Honors Exam – Compilers – Spring 2020

Part A : Lamb

Over this exam we will use a simple base programming language Lamb which is largely a subset of (untyped) OCaml. Lamb includes the following features: anonymous higher-order functions of one argument, function application, variables, integers, ifzero-then (conditional on 0), and basic arithmetic binary operations + and -.

Here is an example Lamb program returning -7:

```
(function f -> f 2 3)
  (function x -> function y -> ifzero x - y then 22 else -7)
```

This program runs in OCaml except for the ifzero, a simplified conditional expression which means we do not require booleans in Lamb.

For coding questions below, you may use OCaml-like syntax and assume standard data structures and operations are available: lists, sets, hash tables, etc.

0. Write out an EBNF (i.e. a grammar) for Lamb. Note that there are several reasonable EBNFs, you only need to write out one of them which satisfies the informal description above.

1. Give an OCaml type definition which defines abstract syntax trees (ASTs) corresponding to your EBNF. Again there are several good answers.

2. Lamb is more like an intermediate language in a compiler as it lacks obviously needed syntax including let-definitions, booleans and boolean operators, equality on numbers, and OCaml-style if-then-else, but simple macros can define that syntax. For this question you are to develop a richer feature set language called ExtendedLamb. Give the ExtendedLamb AST (in the form of a new OCaml type) which includes all of the aforementioned additional syntax, and then write a translation function elamb_to_lamb which takes Extended Lamb ASTs and expands on these abbreviations to produce an equivalently-functioning Lamb AST.

Part B: MetaLamb

This question concerns metalanguages. You don’t have to know anything about metalanguages as we will give the details you need to know in this question. OCaml in fact traces its ancestry to a metalanguage — the “ml” in the name stands for “metalanguage”. ML, the predecessor of OCaml, was originally created as a language for reasoning about programs in a very simple programming language called PCF. The “meta” means it is a language for manipulating languages, in this historical case ML manipulating PCF.

For this question we are going to outline the design of MetaLamb, a metalanguage for Lamb. MetaLamb programs will at runtime be able to compose, compile, and
run Lambda programs. But, unlike ML/PCF, the MetaLamb syntax will largely be the same as Lamb’s, with only small extensions to support meta-operations.

Languages which can manipulate self-similar programs come in several different flavors; MetaLamb is more in the family of staged programming languages.

Here is an example of the kind of thing MetaLamb should be able to do (note for readability we will use let syntax in these examples as we know it can be added in an ExtendedLamb front-end as we saw in Part A)

```ocaml
let p1 = '(function x -> x + 1) 5' in
run p1
```

The use of special escapes such as `..` is common in metaprogramming — the code inside the escapes is Lamb, and the whole program is MetaLamb. We are going to do things a bit more simply however, and directly write Lamb-AST syntax inside of MetaLamb - this will make it more clear that we are just treating Lamb code as data. For example depending on the form of your AST in Part A the above example could perhaps be equivalently expressed in MetaLamb as

```ocaml
let p1 = Apply(Function(Var "x", Add(Var "x", Int(1))), Int(5)) in
run p1
```

The `run` keyword of MetaLamb should compile and run the (closed) program `p1`, and return a Lamb AST for the value returned.

To be clear, the only new syntax we add to Lamb to make MetaLamb is `run` and construction of Lamb-AST’s-as-data. There are no other new data types in MetaLamb, and for simplicity we will not even be able to `match` on these ASTs in MetaLamb (all for simplicity - in a real metalanguage we would want to destruct syntax as well).

With the AST syntax it is easier to show how Lamb programs can be composed in MetaLamb. Here for example is MetaLamb code which creates a Lamb add-n function for a given n:

```ocaml
let addn =
  function lambn -> Function(Var "x", (Add(Var "x", lambn)))
in
let p1 = function n -> addn (Int n) in
run (Apply((p1 1),(Int 5)))
```

This MetaLamb program should return `Int(6)`. Here, `addn` is a MetaLamb function taking some Lamb code expected to return an integer (the `lambn` variable should contain that code), and returning Lamb code for an add function over that value. The interesting feature of this program is how we can compose different pieces of Lamb syntax in MetaLamb to create a new Lamb program which we can then `run`.

Finally, here are the questions.

1. Define an OCaml type which can serve as the AST for MetaLamb.
2. We can in fact use MetaLamb as a macro preprocessor for Lamb, accomplishing a goal similar to ExtendedLamb of Part A.2. Write a MetaLamb function `letlamb v e1 e2` expressing “let v = e1 in e2” which when executed expands to a function application as in ExtendedLamb. A test case for your `letlamb` is a let-version of the example above:

```ocaml
let letlamb = ... fill in ... in
let p1 = letlamb (Var "x") (Int(5)) (Add(Var "x", Int(1))) in
run p1
```

3. Based on this one example, describe in words how MetaLamb could more generally be purposed to serve as a macro pre-processor for Lamb programs, similarly to `#define` in C programs. (Note that to make it usable in practice we would need to use the `...` convenience syntax as described above; ignore that issue here.)

4. We would like to extend a Lamb compiler to a MetaLamb compiler. Composing Lamb ASTs as in the examples is not so difficult, the difficult part is `run` which needs to compile and run the AST, and to return a value. For this question, outline the architecture of a MetaLamb compiler and runtime system which could support this behavior. You don’t need to describe how Lamb itself is compiled, only the extensions. Do try to cover all the key challenges of such a design.

5. To go even more meta, imagine MetaMetaLamb which supports metaprogramming over MetaLamb, MetaMetaMetaLamb, etc. In the limit we can define MetaStarLamb which allows for arbitrary such levels in a single language. Give an OCaml AST for MetaStarLamb and write a paragraph discussing some challenges of implementing a MetaStarLamb compiler.