CSE-321: Assignment 7 (100 points)

gla@postech.ac.kr

Early bird due at 11:59pm, May 24
Assignment due at 11:59pm, June 3

In this assignment, you will implement a translator for a pure functional programming language called Tiny ML(*TML*). TML is essentially a subset of Standard ML(*SML*) in that its core language is a subset of the core language of SML. In other words, every TML program is also a valid SML program!

TML is powerful enough to support most of the functional features of SML. It does *not* support polymorphic datatypes, mutual recursion/datatype declarations, and the module system. It *does* support polymorphic functions, recursive functions, recursive datatypes and pattern matching. Although TML does not directly support the **if-then-else** clause and other constructs of SML, it is not difficult to emulate them with those constructs available in TML.

The goal of this assignment is to familiarize you with some important concepts and techniques involved in programming language design and implementation. Specifically you will implement a translator which yields machine code for an abstract machine called Mach.

1 Fun with TML

The following TML program "list.tml" reverses an integer list, which demonstrates recursive datatypes, recursive functions and pattern matching.

```
datatype list = Nil | Cons of (int * list);

val rec append =
  fn Nil => (fn x => Cons (x, Nil))
   | Cons (h, t) => (fn x => Cons (h, append t x));

val rec reverse =
  fn Nil => Nil
   | Cons (h, t) => append (reverse t) h;

val l = Cons (1, Cons (2, Nil));

reverse l;;
```

The following TML program "poly.tml" takes two functions and returns their composition, which demonstrates polymorphic functions.

```
fn f => fn g => fn x => g (f x);;
```

You can run these programs using your implementation.

2 Syntax

The concrete grammar for TML is defined as follows:

$$
\begin{array}{rcl}
\mathit{sconty} & ::= & \textbf{int} \mid \textbf{bool} \mid \textbf{unit} \\
\mathit{sconpat} & ::= & \mathit{num} \mid \textbf{true} \mid \textbf{false} \mid () \\
\mathit{scon} & ::= & \mathit{num} \mid \textbf{true} \mid \textbf{false} \mid () \\
\mathit{op} & ::= & + \mid - \mid * \mid = \mid <> \\
\mathit{ty} & ::= & \mathit{sconty} \mid \mathit{tycon} \mid (\mathit{ty} * \mathit{ty}) \mid (\mathit{ty}- > \mathit{ty}) \mid (\mathit{ty}) \\
\mathit{pat} & ::= & \_ \mid \mathit{sconpat} \mid \mathit{vid}\ \langle \mathit{pat} \rangle \mid (\mathit{pat},\ \mathit{pat}) \mid (\mathit{pat}) \mid (\mathit{pat} : \mathit{ty}) \\
\mathit{mrule} & ::= & \mathit{pat} => \mathit{exp} \\
\mathit{mlist} & ::= & \mathit{mrule} \langle \mid \mathit{mlist} \rangle \\
\mathit{exp} & ::= & \mathit{scon} \mid \mathit{vid} \mid \textbf{fn}\ \mathit{mlist} \mid \mathit{exp}\ \mathit{exp} \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid (\mathit{exp},\ \mathit{exp}) \\
 & & \textbf{let}\ \mathit{dec}\ \textbf{in}\ \mathit{exp}\ \textbf{end} \mid (\mathit{exp}) \mid (\mathit{exp} : \mathit{ty}) \\
\mathit{conbinding} & ::= & \mathit{vid}\ \langle \textbf{of}\ \mathit{ty} \rangle \\
\mathit{conbind} & ::= & \mathit{conbinding}\ \langle \mid \mathit{conbind} \rangle \\
\mathit{dec} & ::= & \textbf{val}\ \mathit{pat} = \mathit{exp} \mid \textbf{val rec}\ \mathit{pat} = \mathit{exp} \\
 & & \textbf{datatype}\ \mathit{tycon} = \mathit{conbind} \\
\mathit{dlist} & ::= & \langle \mathit{dec}; \rangle * \\
\mathit{program} & ::= & \mathit{dlist}\ \mathit{exp}
\end{array}
$$

