# MPRI course 2-4
# “Programmation fonctionnelle et systèmes de types”
# Programming project

François Pottier

2017–2018

## 1 Summary

The purpose of this programming project is to implement a tiny compiler from **Lambda**, an untyped $\lambda$-calculus, down to **C**.

## 2 Required software

To use the sources that we provide, you need OCaml and Menhir. Any reasonably recent version should do. You also need the OCaml packages `process`, `pprint`, and `ppx_deriving`. If you have installed OCaml via `opam`, issue the following command:

```
opam install menhir process pprint ppx_deriving
```

## 3 Overview of the provided sources

Many components of the compiler are provided, including: definitions of the syntax of terms; a lexer and parser; a pretty-printer for C programs; some code for dealing with names and binders.

In the `src/` directory, you will find the following files:

**alphalib/Atom.{ml, mli}** An atom is an internal object used to represent a name. It is a pair of a unique integer identity and a (not necessarily unique) string. The function `Atom.fresh` creates a fresh atom.

**Error.{ml, mli}** This module deals with places (positions in the source code) and error messages. It is used to report syntax errors and unbound variables.

**Parser.mly, Lexer.mll** Together, the lexer and parser define the concrete syntax of the surface language.

**RawLambda.ml** This is the surface language. It is an untyped $\lambda$-calculus, extended with possibly recursive `let` bindings, primitive integer constants, four primitive integer arithmetic operations, and a primitive integer display operation `print`. We do not explicitly define the semantics of this calculus: it is standard. Let us just note the following: (1) the semantics is call-by-value; (2) the right-hand side of a `let rec` construct must be a $\lambda$-abstraction; (3) the `print` operation prints a primitive integer value (followed with a newline character) on the standard output channel and returns this value; (4) the final value of a program is not displayed; it is dropped.

**Cook.{ml, mli}** The translation of **RawLambda** to **Lambda**.

**Lambda.ml** This is a slightly simplified version of the surface language. It is an untyped $\lambda$-calculus, extended with possibly recursive $\lambda$-abstractions, nonrecursive `let` bindings, and the primitive integer constants and operations. Some well-formedness properties (such as the fact that every variable is properly bound) are checked during the translation of **RawLambda** to **Lambda**. From this point on, no compilation errors are expected: every **Lambda** program must be translated to a **C** program. Because **Lambda** is untyped, some **Lambda** programs go wrong; they can be translated to **C** programs that crash. A well-typed **Lambda** program must be translated to a **C** program that runs safely.

**CPS.{ml, mli}** Through a CPS transformation, the surface language **Lambda** is translated down to the intermediate language **Tail**. **It is up to you to implement this transformation.**

**Tail.ml** This intermediate language describes the result of the CPS transformation. It is a lambda-calculus where the ordering of computations is explicit and where every function call is a tail call. Like the surface calculus, it allows $\lambda$-abstractions that have free variables.

**Defun.{ml, mli}** Through defunctionalization, the intermediate language **Tail** is translated down to the next intermediate language, **Top**. **It is up to you to implement this transformation.**

**Top.ml** This intermediate language describes the result of defunctionalization. It retains the key features of the previous calculus, **Tail**, in that the ordering of computations is explicit and every function call is a tail call. Furthermore, $\lambda$-abstractions disappear. A memory block `Con` now contains an integer tag followed with a number of fields, which hold values. A `switch` construct appears, which allows testing the tag of a memory block. A number of (closed, mutually recursive) functions can be defined at the top level.

**Finish.{ml, mli}** This function implements a translation of the intermediate language **Top** down to **C**. This transformation is mostly a matter of choosing appropriate **C** constructs to reflect the concepts of the language **Top**.

**prologue.h** This **C** header file defines a small number of types and macros which are used in the generated code. You may find it interesting.

**kremlin/C.ml** This is an abstract syntax tree for a subset of the **C** language. It is borrowed from Jonathan Protzenko's Kremlin, a tool which translates a subset of **F*** down to **C**.

