MPRI 2.4
Programmation fonctionnelle et systèmes de types
Programming project

Didier Rémy

2021–2022

Abstract

The purpose of this programming project is to implement a type-checker for System F^ω.

1 Presentation

The purpose of this programming project is to implement a type-checker for System F^ω. Our version of F^ω includes let-bindings, tuples, structural records, and primitive existential types. We also allow type definitions and toplevel unpacking of existential types.

Concrete syntax Figures 1 and 2 give the concrete syntax. (This is actually just an approximation, as, for instance, this syntax does not describe priorities of constructs and omits some details. The full definition appears in the file `src/parser.mly`.

Functions are n-ary, taking a sequence of parameters, which include type abstractions and value abstractions. The syntax uses patterns p for value abstractions. Simple patterns are just variables x or annotated variables $x : \tau$. Complex patterns are there for possible extensions and typechecking of expressions may ignore them in all of the required tasks (by raising the typing error `NotImplemented` with the argument `"Complex pattern"`).

Recursive definitions Let bindings of recursive definitions must be annotated. This is not wired in the syntax, because some of the extensions may late relax this condition. Tyepecking should then raise the exception `Annotation` with the name of the variable that is missing a type annotation for argument (as with program `fail_fact_T1.fw`).

Packages are built as $\langle \tau, M \ \mathsf{as}\ \tau' \rangle$ where τ is the witness type and τ' must be (equivalent to) an existential type. Unpacking uses the let-binding syntax $\mathsf{let}\ \langle \alpha, x \rangle = M\ \mathsf{in}\ N$.

Toplevel declarations include definitions $\mathsf{let}\ p = M$ as usual, but also abstract definitions $\mathsf{let}\ \langle \alpha, x \rangle = M$, which opens the package M in the rest of the program, and type definition $\mathsf{type}\ \alpha = \tau$ which introduces α as an abbreviation for τ in the rest of the program. Their typing is detailed below.

Identifiers include type variables α, value variables x, and record labels ℓ. They are lexically indistinguishable: they start with lower or uppercase letters (with "_" treated as a letter) followed by a sequence compose of letters, digits or the character "'". See `lexer.mll` for details.

κ	::=	$\mathsf{Type} \mid \kappa \to \kappa$	**Kinds**
τ	::=		**Types**
	$\mid$	α	
	$\mid$	$\tau \to \tau$	Arrow
	$\mid$	$\mathsf{lam}\ \alpha :: \kappa \cdot \tau$	Function
	$\mid$	$\mathsf{all}\ \alpha :: \kappa \cdot \tau$	Universal
	$\mid$	$\mathsf{exi}\ \alpha :: \kappa \cdot \tau$	Existential
	$\mid$	$\tau \times \cdots \tau$	Tuple
	$\mid$	$\{\ell : \tau; \cdots \ell : \tau\}$	Record
	$\mid$	(τ)	
p	::=	$x \mid (x : \tau)$	**Patterns**
	$\mid$	$(p : \tau) \mid (p * \cdots p) \mid \{\ell : p; \cdots \ell : p\}$	Complex pattern
b	::=	$[\alpha] \mid [\alpha :: \kappa] \mid p$	bindings
M	::=		**Expressions**
	$\mid$	x	
	$\mid$	$M\ M$	Application
	$\mid$	$M\ [\tau]$	Type application
	$\mid$	$\mathsf{fun}\ b\ \cdots b \Rightarrow M$	Function
	$\mid$	$\mathsf{let}\ \ \mathsf{rec}^?\ p\ b\ \cdots b = M\ \mathsf{in}\ M$	Let binding
	$\mid$	$\langle \tau, x\ \mathsf{as}\ \tau \rangle$	Package
	$\mid$	$\mathsf{let}\ \langle \tau, x \rangle = M\ \mathsf{in}\ M$	Package opening
	$\mid$	$M, \cdots M \mid M.n$	Tuple and tuple access
	$\mid$	$\{\ell : M; \cdots \ell : M\} \mid M.\ell$	Record and record access
	$\mid$	$M : \tau$	Type annotation
	$\mid$	(M)	
D	::=		**Declaration**
	$\mid$	$\mathsf{let}\ \ \mathsf{rec}^?\ p\ b\ \cdots b = M$	Concrete definition
	$\mid$	$\mathsf{let}\ \langle \alpha, x \rangle = M$	Abstract definition
	$\mid$	$\mathsf{type}\ \alpha \mid \mathsf{type}\ \alpha :: \kappa$	Abstract type definition
	$\mid$	$\mathsf{type}\ \alpha = \tau$	Concrete type definition
P	::=	$I\ \cdots I\ D\ \cdots D$	Programs
I	::=	$\#\mathsf{include}\ \texttt{"}filename\texttt{"} \mid \#\mathsf{option}\ \texttt{"fail"}$	Directives

