IIT CS443: Compiler Construction

Project 4: MiniML to C Compiler

Prof. Stefan Muller

Out: Thursday, Oct. 16
Due: Thursday, Oct. 31, 11:59pm CDT

Total: 50 points (Programming: 45 points, Test case: 5 points)

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- The starter code for Project 4 has been distributed through a pull request to your Project 2 GitHub repo. Merge this pull request into your main branch to start working. (This shouldn't cause merge conflicts, but if it does and you aren't sure how to handle them, ask.)
- **Important new submission instructions!** When you want to submit, commit the latest changes to your GitHub repo **with a commit message that clearly indicates this is your Project 4 submission** (e.g., "Project 4 Submission" would be a good commit message). This is so I know which commit to grade. If you resubmit later, just use a similar commit message. I'll grade the last commit that clearly indicates it's a Project 4 submission (and count your late days based on the time stamp of this submission).
- Compile (by running `make`) before submitting. **Submissions that don't compile will not get credit.**

Collaboration and Academic Honesty

You may work in groups of at most 2 on this project. Read the policy on the website and be sure you understand it.

1 MiniML Language Specification

The MiniML abstract syntax is below, using red | characters between BNF productions to distinguish them from | characters that actually appear in the MiniML syntax. MiniML is designed to be a subset of OCaml, and the semantics are the same as for OCaml (so rather than give a full spec here, you can just look at the OCaml docs). Please feel free to ask any clarification questions at all on Discord to make sure you understand the spec before starting the compiler.

Types	t	::=	int | bool | unit | t list | t -> t | $t * t$
Binary Operators	bop	::=	$+$ | $-$ | $*$ | / | AND | OR | $<$ | $\leq$ | $>$ | $\geq$ | <> | $=$
Unary Operators	$unop$	::=	$\sim -$ | not
Constants	c	::=	num | true | false | () | [ ]
OptAnnot	ot	::=	ϵ | $: t$
Expressions	e	::=	x | c | e bop e | $unop$ e | fun $(x : t)$ -> e | if e then e else e | let x $ot = e$ in e
			| let f $(x : t)$ $ot = e$ in e | let rec f $(x : t) : t = e$ in e | let $(x, y) = e$ in e | e e
			| match e with | [ ] -> e | h::t -> e | (e, e) | e::e | $(e : t)$
Declarations	d	::=	e | let x $ot = e$ | let f $(x : t)$ $ot = e$ | let rec f $(x : t) : t = e$
Programs	p	::=	d;; | d;; p

Note some important differences with OCaml:

- While OCaml accepts both $-$ and $\sim -$ for integer negation, we will allow only the latter to prevent some syntactic ambiguity.
- Type annotations on function arguments are *required*. Return type annotations are *optional*, except for recursive functions. Recursive functions must also have their return type annotated (though this is actually enforced by type checking, not the parser).
- Lists, pairs, and functions are the only non-base types.
- Pattern matching is very limited: only the syntactic forms above are allowed.
- Tuples (both in pattern matches and when constructing them) must be enclosed in parentheses. I consider this a feature rather than a limitation. Tuples of size > 2 are allowed; the parser desugars them to nested pairs, so, e.g., (1, 2, 3) would desugar to (1, (2, 3)). Only pattern matching on pairs is allowed, though, so you'd need to match against (1, 2, 3) with `let (one, twothree) = (1, 2, 3) in let (two, three) = twothree ...`.
- Functions take only one argument. You can make multi-argument functions using currying or using tuples of arguments.
- MiniML is purely functional, so, e.g., the order you evaluate expressions doesn't matter (did you know OCaml evaluates function arguments right-to-left?)
- Each top-level declaration or expression in a program is required to be followed by a double semicolon (;;). OCaml allows this mostly for historical reasons.

None of the above contradicts OCaml, so all MiniML programs are valid OCaml programs and you can test the desired behavior of your test cases by running them through OCaml.

2 Compilation Strategy

Below is an overview of how I suggest you build your compiler. The scaffolding code distributed with the assignment assumes you're doing it this way, so this'll be less work. But, technically, it's up to you.

