CS 3110 Spring 2015	**Soft deadline: 3/26/15 11:59pm**
Problem Set 3	**Hard deadline: 3/28/15 1:00pm**
Version 0 (last modified January 13, 2021)	

Overview

In this assignment you will implement several functions over infinite streams of data, an algorithm for inferring types, and an interpreter for OCalf - a functional language containing the core features of OCaml.

Objectives

This assignment should help you develop the following skills:

- Developing software with a partner.
- Working with a larger OCaml source code base.
- Using software development tools like source control.
- Understanding the environment model and type inference.
- Experience with infinite data structures and lazy evaluation.

Additional references

The following supplementary materials may be helpful in completing this assignment:

- Lectures 7, 8
- Recitations 7, 8
- The OCaml List Module and the Printf Module
- Git Tutorial
- Real World OCaml, Chapter 6

Academic integrity

You are allowed to work with one other student in this class for this problem set. The sharing of code is only allowed between you and your partner; sharing code among groups is strictly prohibited. Please review the course policy on academic integrity.

Provided code

This is a larger project than those you have worked on in PS1 and 2. Here is an overview of the release code. We strongly encourage you to **read all of the `.mli` files before you start coding**, so that you know what functions are available and understand the structure of the source code.

Here is a brief list of the provided modules:

Streams is the skeleton code for part 2.

Eval and Infer are the skeleton code for parts 3 and 4. In addition to the functions that you must implement, there are various unimplemented helper functions defined in the `.ml` files. We found these helper functions useful when we implemented the assignment, but you should feel free to implement them, change them, extend them, or remove them. They will not be tested.

Printer contains functions for printing out values of various types. This will be extremely helpful for testing and debugging.

Examples and Meta These modules contain example values to help you start testing. `Examples` contains all of the examples from this document; `Meta` contains an extended example for parts 3 and 4.

Ast and TypedAst These modules contain the type definitions for parts 3 and 4 respectively. The `TypedAst` module also contains the `annotate` and `strip` function for converting between the two.

Parser contains `parse` functions that allow you to read values from strings. It will be useful for testing and debugging parts 3 and 4.

Running your code

In addition to using `cs3110 test` to test your code, we expect you to interact extensively with your code in the toplevel. To support this, we have provided a `.ocamlinit` file[1] that automatically makes all of your compiled modules available in the toplevel. For convenience, it also `open`s many of them.

A `.ocamlinit` file is just an OCaml file that is automatically `#use`d by the toplevel when it starts. We encourage you to add to it; any time you find yourself typing the same thing into the toplevel more than once, add it to your `.ocamlinit`!

In order for the files to load correctly, **you must compile them first**. If you change them, the **changes will not be reflected in the toplevel until you recompile**, even if you restart it. The file `examples.ml` references all other files in the project, so you should be able to recompile everything by simply running `cs3110 compile examples.ml`.

[1] Note that on Linux, files whose names start with '.' are hidden; use `ls -A` to include them in a directory listing.

Part 1: Source control (5 points)

You are required to use some version control system for this project. We recommend using `git`. As part of good software engineering practices, you should use version control from the very beginning of the project. For a tutorial on `git`, see the git tutorial in the additional references section.

You will submit your version control logs in the file `logs.txt`. These can be extracted from git by running "`git log --stat > logs.txt`" at the command line.

Part 2: Lazy Evaluation (30 points)

A **stream** is an infinite sequence of values. We can model streams in OCaml using the type

```
type 'a stream = Stream of 'a * (unit -> 'a stream)
```

Intuitively, the value of type `'a` represents the current element of the stream and the `unit -> 'a stream` is a function that generates the rest of the stream.

"But wait!" you might ask, "why not just use a list?" It's a good question. We can create arbitrarily long finite lists in OCaml and we can even make simple infinite lists. For example:

```
# let rec sevens = 7 :: sevens;;
val sevens : int list =
  [7; 7; 7; 7; 7; 7; ...]
```

However, there are some issues in using these lists. Long finite lists must be explicitly represented in memory, even if we only care about a small piece of them. Also most standard functions will loop forever on infinite lists:

```
# let eights = List.rev_map ((+) 1) sevens;;
(loops forever)
```

In contrast, streams provide a compact, and space-efficient way to represent conceptually infinite sequences of values. The magic happens in the tail of the stream, which is evaluated lazily—that is, only when we explicitly ask for it. Assuming we have a function `map_str : ('a -> 'b) -> 'a stream -> 'b stream` that works analogously to `List.map` and a stream `sevens_str`, we can call

```
# let eights_str = map_str ((+) 1) sevens_str;;
val eights_str : int stream
```

and obtain an immediate result. This result is the stream obtained after applying the map function to the first element of `sevens_str`. Map is applied to the second element only when we ask for the tail of this stream.

