. *Standard ML for the Working Programmer*. Cambridge University Press, 1991.

[14] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[15] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. Electronic, 2011.

[16] Gordon D. Plotkin. LCF considered as a programming language. *Journal of Theoretical Computer Science*, 5:223–255, 1977.

[17] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[18] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[19] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.

[20] G. Winskel. *Formal Semantics of Programming Languages*. MIT Press, Cambridge, 1993.