All the grammar variables (nonterminals) are written in *italic*. For example, *pat*, *exp* and *program* are grammar variables. The meaning of each grammar variable is defined in a corresponding grammar rule. The only exceptions are *num*, *vid* and *tycon*. *num* denotes an integer constant, which is an optional negation symbol ∼ followed by a non-empty sequence of decimal digits **0** · · · **9**. Both *vid* and *tycon* are alpha-numeric identifiers: any sequence *s* of letters, digits, apostrophes ' and underscore _ starting with a letter or an apostrophes, where *s* is not a reserved word. All TML reserved words are written in **boldface**. For example, **let**, **val** and **fn** are all reserved words used in TML.

Each grammar variable is read as follows:

- *num*: number
- *vid*: value identifier
- *tycon*: type constructor
- *sconty*: special constant type
- *sconpat*: special constant pattern
- *scon*: special constant
- *op*: primitive operation on integers of type **int** (returning an integer or boolean)
- *ty*: type
- *pat*: pattern
- *mrule*: match rule
- *mlist*: match rule list
- *exp*: expression
- *conbinding*: constructor binding
- *conbind*: constructor binding list
- *dec*: declaration
- *dlist*: declaration list
- *program*: TML program

Each grammar rule consists of a grammar variable on the left side and its expansion forms on the right side. Expansion forms are separated by |. For example, *scon* can be expanded into *num*, **true**, **false** or (). A pair of brackets ⟨⟩ enclose an optional phrase. For example, *conbinding* is expanded into either *vid* or *vid* followed by **of** *ty*. Starred brackets ⟨⟩∗ represents zero or more repetitions of the enclosed phrase. For example, *dlist* can have zero or more *dec*'s each of which is followed by ;. Note that the end of the program is followed by ;;.

There are further syntactic restrictions:

- No pattern *pat* can have the same value identifier *vid* twice. For example, $(x,\ y)$ is a legal pattern while $(x,\ x)$ is not.
- For each '**val rec** *pat* = *exp*', *exp* must be of the form '**fn** *mlist*'. In other words, '**val rec** *pat* = *exp*' can be used only to create functions.
- No '**val** *pat* = *exp*' can bind **true** or **false**.
- No *conbinding* can have duplicate value identifiers. For example, **datatype** $t = A \mid A$ **of int** is not valid.

"ast.ml" defines types and datatypes for the grammar variables in the structure `Ast`. The representation for each grammar variable in "ast.ml" is as follows.

- *num*: type `int`
- *vid*: type `vid`
- *tycon*: type `tycon`
- *sconty*: datatype `ty`(constructors `T_INT`, `T_BOOL`, `T_UNIT`)
- *scontpat*: datatype `pat`(constructors `P_INT`, `P_BOOL`, `P_UNIT`)
- *scon*: datatype `exp`(constructors `E_INT`, `E_BOOL`, `E_UNIT`)
- *op*: datatype `exp`(constructors `E_PLUS`, `E_MINUS`, `E_MULT`, `E_EQ`, `E_NEQ`)
- *ty*: datatype `ty`
- *pat*: datatype `pat`
- *mrule*: datatype `mrule`
- *match*: type `mrule list`
- *exp*: datatype `exp`
- *conbinding*: datatype `conbinding`
- *conbind*: type `conbinding list`
- *dec*: datatype `dec`
- *dlist*: type `dec list`
- *program*: type `program`

See the structure `Ast_valid` in "ast_valid.ml" for syntactic validation.

`Ast_valid` declares a function `vprogram` which takes an `Ast.program` value, checks whether the `Ast.program` value obeys the syntactic restrictions, and returns the `Ast.program` value. `vprogram` raises `AstValidateError` if the `Ast.program` value violates the syntactic restrictions.

You can use the structure `Ast_print` defined in "ast_print.ml" to print the content of grammar variables when debugging your code. The meaning of each function in the structure should be clear. For example, `exp2str exp` returns a string representation of the `Ast.exp` value `exp`.