In general, a function of type `(unit -> 'a)` is called a **thunk**. Conceptually, thunks are expressions whose evaluation has been delayed. We can explicitly wrap an arbitrary expression (of type `'a`) inside of a thunk to delay its evaluation. To force evaluation, we apply the thunk to a unit value. To see how this works, consider the expression

```
let v = failwith "yolo";;
Exception: Failure "yolo"
```

which immediately raises an exception when evaluated. However, if we write

```
let lazy_v = fun () -> failwith "yolo";;
val lazy_v : unit -> 'a = <fun>
```

then no exception is raised. Instead, when the thunk is forced, we see the error

```
lazy_v ();;
Exception: Failure "yolo"
```

In this way, wrapping an expression in a thunk gives rise to **lazy evaluation**.

Exercise 1: Implementing Stream Functions (20 points).

(a) Implement a function

```
take : int -> 'a stream -> 'a list
```

that returns the first `n` elements of the stream `s` in a list. If `n < 0`, return an empty list.

(b) Implement a function

```
repeat : 'a -> 'a stream
```

that produces a stream whose elements are all equal to `x`.

(c) Implement a function

```
map : ('a -> 'b) -> 'a stream -> 'b stream
```

that (lazily) applies the input function to each element of the stream.

(d) Implement a function

```
diag : 'a stream stream -> 'a stream
```

that takes a stream of streams of the form

$$\begin{matrix} s_{0,0} & s_{0,1} & s_{0,2} & \cdots \\ s_{1,0} & s_{1,1} & s_{1,2} & \cdots \\ s_{2,0} & s_{2,1} & s_{2,2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

and returns the stream containing just the diagonal elements

$$s_{0,0}s_{1,1}s_{2,2}\cdots$$

(e) Implement a function

```
suffixes : 'a stream -> 'a stream stream
```

that takes a stream of the form

$$s_0 \quad s_1 \quad s_2 \quad s_3 \quad s_4 \quad \cdots$$

and returns a stream of streams containing all suffixes of the input

$$\begin{array}{cccc} s_0 & s_1 & s_2 & \cdots \\ s_1 & s_2 & s_3 & \cdots \\ s_2 & s_3 & s_4 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

(f) Implement a function

```
interleave : 'a stream -> 'a stream -> 'a stream
```

that takes two streams $s_0 s_1 \cdots$ and $t_0 t_1 \cdots$ as input and returns the stream

$$s_0 t_0 s_1 t_1 s_2 t_2 \cdots$$

Exercise 2: Creating Infinite Streams (10 points).

In this exercise you will create some interesting infinite streams. We **highly** recommend using the printing functions we have provided in `printer.ml`. A part of being a good software engineer is being able to implement tools that will help you test and implement code. You will have the opportunity to implement your own print function in a later exercise, but for now, you may use the ones we have provided. See `printer.mli` for more details.

(a) The Fibonacci numbers a_i can be specified by the recurrence relation $a_0 = 0$, $a_1 = 1$, and $a_n = a_{n-1} + a_{n-2}$. Create a stream `fibs : int stream` whose nth element is the nth Fibonacci number:

```
print_stream_int (fibs ());;
[0, 1, 1, 2, 3, 5, 8, 13, ...]
```

(b) The irrational number π can be approximated via the formula

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}.$$

Write a stream `pi : float stream` whose nth element is the nth partial sum in the formula above:

```
print_stream_float (pi ());;
[4., 2.66666666667, 3.46666666667, 2.89523809524,
3.33968253968, 2.97604617605, 3.28373848374,
3.017071871707, ...]
```

(c) The **look-and-say sequence** a_i is defined recursively; $a_0 = 1$, and a_n is generated from a_{n-1} by reading off the digits of a_{n-1} as follows: for each consecutive sequence of identical digits, pronounce the number of consecutive digits and the digit itself. For example:

- the first element is the number 1,
- the second element is 11 because the previous element is read as "one one",
- the third element is 21 because 11 is read as "two one",
- the fourth element is 1211 because 21 is read as "one two, one one",
- the fifth element is 111221 because 1211 is read as "one one, one two, two one".