Figure 1: Concrete syntax

$\mathsf{fun}\ \alpha :: \kappa \Rightarrow M$	$\stackrel{\mathsf{def}}{=}$	$\mathsf{fun}\ [\alpha :: \kappa] \Rightarrow M$
$\mathsf{fun}\ x : \tau \Rightarrow M$	$\stackrel{\mathsf{def}}{=}$	$\mathsf{fun}\ (x : \tau) \Rightarrow M$
$\mathsf{let}\ p\ b\ \cdots b = M$	$\stackrel{\mathsf{def}}{=}$	$\mathsf{let}\ p = \mathsf{fun}\ b\ \cdots b \Rightarrow M$
$\mathsf{let}\ p\ b\ \cdots b : \tau = M$	$\stackrel{\mathsf{def}}{=}$	$\mathsf{let}\ p\ b\ \cdots b = M : \tau$
$\{.. \ell\,;\ ..\}$	$\stackrel{\mathsf{def}}{=}$	$\{.. \ell = x_\ell\,;\ ..\}$

Figure 2: Syntactic sugar

Syntactic sugar. We also allow for some shortcuts, summarized on Figure 2:

- A function with a single annotated argument may omit brackets or parentheses.
- A function definition $\mathsf{let}\ p = \mathsf{fun}\ b\ \cdots b \Rightarrow M$ may be written $\mathsf{let}\ p\ b\ \cdots b = M$. Besides, if M is a type annotation $N : \tau$, it can be written $\mathsf{let}\ p \ldots : \tau = N$ instead of be written $\mathsf{let}\ p \ldots = N : \tau$.
- In record expressions, a field ℓ without a definition M stands for $\ell = x_\ell$ where ℓ and x_ℓ are actually the same string (as is the case in OCaml).
- A type variable with a Type kind annotation $\alpha :: \mathsf{Type}$ may be abbreviated as α (except in $\mathsf{fun}\ \alpha :: \mathsf{Type} \Rightarrow M$, where it remains mandatory).

Abstract syntax The abstract syntax of F^ω appears in the file `src/syntax.mli` with some inlined comments.

Type-checking expressions. Typing rules for core expressions and existential types are as in the course. Typing rules for tuple and records are the following:

$$\frac{(\Gamma \vdash M_i : \tau_i)^{i \in 1..n}}{\Gamma \vdash \{\ell_1 = M_1; .. \ell_n = M_n\} : \{\ell_1 : \tau_1; .. \ell_n : \tau_n\}} \qquad \frac{(\Gamma \vdash M_i : \tau_i)^{i \in 1..n}}{\Gamma \vdash (M_1, .. M_n) : \tau_1 \times .. \tau_n}$$

$$\frac{\Gamma \vdash M : \{\ell_1 : \tau_1; .. \ell_n : \tau_n\} \qquad i \in 1..n}{\Gamma \vdash M.\ell_i : \tau_i} \qquad \frac{\Gamma \vdash M : \tau_1 \times .. \tau_n \qquad i \in 1..n}{\Gamma \vdash M.i : \tau_i}$$

Notice that records are structural and thus differ from named records presented in the course.

Typechecking declarations A program P is a list of definitions $D_1 .. D_n$—or rather a telescope of definitions, since definition D_i may be used in definition D_{i+k}. Instead of typing declarations one by one as independent phrases, we view a program P as a "structure" to which we assign a "signature" Δ, which is a small piece of typing context summarizing D that will be appended to Γ when P is included and used to type another program P'. That is, programs are typed with judgments of the form $\Gamma \vdash P : \Delta$ and obtained by folding typing of each definition from left to right:

$$\textsc{Let}\ \frac{\Gamma \vdash M : \tau \qquad \Gamma, x : \tau \vdash P : \Delta}{\Gamma \vdash \mathsf{let}\ x = M\ P : (x : \tau, \Delta)} \qquad \textsc{Type}\ \frac{\Gamma, \alpha = \tau \vdash P : \Delta}{\Gamma \vdash \mathsf{type}\ \alpha = \tau\ P : (\alpha = \tau, \Delta)}$$