3 Typing and Monomorphising

We have obtained an `Ast.program` value from a TML source program, which obeys the syntactic restrictions. Now `Typing.tprogram` from "typing.cmo" (the result of compiling "typing.ml" which is not distributed to students) takes an `Ast.program` value and produces a `Core.program` value. Any well typed `Ast.program` value should be translated to a `Core.program` by `tprogram`. If the `Ast.program` value does not typecheck, `tprogram` should raise `TypingError`. The remaining part is to implement a translator which yields machine code.

When translating a TML program into machine code, we may use the type of expressions to optimize machine code. For example, let us consider the translation of `(fn x => x) ()`. Note that `x` in the body of `(fn x => x)` has `unit` type. Since the only possible value for `unit` type is `()`, the machine code for `(fn x => x)` may return `()` instead of loading the argument `x` and returning it. Furthermore, the machine code for `(fn x => x)` does not use the argument, so we need not pass `()` to `(fn x => x)` when translating `(fn` x => x) (). Using this optimization, we save the cost of passing and loading (). This example illustrates that types can be useful to optimize machine code.

In the presence of polymorphic functions, however, it becomes more complicated to use these types to optimize machine code. For example, let us consider the following TML program:

```
val f = (fn x => x);

(f (), f 1);;
```

This TML program contains f whose type is 'a -> 'a. When translating this program, we cannot translate (fn x => x) into machine code that returns () as before. Rather, we should translate it into machine code that loads the argument x and returns it because the type of x is unknown. Consequently, we should pass () to f when translating f (). Although f () looks similar to (fn x => x) (), we cannot optimize the machine code for this TML program as we optimize the machine code for (fn x => x) ().

In order for you to have more chances to optimize machine code, we use a technique called *monomorphising* which eliminates these polymorphic functions. The basic idea is simple: polymorphic functions are specialized for each type that they are actually used with. For example, we may *monomorphise* the previous TML program as follows:

```
val f_1 = (fn x : unit => x);
val f_2 = (fn x : int => x);

(f_1 (), f_2 1);;
```

Because f is used with unit -> unit type at f (), f is specialized into f_1 which has unit -> unit type. Similarly, f is specialized into f_2 which has int -> int type. Now, x in the body of f_1 has unit type, and the machine code for (fn x : unit => x) may use the aforementioned optimization.

core2mono in "monomorphise.ml" is an implementation of this technique. It monomorphises a valid Core.program value, which may contain polymorphic functions, to a Mono.program value, which contains only monomorphic functions, and raises MonomorphiseError if the Core.program value is not valid.

Note that your translator will not take a Core.program value which is the result of your type inference algorithm, but takes a Mono.program value which is the result of monomorphising the Core.program value. *Hence your translator does not need to consider polymorphic functions at all.*

4 Translation

Finally, we are ready to implement a translator that yields machine code for the abstract machine Mach. The objective here is to implement the function program2code in the structure Translate in "translate.ml", which takes a Mono.program value and produces a Mach.code value. Strictly speaking, we should first explain the dynamic semantics of TML, that is, how to evaluate TML programs. This in turn would require a complete implementation of either an interpreter or a compiler along with an abstract or concrete machine on which generated code runs. Fortunately, TML is a subset of SML, and we adopt the relevant part of the dynamic semantics of SML, which we assume you know exactly, and use it as the dynamic semantics of TML; we will not formally present the precise dynamic semantics to you.

We will not guide you through the whole implementation of a translator, we will provide you with the basics for implementing a translator and a few hints only. You will have to come up with your own strategy to implement a translator correctly and to optimize it. Since there are many different reasonable strategies, we would not be surprised that each group's translator produces different code for a common TML program. Your score will be based upon not only the correctness but also the performance of your translator. The performance will be measured in terms of both memory use and running time of Mach code produced by your translator (not the translator itself); we do not measure the amount of memory and time required for your translator to translate TML programs.