Write a stream `look_and_say : int list stream` whose nth element is a list containing the digits of a_n ordered from most significant to least significant. Similar to the Fibonacci and pi streams, we strongly suggest you test your implementation. To print the **look-and-say sequence**, you can use

```
print_stream_int_list (look_and_say ());;
[[ 1 ], [ 1 1 ], [ 2 1 ], [ 1 2 1 1 ], [ 1 1 1 2 2 1 ],
[ 3 1 2 2 1 1 ], [ 1 3 1 1 2 2 2 1 ],
[ 1 1 1 3 2 1 3 2 1 1 ], ...]
```

Part 3: ML interpreter (40 points)

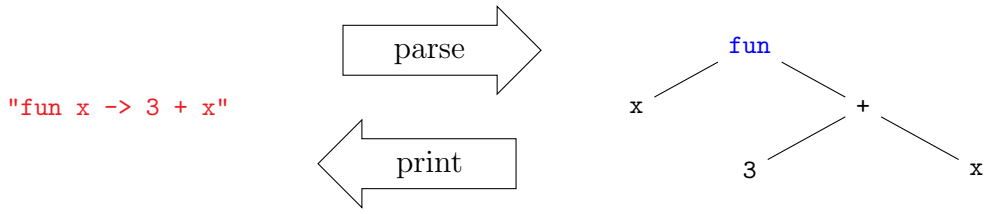
In this part of this assignment, you will implement the evaluation of an interpreter for a subset of OCaml called OCalf. This will provide functionality similar to the top-level loop we have been using for much of the semester.

Background on interpreters

An interpreter is a program that takes a source program and computes the result described by the program. An interpreter differs from a **compiler** in that it carries out the computations itself rather than emitting code which implements the specified computation.

A typical interpreter is implemented in several **stages**:

1. Parsing: the interpreter reads a string containing a program, and converts it into a representation called an **abstract syntax tree** (AST). An AST is a convenient data structure for representing programs: it is a tree containing a node for each subexpression; the children of the node are the subexpressions of the expression. For example, here is an abstract syntax tree representing the program `fun x -> 3 + x`:




```
utop# parse_expr "fun x -> 3 + x";;
- : expr = Fun ("x", BinOp (Plus, Int 3, Var "x"))
```

2. Type checking: the interpreter verifies that the program is well-formed according to the type system for the language—e.g. the program does not contain expressions such as `3 + true`.

3. Evaluation: the interpreter evaluates the AST into a final **value** (or runs forever).

We have provided you with a type `Ast.expr` that represents abstract syntax trees, as well as a parser that transforms strings into ASTs and a printer that transforms ASTs into strings. Your task for this part will be to implement evaluation (typechecking is part 4).

Your tasks

For this part of the assignment, you will be working in the file `eval.ml`. In particular, you will implement the function

```
eval : Eval.environment -> Ast.expr -> Eval.value
```

which evaluates OCalf expressions as prescribed by the **environment model**. For example, we can evaluate the expression "`if false then 3 + 5 else 3 * 5`" in the empty environment as follows:

```
eval [] (parse_expr "if false then 3 + 5 else 3 * 5");;
- : value = VInt 15
```

OCalf has the following kinds of values:

1. Unit, integers, booleans, and strings.
2. Closures, which represent functions and the variables in their scopes.
3. Variants, which represent user-defined values like `Some 3` and `None`; unlike OCaml, all variant values must have an argument. For example, `None` and `Nil` are represented as `None ()` and `Nil ()`.
4. Pairs, which represent 2-tuples. Unlike OCaml, all tuples are pairs; you can represent larger tuples as pairs of pairs (e.g. `(1,(2,3))` instead of `(1,2,3)`).

As a simple example of evaluation, using the notation learned in class, `[] :: 7 + 35 || 42`. Your interpreter should evaluate both sides of the `+` operator, which are already values (`7` and `35`), and then return an integer representing the sum of values `7` and `35`, i.e. `42`. More details and examples on the environment model can be found in the lecture notes.

Because you can run OCalf programs without type checking them first, your interpreter may come across expressions it can't evaluate (such as `3 + true` or `match 3 with | 1 -> false`). These expressions should evaluate to the special value `VError`.

Exercise 3: Implement eval without LetRec or Match (30 points).

We recommend the following plan for this exercise:

(a) Implement `eval` for the primitive types (`Unit`, `Int`, etc.), `BinOp` (which represents binary operations like `+`, `*` and `^`), `If`, `Var`, `Fun`, `Pair`, and `Variant`.

Note: for OCalf, the comparison operators `=`, `<`, `<=`, `>`, `>=`, and `<>` only operate on integers.

(b) Implement `eval` for `App` and `Let`.

Exercise 4: Implement LetRec (5 points).

Extend `eval` to include the evaluation of `LetRec`.