$$\textsc{Open}\ \frac{\Gamma \vdash M : \mathsf{exi}\ \alpha \cdot \tau \qquad \Gamma, \alpha, x : \tau \vdash P : \Delta}{\Gamma \vdash \mathsf{let}\ \langle \alpha, x \rangle = M\ P : (\alpha, x : \tau, \Delta)} \qquad \textsc{AbsType}\ \frac{\Gamma, \alpha :: \kappa \vdash P : \Delta}{\Gamma \vdash \mathsf{type}\ \alpha :: \kappa\ P : (\alpha :: \kappa, \Delta)}$$

A concrete definition of the form $\mathsf{let}\ x = M$ extends the environment with a new binding x of type τ—the type of M. (Definition may also be recursive, and x may be a type annotation—or a more complex pattern if implemented; we let the reader adapt the typing rule.)

A type definition $\mathsf{type}\ \alpha = \tau$ introduces a new type name α that stands for (and is convertible to) τ. It behaves as a type abbreviation in OCaml. For example, the OCaml type declaration

```
type 'a pair = 'a * 'a
```

would be encoded as $\mathsf{type\ \ pair} = \mathsf{lam}\ \alpha \cdot \alpha \times \alpha$. Then $\mathsf{pair}\,\tau$ is equivalent to $(\mathsf{lam}\ \alpha \cdot \alpha \times \alpha)\ \tau$, by expansion of the type definition (a δ-reduction at the type level) which is itself equivalent to $\tau \times \tau$ by β-reduction.

Abstract definitions are similar to unpacking existential types but with a scope that extends to the remaining of the program. An example is `church_sum.fw`.

For convenience, we also allow abstract type definitions $\mathsf{type}\ \alpha :: \kappa$, which just introduces a new abstract type α of kind κ in the context Γ. These may be used to model generative type definitions as in OCaml or to emulate new primitive types as in the example `prim_sum.fw`. As a hint, an abstract type definition can also be seen as syntactic sugar for $\mathsf{let}\ \langle \alpha, \_ \rangle = V_0$ where $\_$ is a special variable inaccessible to the user (hence no pushed in the environment) and V_0 is any value of type $\mathsf{exi}\ \alpha \cdot \alpha$ (take for instance $\langle \tau, V\ \mathsf{as\ exi}\ \alpha \cdot \alpha \rangle$ given any value V of type τ).

Include directives A program may start with include directives. Each directive I stands for some program P_I stored in another file. Program with includes are typed as follows:

$$\frac{\Gamma \vdash_\flat P_I : \Delta_I \qquad \Gamma, \Delta_I \vdash P : \Delta}{\Gamma \vdash I\ P : \Delta} \qquad\qquad \frac{\Gamma \vdash_\flat P_I : \Delta_I \qquad \Gamma, \Delta_I \vdash_\flat P : \Delta}{\Gamma \vdash_\flat I\ P : \Delta_I\ \Delta}$$

That is, an included program P_I is typed prior to P and so recursively (P_i may itself contain include directives). We however type P_I in include mode $\vdash_\flat$ so that both its recursive includes and its own definitions contribute to increase the typing environment for P. By contrast, in normal mode, include environments are only used for actual definitions in P and only those contribute to the signature of P that will be returned and printed.

Kind checking Since F^ω extends System F with type functions, types must first be checked for well-kinding, prior to—or rather interleaved with—type checking. This is implemented by the function `Typing.type_typ`, which simultaneously translate source types into internal types and compute its kind.

Convertibility Since F^ω extends System F with type functions, types must be compared up to β-reduction. This is mandatory.

Earler computation. Once types have been checked for well-sortedness, reduction will terminate. Thus, one possibility is to expand type definitions and compute normal forms eagerly. This is the simplest and the way to start, as then convertibility is just equality of normal forms, i.e., equality up to α-conversion. Hence, this is only what is required until task 4.

Lazy computation. However, earger computation may perform unnecessary computations and, worse, duplicate types and return large, often unreadable type expressions (in return types or output messages). Therefore, it is much better (and recommended) to compute types lazily, unfolding type definitions and performing type reduction only on demand, that is just reducing types to head normal forms when checking for convertibility, and keep definitions as long as possible.