First read the description of the abstract machine Mach, which is self-contained and available as a separate document. It is important to have a good grasp of Mach because your translator will produce machine code running on Mach. Do not jump to coding right after reading the description of Mach. We suggest that you spend enough time thinking about the design of your translator before you start any coding.

4.1 Environment

A translator can maintain an "info" environment to keep various kinds of information about the source code. Then the question is: what kinds of information would we store in an "info" environment? Since there is no standard implementation of translators, you can choose your own definition of environments to be used by your translator; a particular kind of information may be used by one translator but not by other translators. However, there are a few kinds of information which would be employed by all different translators. Consider the following expression:

```
f (x, y)
```

To translate this expression, we need to know where in Mach the values of `f`, `x` and `y` are stored at the point when the code for this expression is executed. Note that we are not seeking out the locations of `f`, `x` and `y` in the source program. This may sound subtle, but it should be clear once you understand that we are about to produce machine code for this expression. For example, the value of `x` may be in the machine stack, in a register, or in a heap chunk; it does not matter where the definition of `x` appears in the source code because we know that the source code typechecks and the definition of `x` is valid. (Recall the catchphrase of SML "Well typed programs cannot go wrong", which also applies to TML because TML is a subset of SML)

If we can tell that the value of `x` is actually a constant or we can directly access its value through either register reference or memory reference, we do not need to produce new code to retrieve its value. Otherwise, we may have to produce new code to retrieve its value. For example, if AX stores a heap address *ha*, the second memory cell in the heap chunk associated with *ha* stores another heap address *hb*, the first memory cell in the heap chunk associated with *hb* stores the actual value of `x`, we may have to execute a few instructions to reach the memory cell containing the value of `x`. The following code retrieves the value of `x` and stores it back in ax; below we omit the structure name `Mach` and list notations:

```
MOVE (LREG tr, REFREG (ax, 1))
MOVE (LREG ax, REFREG (tr, 0))
```

This observation leads to a concept called *location*: a location specifies how to retrieve a `Mach.avalue` value. The following is the definition of datatype `loc` defined in "translate.ml". The function `loc2rvalue` may be helpful to you which takes a `loc` value and generates code and `Mach.rvalue` for the corresponding variable.

```
(* location *)
type loc =
    L_INT of int          (* integer constant *)
  | L_BOOL of bool        (* boolean constant *)
  | L_UNIT                (* unit constant *)
  | L_STR of string       (* string constant *)
  | L_ADDR of Mach.addr   (* at the specified address *)
  | L_REG of Mach.reg     (* at the specified register *)
  | L_DREF of loc * int   (* at the specified location with the specified offset *)
```

Then, a mapping from `Mono.avid` to `loc` can be used to keep track of where in Mach each variable is stored. We refer to such a mapping as a variable environment, which can be part of the "info" environment; `('k, 'v) dict` is a dictionary with key type `'k` and value type `'v` (defined in "dict.ml"):

```
type venv = (Mono.avid, loc) dict  (* variable environment *)
```

Another kind of information that can be useful during translation is the number of value bindings found within a function. Consider the following part of code:

```
val f =
  fn x =>
    let
      val y = x
    in
      let
        val z = x
      in
        ...
      end
    end
```

y and z can be thought of as local variables in the sense that they are both defined in value bindings in the body of the function f. If we are to store local variables in memory, we need to keep track of how many value bindings there have been so far. For example, if we choose to store local variables in stack using locations L_DREF (L_REG bp, i) for non-negative integer i, the location for a new value binding would depend on the number of value bindings declared before it (unless we employed a particular optimization idea). Thus, in executing the Mach code for the function f, we could use locations L_DREF (L_REG bp, 0) and L_DREF (L_REG bp, 1) for y and z respectively. With the assumption that the argument to f is stored in ax, a translator could generate the following code:

```
LABEL "code_f"
MOVE (LREFREG (bp, 0), REG ax)
MOVE (LREFREG (bp, 1), REG ax)
...
```

Note that the assumption that the argument to f is stored in AX is just one possible choice. If your strategy pushes an argument to the stack before executing a CALL instruction, the resultant code will be different:

```
LABEL "code_f"
MOVE (LREFREG (bp, 0), REFREG (bp, -3))
MOVE (LREFREG (bp, 1), REFREG (bp, -3))
...
```

The environment in "translate.ml" employs the above two ideas and is defined as follows:

```
type env = venv * int
```

Of course you may extend this definition of environments to implement your own idea.