Evaluating `let rec` expressions is tricky because you need a binding in the environment for the defined before you can evaluate its definition. In lecture we extended the definition of a closure to account for recursive functions; in this assignment we will use another common technique called **backpatching**.

To evaluate the expression `let rec f = e1 in e2` using backpatching, we first evaluate `e1` in an environment where `f` is bound to a dummy value. If the value of `f` is used while evaluating `e1`, it is an error (this would occur if the programmer wrote `let rec x = 3 + x` for example), however it is likely that `e1` will evaluate to a closure that contains a binding for `f` in its environment.

Once we have evalutated `e1` to `v1`, we imperatively update the binding of `f` within `v1` to refer to `v1`. This "ties the knot", allowing `v1` to refer to itself. We then proceed to evaluate `e2` in the environment where `f` is bound to `v1`.

To support backpatching, we have defined the `environment` type to contain `binding ref`s instead of just `binding`s. This enables the bindings to be imperatively updated.

Exercise 5: Implement Match (5 points).

Extend `eval` to support `match` statements. Take care to correctly bind the variables that are defined by the pattern. You may find it helpful to implement a function

```
find_match : pattern -> value -> environment option
```

to check for a match and return the bindings if one is found.

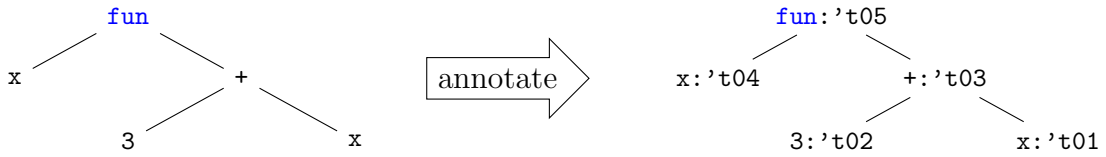
If a given match expression does not match any of the patterns in a match expression, the pattern matching is said to be **inexhaustive**. However, given pattern matchings, you **do not need to check if the pattern matchings are inexhaustive or if there are repetitive match cases.** Your implementation should return `VError` if pattern matching fails at **run-time**.

Part 4: Type inference (50 points)

For the third part of the assignment, you will implement type inference for your interpreted language. OCaml's type inference algorithm proceeds in three steps. First, each node of the AST is **annotated** with a different type variable. Second, the AST is traversed to **collect** a set of equations between types that must hold for the expression to be well-typed. Finally, those equations are solved to find an assignment of types to the original variables (this step is called **unification**).

For example, suppose we were typechecking the expression `(fun x -> 3 + x)` (this example is `infer_example` in the `Examples` module of the provided code). The annotation phase would add a new type variable to each subexpression, yielding

```
print_aexpr (annotate infer_example);;
  (fun (x:'t04) -> ((x : 't02) + (3 : 't01) :'t03) :'t05)
```



Next, we would collect equations from each node. From our typing rules, we know that `(fun x -> e) : t1 -> t2` if `e:t2` under the assumption `x:t1`. Stated differently, `(fun (x:t1) -> e:t2) : t3` if and only if `t3 = t1 -> t2`. Therefore, while collecting constraints from the `fun` node above, we output the constraint `'t05 = 't04 -> 't03`.

Similarly, we must also collect constraints from the `+` node. Our typing rules tell us that `e1 + e2 : int` if and only if `e1 : int` and `e2 : int`. Put another way, `(e1:t1) + (e2:t2) : t3` if and only if `t1 = int`, `t2 = int`, and `t3 = int`. Therefore, while collecting constraints from the `+` node above, we output the three equations `'t03 = int`, `'t02 = int`, and `'t01 = int`.

We would also recursively collect the constraints from the `3` node and the `x` node. The rules for typing `3` say that `3 : t` if and only if `t = int`, so we output the constraint `'t02 = int`. The rules for variables tell us that if `x` had type `t` when it was bound then it has type `t` when it is used. Therefore, when collecting constraints from the `x:'t01` node, we output the equation `'t01 = 't04`.

Putting this together, we have

```
print_eqns (collect [] (annotate infer_example));;
      't01 = int              't03 = int
      't02 = int              't04 = 't01
      't02 = int              't05 = 't04 -> 't03
```

Finally, these equations are solved in the unification step, assigning `int` to `'t01` through `'t04`, and `int->int` to `'t05`. Substituting this back into the original expression, we have

```
print_aexpr (infer [] infer_example);;
  (fun (x:int) -> ((3 : int) + (x : int) :int) : int -> int)
```

If the constraints output by collect are not satisfiable (for example they may require `int = bool` or `'t = 't -> 't`), then it is impossible to give a type to the expression, so a type error is raised.