The code is not significant longer, but must trickier. Hence, this is only part of task 4. There is no optimal solution in the unfolding strategy, so your output types may differ from the reference and still be fine. Notice that you may still skip this task or have an imperfect implementation to do some of the extensions.

In principle, one could separate computation to head-normal forms and lazy expansion of definitions, and you may implement one and not the other, but they two go well together.

An simple example to test the expansion of definitions is `lazy_keep_def.fw`. An special test example `reduce_expensive.FW` just a few miliseconds in lazy mode but may take up to a few minutes in eager mode. It is not run by default, but you may run it by typing `make timing` in the `test/inputs` directory.

Be aware that since functions may not be injective, checking for free type variables may also require expansion and reduction in some cases: for example α looks like a few variable in $(\mathsf{lam}\ \beta \cdot \tau)\ \alpha$ while it is not when β is not free in τ.

Representation of binders Since β-reduction requires some renaming to avoid name clashes, we need a suitable convention for the representation of binders. The implementation skeleton represents free and bound type variables by names, maintaining the invariant that variables may never shadow one another.

Of course, we do not wish to impose this convention on the user, who may use (and sometimes play with) shadowing. Hence, we translate between user names and internal names, i.e., between source types (of type `styp`) and internal types (of type `ctyp`).

The internal name of a source variable α is a pair (α, i) composed of the (original) source name α and an integer suffix i, so as to avoid shadowing. The question is how to choose the suffix i ?

- ***(i1) Global counter.*** The easiest solution is to use a global counter and generate a different i for each new variable α. This is correct—and a good chose for internal use.

- ***(i2) Smallest name-dependent suffixes.*** However, types printing to the user will have large suffixes and be hard to read. In fact, we make choose the same suffix for variables of different names. This can be easily implemted by choosing suffixes independently for each variable and in increasing order. Hence, the lastest suffix used is that of the last internal variable of the same source name.

- ***(i3) Allow printing types to be reread.*** The previous solutions are only correct as long as variables are compared as pairs of (α, i). However, if we wish to print renamed variables so that they can be read back, then source variables should be allowed to end with digits, which may create a capture between, e.g., the internal variables (`a1`, 2) and (`a`, 12) which although different internally will be both printed as `a12`—and thus misleadingly identified when read back.

  To fix that, we may split names α as $\overleftarrow{\alpha} \cdot \overrightarrow{\alpha}$ where $\overrightarrow{\alpha}$ is the longest suffix string of α that represents an integer and $\overleftarrow{\alpha}$, called the basename of α, is the prefix of α such that $\overleftarrow{\alpha} +\!\!+ \overrightarrow{\alpha}$ is equal α (where $+\!\!+$ is string concatenation). Then, we choose for i the smallest integer so that there is no other internal variable (β, j) in scope for which both $\overleftarrow{\alpha} = \overleftarrow{\beta}$ and $\overrightarrow{\alpha} +\!\!+ i = \overrightarrow{\beta} +\!\!+ j$ (here i and j mean their string representations where 0 is exceptionally represented as the empty string).

  Implementing this is not mandatory. It is independent of other features and the code is quite localized, but it requires to keep an additional map from base names the set of of all stings of the form $\overrightarrow{\beta} +\!\!+ i$ in scope for that base name.

  This solution may be tested with examples `shadowing_core_X1.fw` and `shadowing_typedef_X1.fw`. Other program examples should not exhibit the form of shadowing solved by (i3), hence they should produce identical names.

This choice is may implemented by the function `Typing.fresh_id_fo`, which comes with a default implementation for solution (i1).

The implementation of (i2) is mandatory, as otherwise names will be dependent of the order in which computations are done. Otherwise, you won't be able to automatically test your programs aginsts the given testbase.

Minimizing variable suffixes Types are printed to the user without shadowing, hence using internal names. To keep types readable, we maintain may smaller integer suffixes by renaming types after typechecking of expressions: indeed, typechecking may introduce many auxiliary variables (for internal types and for renaming during type computations, which often do not appear in resulting types). We therefore rename (bound) type variablesk of inferred types τ before they extend the environment Γ with $x : \tau$ by reducing their suffixes to the smallest value not present in Γ. Minimization is implemented by the function `Typing.minimize_typ` with the identity for default implementation.

The implementation of minimization is mandatory, as otherwise names will be dependent of the amount and order of computations and not on the end result. Otherwise, you won't be able to automatically test your programs aginsts the given testbase.