Environments. I suggest keeping variable names as they are in the program for easier debugging, but keeping around an "environment record" mapping variable names to their deBruijn indices. An environment then is just a list of values (indeed, we will reuse the same struct for environments and lists). The current environment is kept in a variable named `__env`. The function `lookup_in_env i t` returns C code that looks up the i^{th} deBruijn index in the current environment and casts it to type t. This produces a call to the function `__lookup`, whose code is contained in the definition `lookup` in `ml.c`, and which will be automatically included in the C file you generate. Functions `extend_env` and `extend_with_placeholder` extend the environment with a binding (and placeholder binding, respectively) and return the code to perform the environment extension, paired with the new environment record. Both take the current environment record as their first argument. Function `pop_env` takes the current environment record and returns a pair of the code to pop the first binding of the environment, and the new environment record.

Closures. We will represent closures using the following structure, whose declaration is automatically added to C files you produce:

```
struct __clos{
  __list clos_env;
  (int(*)()) clos_fun();
};
```

The first field is the environment, and the second is the function pointer. The type of `clos_fun` is that of a pointer to a function that takes no arguments and returns `int`. The C AST for this type is bound for you as `fptr_typ`. The field names and signatures are bound in `mlc.ml` under the comment "Definitions for compiling closures" so you don't have to hardcode them. The function `init_struct` is useful for producing C code to initialize a new structure (and will be useful for closures, as well as lists and pairs). You can see an example of it being used in the code for `compile_cons` in `mlc.ml`.

Values and Types. Integers and Booleans are unboxed. Integers compile to `int` (as does the type `unit`; we will just represent `()` with 0). Booleans compile to `bool`. Functions are represented as closures, described above. Lists and pairs are boxed, and will be described below. The function `compile_typ` compiles MiniML types to MiniC types using the above strategy.

Lists. We will compile lists to linked lists in C, represented by the following structure, whose declaration is automatically added to C files you produce:

```
struct __list {
  int list_hd;
  __list list_tl;
};
```

Note that `int` is used as a default type for the head element. You will need to cast elements (or pointers) of other types in and out of `int`. (The variable `def_typ` under "Convenience functions" is an alias for the integer type, so you don't have to keep hardcoding it.) Rather than using tags, we'll just represent the empty list `[]` as a null pointer (i.e. `(__list)0`, recall the treatment of null pointers in MiniC).

The definitions `compile_nil` and `compile_cons` are provided for you under "Definitions for compiling lists" in `mlc.ml`.

Pairs. We will compile pairs to this structure, whose declaration is automatically added to C files you produce:

```
struct __pair {
  int pair_fst;
  int pair_snd;
};
```

The relevant field names are bound in `mlc.ml` under the comment "Definitions for compiling pairs."

3 Module Structure and Helpful Functions

Many functions you may find helpful are described above. Here, we'll go over the structure of the rest of the code. The definition of MiniC ASTs is in `c/c_ast.ml`, and the module is `C.Ast`. It's bound to `Ca` in `mlc.ml` (we can't call it `C` since that would hide all of the other definitions in the `C` structure). The definition of MiniML ASTs is in `ml/ml_ast.ml`, and the module is `ML.Ast`. This module is opened in `mlc.ml`. There are the usual ASTs for constants, types, binops, and unops, and one for expressions (the one for expressions is, as usual, defined as a descriptor `'a exp_` which is combined with a location and a `'a` to make an `'a exp`, and there are the usual functions for going back and forth). Unlike in the other languages

we've been compiling, there's nothing higher-level than expressions (the syntax allows a file to have multiple expressions/declarations but the parser desugars the whole program into one big expression).

There is also a type `cfunction` which is a record containing information about a C function, including its body (already compiled to C).

The function

`ML.Ast.new_var : unit -> string`

creates a new variable name. The function

`ML.Ast.new_mangle : string -> string`

"mangles" an ML variable name by appending a unique number to the end.

Some other helpful functions for building types of syntax you'll be using frequently are provided under "Convenience functions", along with the functions described above and many other functions you may find helpful. I highly suggest you look through these functions before you start so you don't wind up reimplementing any of them (if you have questions about what any of these functions do, feel free to ask on Discord). You can, of course, define your own functions along these lines if you find yourself using particular patterns over and over.

As usual, there is a pretty-printer for MiniML in the module `ML.Print`, and the code is in `ml/ml_print.ml`. The main functions are:

```
string_of_typ : Ml_ast.typ -> string
pprint_expr : Format.formatter -> ML.Ast.t_exp -> unit
```

4 Programming Task: MiniML Compiler (45 points)

Your task is to implement the two mutually recursive functions

1. `compile_body: env_record -> string -> var -> typ -> t_exp -> Ca.p_stmt list * Ca.p_exp * cfunction list`. The return value of this function, as we discussed in class, is a list of C statements that compute an expression, the final expression, and a list of nested closures. In this case, the expression you're compiling is the body of a function. The `env_record` argument is an environment record (described in more detail above) which is an association list (`(var * int) list`) mapping variables to their deBruijn indices. It's your responsibility to keep `env_record` in sync with the actual environment.