4.2 Translation functions

We can facilitate and systematize the implementation of program2code by introducing translation functions corresponding to classes of elements of TML defined in structure Mono. We provide you with the types of these functions in a particular implementation strategy. Note that we do not define an exception: your translator *must* always produce valid Mach code (why)?

```
val pat2code : label -> label -> loc -> Mono.pat -> code * venv
val patty2code : label -> label -> loc -> Mono.patty -> code * venv
val exp2code : env -> label -> Mono.exp -> code * rvalue
val expty2code : env -> label -> Mono.expty -> code * rvalue
val dec2code : env -> label -> Mono.dec -> code * env
val mrule2code : env -> label -> label -> Mono.mrule -> code
```

The above six functions are one possible choice of translation functions (`label`, `code` and `rvalue` denote `Mach.label`, `Mach.code`, and `Mach.rvalue` respectively). As seen from the types, $< T >$`2code` generates code for a `Mono.`$< T >$ value. The intended meaning of each function is as follows:

- `pat2code` l_{start} $l_{failure}$ l *pat* returns (*code*, *venv*) where *pat* is a pattern, l_{start} is the label that *code* begins with, $l_{failure}$ is the label that the execution jumps to when the pattern matching fails, l is the location of an `Mach.avalue` value which will be matched against *pat*, *code* is the resultant code, and *venv* is the variable environment introduced by *pat* and l.
- `patty2code` is similar to `pat2code`.
- `exp2code` *env* l_{start} *exp* returns (*code*, *rvalue*) where *exp* is an expression, *env* is an environment, l_{start} is the label that code begins with, *code* is the resultant code, and *rvalue* is a `Mach.rvalue` value which stores the value of *exp*.
- `expty2code` is similar to `exp2code`.
- `dec2code` *env* l_{start} *dec* returns (*code*, *env'*) where *dec* is a declaration, *env* is an environment, l_{start} is the label that *code* begins with, *code* is the resultant code, and *env'* is a new environment induced by *dec*(not including *env*).
- `mrule2code` *env* l_{start} $l_{failure}$ *mrule* returns *code* where *mrule* is a match rule, *env* is an environment, l_{start} is the label that *code* begins with, $l_{failure}$ is the label that the execution jumps to when the pattern matching fail, and *code* is the resultant code.

As an example, consider `pat2code`. Below is an implementation for constructor `P_INT` according to the above definition(`venv0` is an empty variable environment):

```
| pat2code saddr faddr l (P_INT i) =
  let (code, rvalue) = loc2rvalue l in
  let code' = clist [JMPNEQ (ADDR (CADDR faddr), rvalue, INT i)]
  in
    (cpre [LABEL saddr] (code @@ code'), venv0)
```

If the `Mach.avalue` value located by `l` is not `Mach.AINT i`, the generated code moves execution to the label specified by `faddr`.

You can use the above translation functions, modify the types of these functions, or come up with your own translation functions depending on your implementation strategy. All you have to do is to implement `program2code` correctly. For example, you could have all the translation functions return multiple piece of code which could be placed in the final code independently of one another. Alternatively, you could have all the above translation functions return a single contiguous piece of code, which may be inefficient in some cases.

4.3 Design choices

Before writing code for `program2code` and all its auxiliary functions, you want to take into consideration a number of design choices. These design choices will determine the conciseness and correctness of your code as well as the efficiency of the translated code. Please think about these design choices *before* you write code for `program2code`, *not* while you are writing code for `program2code`. Think about possible problems incurred by a particular design choice before jumping to coding. Judicious design choices will eventually save your time and also result in an elegant translator. Below are eleven design considerations that we recommend you think about before implementing your translator.