Tasks The sources that we provide are described next. Your tasks will be to complete files `src/type.ml` and `src/typing.ml` to have a fully working typechecker for our language. You may proceed in incremental steps:

- ***T0.*** Implement allocation (i1) for internal type variables and minimization.
- ***T1.*** Have a working typechecker for System F: this should cover all core expressions except type functions and existential types, and only consider concrete definitions for declarations.
- ***T2.*** Complete the typechecker with existential types, abstract definitions. Still without type functions nor type definitions.
- ***T3.*** Add type functions and type definitions, but expand and reduce them eagerly. (Do not worry about efficiency, such as sharing type computations.)
- ***T4.*** Expand type definition and reduce type functions them on demand. (You still need not share computations.)

The implementation of task 0 is easy, and independent of other tasks and can be implemented at any time. However, you won't be able to automatically test your programs aginsts the given testbase until this is implemented. Task 3 and 4 are more difficult but not very long. Task 4 is recommanded but not required for most of the extensions below.

Possible extensions There are many directions in which you may extend the prototype.

- ***X1. Allow reparsing of printed types.*** Implement solution (i3) for the allocation of integer suffixes so that source type variable may ending with integers.
- ***X2. Add primitive variant types.*** These are structural much as (and dual of) record types. A variant type is a type $[A_1 : \tau_1; .. A_n : \tau_n]$ that represent a value that has been built with one of the constructor A_i and a value of type τ_i. Variants are destructed with a match construct $\mathsf{match}\ M\ \mathsf{with}\ A_1\ x_1 \Rightarrow \tau_1 \mid .. A_n\ x_n \Rightarrow \tau_n$. Notice that parameters x_i need not be annotated, since their type is known from the type of M.
- ***X3. Avoid repeating type annotations in let rec.*** Allow to omit the annotation on a recursive definition of a function whose body is an annotation. For example, the program `let_rec_annot_X3.fw` should be accepted.

X4. Omit some type annotations. In $\mathsf{fun}\ x \Rightarrow M$ when the type of the argument x need not be annotated because the type of the function is known from context, typically when the function is in argument position or when there is a type annotation above it.

X5. Omit some type arguments. In a mixed application $M\ [\tau_1]\ ..\ [\tau_n]\ N_1\ ..\ N_k$, the type of some parameter τ_i may sometimes be inferred from the type of the arguments. You may let the user write `_` instead of $[\tau_i]$ in such cases. Remark: you should consider a maximal multi-application and never attempt to use the type of N_i to help typing N_j.

(The design space is large with increasing difficulty. You may just do some easy cases.)

X6. Improve reduction. You may try sharing some of the computations during convertibility. The design space is large and there is no optimal strategy. Just do some reasonable effort to avoid obvious recomputations.

X7. Alternate representations of binders. Use De Bruijn indices or a locally nameless internal representation of binders.

X8. Implement typechecking for complex pattern matching. In a first step, you may require that type variables of function are always annotated on the leaves. However, you may relax this and just require that the pattern contains an annotation so that the types of parameters of functions are always known.

X9. Allow kind abstraction. That is, introduce kind variables and the ability to abstract over kinds. For instance, this would allow to give a more polymorphic version `map_pair.fw` or define church existential types for arbitrary kinds rather than just a fixed kind.

X10. Propose your own.

2 Overview of the sources and tasks

The directory `src/` contains the following source files:

`util.ml[i]` This module defines a few missing library functions.

`syntax.ml[i]` This mainly defines the abstract syntax of types and expressions and a few helper functions. The interface file has a lot of comments. Source and internal types only differ by their representation of type variables. This allows to share their definition by abstracting over the representation of type variables.

`locations.ml,lexer.mll,parser.mly` These files define a lexer and parser for programs. The abstract syntax is instrumented with locations by helper functions defined in module `Locations` so that typing (and syntax) errors can be reported with precision.

`print.ml[i]` This module provides printers for types, expressions and programs. Printers for types are abstracted over printing of type variables.

This not only allows to share the printing of source and internal types, but also allows to change the printing of internal variables to take as reference the environment when printing rather that at the beginning of typechecking.

`type.ml[i]` This module defines operations on types

It is up to you to complete it. All functions have default trivial but incorrect or incomplete definitions, which should be fixed. Some functions may keep their default implementation for the first tasks.