2. `compile_exp: env -> t_exp -> Ca.p_stmt list * Ca.p_exp * closure_typ list`, which compiles arbitrary expressions. The return value is the same as described above.

Some hints and additional information (ignore this at your peril!)

1. You are producing `Ca.p_stmt`s and `Ca.p_exp`s, so you don't have to include type information in the C code you produce; the MiniC type checker will do this. You *do* have to supply location information for every C statement and expression. It's valid to just use a "dummy" location for every AST node (the functions `mk_exp` and `mk_stmt` will make a `p_exp` and `p_stmt`, respectively, from an expression or statement description with a dummy location). However, it will make debugging easier if you copy over (at least some of) the location information from the ML AST to the C code you produce (so you'll be producing C code that records the location in the original MiniML source file the corresponding code came from). The function `cloc_of_mlloc : ML.Ast.loc -> C.Ast.loc` will help you here. That way, if your compiled code raises a C type error, the error message will show what ML code you compiled incorrectly.

2. Similar to above, recall that the `EVar` constructor in the MiniC AST takes both a `var` (just a string) and a `var_scope`, which is either `Local` or `Global`. The scope information is overwritten with the correct scope by the C typechecker, so when you generate an `EVar`, you can just make all the variables `Local`.

3. As we discussed in class, the order functions appear in the C file is important. The function `compile_prog`, which calls `compile_exp` and assembles the whole C file, adds functions to the C file in the *reverse* order in which they appear in the list of closures that is returned by `compile_exp`. Keep this in mind

when building up lists of closures. Is this brittle as hell? Yes. Would it work if we added mutually recursive functions to MiniML? No. But MiniC doesn't allow function declarations without a function body, so this is the best we can do.

5 Test Cases (5 points)

Each group should write (at least) one MiniML program that is not substantially the same as the test cases I've given you or others written by other students/groups (this is good incentive to write your tests early and post them before other groups!) You're encouraged to try and find corner cases and explore code paths other tests might have missed. Don't be too adversarial though; reasonable student solutions should pass your test.

Some other rules (the test cases I use for grading will follow these rules as well):

- Your test case should be syntactically valid according to the MiniML syntax in this document.
- Your test case should also be type-correct. That is, it should not raise a type error.
- The intended behavior of the test case *may* be to raise a runtime error.

Post your test case as a new thread on the "Project 4 Test Cases" discussion board on Blackboard. You can (and should!) test your compiler on other students' test cases. I may do so as well during grading. Feel free to ask clarification questions, note issues, etc., as replies in the threads created by other students.

6 Testing

Compile your code using `make` (in the top level of the source tree). This will produce the binary `main`, which you can use as follows to compile test programs:

```
./main -keepc -stopc tests/<file>.ml
```

This will parse and compile the file, and then type-check the resulting MiniC code. By default, it will output human-readable MiniC code in `tests/<file>.c`. Unfortunately, MiniC is not quite compatible with actual C, so you won't be able to test the output by compiling it using a C compiler. Instead, you'll need a MiniC compiler. Luckily, you should have one of those lying around from last week. If you replace `cllvm.ml` with your `cllvm.ml` from Project 3 (or the Project 3 solutions if you prefer) and compile, you can run:

```
./main -keepc -interpllvm tests/<file>.ml
```

This will compile and output the generated MiniC as above, but will also compile the generated MiniC to LLVM, output human-readable LLVM and run it through the LLVM interpreter, as in the Projects 2 and 3 testing.

If you're getting an `Unimplemented` exception (and you've completed all parts of Project 4), you may not have replaced `cllvm.ml` with your MiniC-to-LLVM compiler.

Running `make test` will run all of the tests listed in the `p4tests` file. If you add another test case, it won't automatically be run by `make test` unless you add it to `p4tests` following the existing examples. **Note:** This test script is at best barely production-ready. Feel free to modify it how you want (let me know if you make useful modifications I can push out in future assignments!). It may not be a substitute for running tests individually, as described above (in particular, if you get an error like "Expected: 42, Got:", this probably means that the test raised an exception; you should re-run that test manually to debug it.)