Representation for `T_PAIR` A well typed TML expression will eventually evaluate to a TML value. For example, `1` + `1` will evaluate to `2`. Depending on its type, a TML value may require a single memory cell or multiple memory cells. For example, a single memory cell is enough to store an integer/boolean/unit/string TML value. On the other hand, for instance, a TML value of type (`int` $*$ `int`) requires at least two memory cells. We could create a heap chunk of two memory cells, store each integer to one memory cell, and then use the heap address to represent the TML value. In this case, we may not be able to reclaim the heap chunk. Alternatively, we could store each integer onto the stack consecutively and the stack address relative to bp as its representation, in which case we must make sure that the value is used only within the function currently being executed (because a `CALL` or `RETURN` would change the content of bp).

How would you represent a value from type `Mono.T_PAIR` (`ty`$_1$`, ty`$_2$) when the representations for `ty`$_1$ and `ty`$_2$ are known.

Representation for `T_NAME` Consider the code below, `B 1` is of type `t`, and the expression will evaluate to `false`. Since matching `B 1` against `A` must fail, the TML value to which `B 1` evaluates must record its associated data constructor `B`. Otherwise we would be unable to decide whether `B 1` matches with `A` or not. Therefore, any TML value of type `t` must store its associated constructor.

```
datatype t = A | B of int;

(fn A => true | B _ => false) (B 1);;
```

How would you represent a value from the type `Mono.T_NAME` (`tyname`)?

Representation for `T_FUN` Recall that the value of an SML function is called a closure, which consists of the definition of the function and its context. Likewise, the closure for a TML function `f` could be defined as the machine code for `f` and its context. Since we can uniquely locate the machine code for `f` with its address (or its starting label), we define the closure for `f` as the combination of the address of the machine code for `f` and its context. As an example, consider the following code:

```
val a = 1;
val f = fn x => x + a;
```

The translator will produce a piece of code for the expression `fn x => x + a`, and its address(or the starting label of the code) will be part of the closure for `f`. Its context must store the TML value of `a` at the point when the code from the expression `fn x => x + a` begins to be executed(not produced by the translator).

How would you represent contexts? Then, how would you represent closures?

Recursive value bindings A recursive value binding returns a closure. Since the function defined in the recursive value binding appears in its body, the context in the closure itself must store the TML value of the function. Specifically, if a recursive value binding `val rec f = e` generates a closure C with a context Ctx, Ctx itself must store C as the TML value of `f`.

How would you implement `dec2code` to deal with recursive value bindings?

Context switch Suppose that the TML value of a function `f` is a starting address s and a context Ctx. When `f` is actually invoked, execution must be transferred to s. Furthermore, the context Ctx must be activated so that the reference to non-local variables within `f` return correct TML values.

How would you activate a new context when a function is invoked? How would you restore the old context when the function returns?

Hint Although CP is not a special register, it can be used to hold the current context throughout execution as its name implies.

Function arguments and return values Since every function takes an argument, the caller (or the part of code invoking a function) must deliver an argument to whatever function it invokes. Likewise, every function has a return value, and it must deliver a return value to its callers.

How would you pass arguments and return values?

Hint A typical way of achieving this is by pushing arguments onto the stack before executing `CALL` and storing return values in a register before executing `RETURN`.

Distinction between functions and data constructors The syntax for invoking functions and using data constructors with an argument is the same. However, after the typechecking process, every data constructor with an argument is represented with `Mono.E_VID (avid, Mono.CON_F)` for a data constructor identifier `avid`, not with `Mono.E_FUN mlist` for a list of match rules `mlist`. Still the TML value of a data constructor is expected to be a closure because it is treated as a function. For example, the following TML program typechecks:

```
datatype t = A of int | B of int;

(fn true => A | false => (fn i => B i)) true;;
```

How would you create a closure for data constructors?

Hint The context part of a closure for a data constructor can be ignored and could be used for other purpose, and a datatype binding can create code for each data constructor in it. Alternatively all data constructors in the program can share common code.