`typing.ml[i]` **It is up to you to complete it.** All functions have default trivial but incorrect or incomplete definitions, which should be fixed. Some functions may keep their default implementation for the first tasks.

`main.ml` This file defines the main program. Typing "`make`" in the root directory compiles it and produces the executable program `_build/default/src/main.exe`. This program may be called directly or via the dune command:

```
dune exec src/main.exe --
```

It expects to receive on its command line the name of one or more source files, which will all be typechecked in the same initial environment, i.e., definitions of the first file won't be visible when typechecking the second file.

An example of a source program is `test/inputs/hello.fw`. It may be typechecked with one of the commands

```
dune exec src/main.exe -- hello.fw
```

This parses the source file, type-checks it and prints the inferred signature on `stdout`. Error messages are printed on `stderr` by default. The return code is 0 in case of success and 1 is case of errors. However, the command line option `--fail` or the directive `option "fail"` tells that the typechecking should fail: typing errors are then printed on stdour and return the 0 exit code while typing success will return a non-zero exit code.

To use eager mode for convertibily and normliation, say:

```
dune exec src/main.exe -- --eager hello.fw
```

Run the command

```
dune exec src/main.exe -- --help
```

to see other command line options.

Typechecking may also be called via the command `make`:

```
make lazy src=test/inputs/hello.fw
```

or, for files in `test/inputs`, you make just say:

```
make lazy program=hello.fw
```

For eager mode, replace `lazy` by `eager`.

You may also run test directly from the directory `test/inputs` by running one of the commands:

```
make hellow.fwi
make hellow.fwe
```

to perform typecheckinig with lazy or eager version (in this order). Look at `Makefile` in this directory to see other options.

Testing The directory `test/inputs` contains a number of small programs written in F^ω. These programs can be given as arguments to the executable program `_build/default/src/Main.exe`, together with the options `--lazy`, `--eager`, or both.

In order to run the entire test suite, run "`make test`" in the root directory. There are separate tests for the interpreter and for the type-checker. For each source file, such as `test/inputs/hello.fw`, the type-checker is separately invoked. During each of these tests, the standard output and error channel are recorded. This yields four files, named `hello.{lazy,eager}.{out,err}` and stored in the directory `_build/default/test/inputs`. The command "`make test`" compares the files `hello.`*`mode`*`.out` against the expected-output files `hello.`*`mode`*`.exp` and `hello.`*`mode`*`.exp` which are stored in the directory `test/inputs`. The infix *mode* may take the two values `lazy` and `eager` which means that the program was run with the option `--lazy` or `--eager`.

If there are differences, they are highlighted in color, making failures easy to spot.

Beware! The difference is only textual so far, hence there may be semantically correct differences due to the choice of variable suffixes or the way you repported type errors in case of failure. There may also be some differences in the `lazy` mode, dut to the order of definition expansions, for which this is no optimal strategy.

The files `hello.`*`mode`*`.err` are not compared against an expected-output file, but programs with the directive `"fail"` will be treated as successful if they do fail and typing errors will be printing in `stdout` instead of `stderr`.

A list of all tests can be found in the file `test/inputs/dune.auto`, which is generated by the OCaml script `test/Test.ml`. After adding a new source file in the directory `test/inputs`, or after modifying `test/Test.ml`, run "`make depend`" in the root directory to regenerate `test/inputs/dune.auto`.

The command "`make promote`" overwrites every `.exp` file with the corresponding `.out` file. It is useful when some expected-output files are incorrect or are missing. It can be used, in particular, after a new source file has been added in the directory `test/inputs`, to create a new expected-output file.

3 Required software

To use the sources that we provide, you need OCaml and Menhir. Any reasonably recent version should do. You also need the package `pprint`. Assuming that you have installed OCaml via `opam`, please issue the following commands:

```
opam update
opam install menhir pprint
```

4 Evaluation

You should proceed with tasks 1–4 in order. Task 1, 2 and 3 are essential. Task 4 is slightly more complex, but requires a small amount of code. Tasks 0 is small, easy, and independent of others, but necessary for automatic comparison with the test suite.

There are 3 bonus points reserved for extensions. You need not do many, and certainly not all of them to acquire the 3 points. One well-done extension or two smaller extensions may be enough to get the full bonus!

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.

5 What to send

When you are done, please send to Didier Rémy a `.tar.gz` archive containing your completed programming project.

6 Deadline

Please send your project on or before **February 6, 2022**.