**Main.ml** This driver interprets the command line and invokes the above modules as required.

**Makefile, _tags** Build instructions. Issue the command "`make`" in order to generate the executable. You may need to first run "`opam install menhir ppx_deriving pprint process`".

**joujou** The executable file for the program. Type "`./joujou` *filename*" to process the program stored in *filename*. Use the option "`--debug`" to display every intermediate abstract syntax tree.

**Testing** In the `tests/` directory are small programs written in the source language, **Lambda**, which you can give as arguments to `joujou`.

In order to test your implementation, run "`make test`". The script submits the files `tests/*.lambda` to your compiler, then compiles and runs the resulting C programs, and checks that the outcomes are appropriate.

The file `tests/loop/loop.lambda` is not part of the test suite, but is included for fun. It is a program that prints 0, 1, 2, ... and never terminates. If your compiler produces tail-recursive code, and if your **C** compiler is able to recognize and optimize tail calls, then the compiled **C** program should actually run forever. To try it out, just move the file `tests/loop/loop.lambda` one level up into `tests/`.

**Advice** We strongly recommend that you regularly take checkpoints (that is, snapshots of your work) so that you can later easily roll back to a previous consistent state in case you run into an unforeseen problem. Using a versioning tool such as `git` is highly recommended.

# 4 Task description

**Task 1a** In the files **CPS.{ml, mli}**, implement the translation of the surface language **Lambda** down to the intermediate language **Tail**. This is a CPS transformation.

**Task 1b** In the files **Defun.{ml, mli}**, implement the translation of the intermediate language **Tail** down to the next intermediate language, **Top**. This is a defunctionalization.

**Test** At this point, "`make test`" should work. Feel free to add more test files in the subdirectory `tests/`.

**Task 2** Extend the surface language with a new primitive construct, "`ifzero` $e_0$ `then` $e_1$ `else` $e_2$", which tests whether a primitive integer is zero or nonzero, and takes an appropriate branch. This requires **extending all compiler passes**, beginning with the lexer and parser, all the way down. **Create more test files** in `tests/` that exploit the new construct, and make sure that "`make test`" still works.

**Optional tasks** If you wish to go further and receive extra credit, there are a number of things that you might do. Here are some suggestions. This list is not sorted and not limiting. Not all suggestions are easy! Think before attacking an ambitious extension.

- In the intermediate language **Top**, eliminate variable-variable bindings "`let` $x = y$ `in` $e$", so as to produce cleaner **C** code in the end.
- Add mutually recursive functions to the source language.
- Add exceptions to the source language.
- Add a delimited control operator to the source language.
- Add a form of algebraic data structures to the source language.
- Compile functions of more than one argument in a more efficient way.
- Perform lightweight defunctionalization: if a function `g` refers to a toplevel function `f`, then the closure for `g` need not contain a slot for `f`.
- Gracefully detect runtime errors. An (ill-typed) program that goes wrong should not crash; it should display a nice error message.
- Write a static type-checker or type inference system.
- Plug in a conservative garbage collector.

In each case, please write **a textual explanation** of what you did, how you did it, and where to look for it in your code. Also, propose **test files** that illustrate what you did.

## 5 Evaluation

Assignments will be evaluated by a combination of:

- **Testing**. Your compiler will be tested with the input programs that we provide (make sure that "`make test`" succeeds!) and with additional input programs.
- **Reading**. We will browse through your source code and evaluate its correctness and elegance.

## 6 What to turn in

When you are done, please e-mail to François Pottier and Pierre-Évariste Dagand and Yann Régis-Gianas and Didier Rémy a `.tar.gz` archive containing:

- All your source files.
- Additional test files written in the small programming language, if you wrote any.
- If you implemented "extra credit" features, a `README` file (written in French or English) describing these additional features, how you implemented them, and where we should look in the source code to see how they are implemented.

## 7 Deadline

Please turn in your assignment on or before **Friday, February 16, 2018**.

## References

[1] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[2] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.