Translating programs A TML program is not a TML expression. How would you translate a TML program using the functions that handle expressions and declarations? Make sure that generated code contains a label `Mach.START_LABEL`, from which the execution begins.

Registers Mach has 29 general purpose registers except sp, bp, and zr. All of them are functionally equivalent. The cost of accessing registers is 10 times lower than accessing memory directly. Thus making good use of registers instead of memory whenever possible will greatly improve the speed of the translated code.

How would you exploit 29 general registers? Your translator is not required to use all these registers, but try to use as many registers as possible.

Heap Mach is equipped with an infinite amount of heap memory. However, `FREE` causes Mach to reclaim those heap chunks that are no longer used. Since another criterion on evaluating your translator is the minimum amount of memory required (including all of code, stack, and heap memory), `FREE` could be an important way to reduce the overall memory requirement.

In what situation would you issue `FREE` instructions?

Optimizations There are a number of optimization strategies that can be employed in implementing the translator. For example, $1 + 2$ can be reduce to 3 during the translation process so that it does not have to produce any unnecessary code. As another example, for a tail-recursive function, your translator generate code which consumes a fixed amount of stack regardless of how many times the functions is invoked recursively. This is where your creativity comes to play.

how would you optimize your translator so as to reduce both the execution time and the memory requirement in general?

4.4 Instruction

Implement `program2code` in "translate.ml". `program2code` should translate every valid `Mono.program` value producing a TML value to a `Mach.code` value whose execution terminates with `HALT` *rvalue* where *rvalue* represents the TML value. If it cannot produce a TML value, the execution must terminate with `EXCEPTION`.

Note that all we care about is your implementation for `program2code`. You can introduce new functions, new structures, new functors or whatever you consider to be instrumental to your implementation. However, place all new ones in the same file "translate.ml."

If the source TML program has no input, you can simulate the whole program execution during the translation and generate very short yet correct code. Since one of the goals of this assignment is to teach you how to translate a functional program, you are not supposed to do this. In other words, do *not* try to directly find the result of a whole TML program by simulating it. However, you can optimize part of a source TML program as in reducing `1 + 2` to `3`.

Your implementation should avoid using global mutable references because the main function `program2code` can be called not just once but many times during testing. If you choose to use global mutable references, each invocation of `program2code` should initialize them explicitly.

5 Testing

After implementing "translate.ml," run the command `make` to compile the sources files.

```
$ make
ocamlc -thread  -c set_type.ml -o set_type.cmo
ocamlc -thread  -c dict.ml -o dict.cmo
ocamlc -thread  -c mach.mli -o mach.cmi
ocamlc -thread  -c mach.ml -o mach.cmo
File "mach.ml", line 307, characters 44-65:
Warning 52: Code should not depend on the actual values of
this constructor's arguments. They are only for information
and may change in future versions. (See manual section 8.5)
...
ocamlyacc parser.mly
29 shift/reduce conflicts.
ocamlc -c parser.mli
ocamlc -c parser.ml
ocamllex lexer.mll
65 states, 3973 transitions, table size 16282 bytes
ocamlc -c lexer.ml
ocamlc -thread  -c inout.ml -o inout.cmo
ocamlc -thread  -c ast_valid.ml -o ast_valid.cmo
ocamlc -thread  -c ast_print.ml -o ast_print.cmo
ocamlc -thread  -c core.ml -o core.cmo
ocamlc -thread  -c core_print.ml -o core_print.cmo
ocamlc -thread  -c mono.ml -o mono.cmo
ocamlc -thread  -c monomorphise.ml -o monomorphise.cmo
ocamlc -thread  -c mono_print.ml -o mono_print.cmo
ocamlc -thread  -c translate.ml -o translate.cmo
ocamlc -thread  -c test.ml -o test.cmo
ocamlc -thread  -c hw7.ml -o hw7.cmo
ocamlc -o lib.cma -a set_type.cmo dict.cmo mach.cmo ast.cmo parser.cmo lexer.cmo inout.cmo ast_valid
translate.cmo test.cmo
ocamlc -o hw7 lib.cma hw7.cmo
```

To test your code, you can run `hw7`.

```
$ ./hw7
Parse OK
```

```
val x = 1;
(+) ((x, 1))

Ast_valid.vprogram OK
val x = 1;
(+) ((x, 1))

Typing.tprogram OK
val (x : int) = (1 : int);

(((+ : ((int, int)->int))) ((((x : int), (1 : int)) : (int, int))) : int);;

Monomorphise.core2mono OK
val (x_1 : int) = (1 : int);

(((+ : ((int, int)->int))) ((((x_1 : int), (1 : int)) : (int, int))) : int);;

Translate.program2code OK
Execution OK
Please read test.tml.exe for details.
```

“test.ml” defines the function `test` in the structure `TestTyping` which takes a file name *s* and returns a `Core.program` value.

The following example shows the result when running `TestTyping.test` with “poly.tml”.
Example:

```
$ ocaml
        OCaml version 4.05.0

# #load "lib.cma";;
# open Test;;
# TestTyping.test "example/poly.tml";;
Parse OK
;
fn f => fn g => fn x => (g) ((f) (x))

Ast_valid.vprogram OK
;
fn f => fn g => fn x => (g) ((f) (x))

Typing.tprogram OK

(fn (f : (’3->’4)) => (fn (g : (’4->’5)) => (fn (x : ’3) => (((g : (’4->’5))) ((((f : (’3->’4))) ((x

- : unit = ()
#
```

The type variables in your implementation may differ from from those in the example.

“test.ml” also defines the function `test` in the structure `TestTranslate` which takes a `Mono.program` value and creates a file ”test.exe”. The new file shows the results of all the steps involved and the execution statistics.

In order to test your implementation, you may use the `Mono.program` values in the structure `MonoSample` in “test.ml”, or `Mono.program` values generated by `Typing.tprogram` and `Monomorphise.core2mono` if you already implement `Typing.tprogram` correctly.

The following example shows the result when running `TestTranslate.test` with `MonoSample.s1`, which translates `1 + 1` by your translator and creates "test.tml.exe" by executing the generated code on Mach.

Example:

```
$ ocaml
        OCaml version 4.05.0

# #load "lib.cma";;
# open Test;;
# TestTranslate.test MonoSample.s1;;
Translate.program2code OK
Execution OK
Please read "test.exe" for details.
- : unit = ()
#
```

The overall structure of "test.exe" is as follows:

```
Machine code:
_Start_CSE_321_HW7_:
...
```

(* Shows the generated Mach code. See Mach.instr2str for the format. *)

```
Execution begins at 0:
0  _Start_CSE_321_HW7_:
...
```

(* Shows the trace of execution. Each line begins the offset of the instruction within the code *)

```
Stack(0) =

&Heap_0 = [0] = 1 [1] = 1
```

(* Shows the contents of the stack and each memory cell in each heap chunk after the completion of the execution. In the above case, the stack is empty. The heap chunk at heap address 0 has two memory cells. Both cell contains an integer 1 *)

```
Execution statistics
        code size = 9
        max stack = 1
        max heap = 2
        instructions executed = 8
        memory read = 4
        memory write = 4
        register read = 9
        register write = 7
Total time cost = 184
Total memory cost = 12

Normal termination
        Result = 2
```

(* Shows the execution statistics. The "Total time cost" and "Total memory cost" figures will be considered in measuring the performance of your translator. The interpretation of "Result" will be used to test the correctness of your translator. *)

“test.ml” also defines the function `test` in the structure `TestAll` which takes a file name (as a string), parses the TML source program in the file, reconstructs the type of TML program, monomorphises the TML program, translates the TML program to a `Mach.code` value which runs on the abstract machine Mach, and returns it. You may use `TestAll.test` to test your `Translate.program2code` for any TML source program.

6 Submission instruction

Please submit your implementation “translate.ml” to your hand-in directory. If your Hemos ID is `foo`, your hand-in directory is:

```
/home/class/cs321/handin/foo/
```