If the system is underconstrained, then there will be some unsubstituted variables left over after the unified variables are plugged in. In this case, these variables could have any value, so the typechecker gives them user-friendly names like `'a` and `'b`.

Your type inference tasks

We have provided the `annotate` and `unify` phases of type inference for you; your job is to implement the `collect` phase.

Exercise 6: Implement collect without variants (40 points).

We recommend the following plan for implementing `collect`.

1. Implement `collect` for `Fun`, `Plus`, `Int`, and `Var`. This will allow you to typecheck the `infer_example` example as described above.
2. Implement the `collect` function for all syntactic forms **except** `Variant` and `Match`.
3. Implement the `collect` function for `Match` statements, but leave out the handling of `PVariant` patterns. `Match` statements are a bit trickier because you need to make sure the bindings from the patterns are available while checking the bodies of the cases.

Exercise 7: Implement collect with variants (10 points).

Extend `collect` to handle variant types. Variant types are tricky because you need to make use of a type definition to determine how the types of arguments of a constructor relate to the type that the constructor belongs to.

Type definitions are represented by `TypedAst.variant_specs`, and a list of them is provided to the `collect` function. Each variant spec contains a list `vars` of "free variables" (like the `'a` in `'a last`), a type `name` (`"list"` in the case of `'a list`), and a list of constructors (with their types). See `Examples.list_spec` and `Examples.option_spec` for examples.

Deriving the correct constraints from a `variant_spec` requires some subtlety. Consider the following example:

```
(Some (1 : t1) : t2, Some ("where" : t3) : t4)
```

(this is `Examples.infer_variant`). A naive way to typecheck it would be to collect the constraints `'t2 = 'a option` and `'t1 = 'a` from the `Some 1` subexpression, and the constraints `'t4 = 'a option` and `'t3 = 'a` from the `Some "where"` subexpression.

However this would force `'a` to be both `string` and `int`, so the expression would not typecheck. A better approach is to think of the constructor type as a **template** from which

you create constraints. You can correctly generate the constraints by creating a new type variable for each of the free variables of the `variant_spec`, and substituting them appropriately in the constructor's type.

In the above example, you might create the variable `'x` for `'a` while collecting `Some 1`, and output the constraints `'t1 = 'x` and `'t2 = 'x option`, and you might create the variable `'y` for `'a` while collecting `Some "where"`, and output the constraints `'t3 = 'y` and `'t4 = 'y option`. This would allow `infer_variant` to type check correctly.

Exercise 8: [Optional] Implement let-polymorphism (0 points).

Note: This question **is completely optional and will not affect your grade in any way**.

Note 2: This problem may require you to reorganize your code somewhat. Make sure you make good use of source control, helper functions, and unit tests to ensure that you don't break your existing code while working on it!

The type inference algorithm described above allows us to give expressions polymorphic types: `fun x -> x` will be given the type `'a -> 'a`.

However, it does not let us use them polymorphically. Consider the following expression:

```
let (any:t1) = (fun (x:t3) -> (x:t4)):t2 in (any 1, any "where")
```

(this is `Examples.infer_poly`). OCaml will give this expression the type `int * string`, but our naive inference algorithm fails.

The problem is similar to the subtlety with variants above: we will generate the constraints `(typeof any) = (typeof 1) -> t2` and `(typeof any) = (typeof "where") -> t2`, but these constraints can only be solved if `int = typeof 1 = typeof "where"= string`.

The solution is also similar: every time a variable defined by a `let` expression is used, we create a new type variable corresponding to the use. We use the constraints generated while type checking the body of the `let` as a template for a new set of constraints on the new variable. Using the constraints from the bodies of lets as templates is called **let-polymorphism**.

In the above example, while type checking the body of the let expression, we will generate the constraints `t1 = t2`, `t2 = t3->t4`, and `t3 = t4`. While checking the expression `(any:t5) 1`, we can recreate these constraints with new type variables `t1'`, `t2'`, and so on. We will output the constraints `t1'=t2'`, `t2' = t3'->t4'` and `t3'=t4'`. We also will output the constraint `t5=t1'`.

Because the type variables are cloned, they are free to vary independently. Because we have cloned their constraints, they must be used in a manner that is consistent with their definition.

Part 5: Written questions (10 points)

In the file `written.txt` or `written.pdf`, answer the following questions.

Exercise 9: Unification (5 points).

We did not ask you to implement unify, but the algorithm is essentially the same as the one you use to solve algebraic equations. To give you a sense of how it works, solve the following type equations for `'t1`, `'t2`, and `'t3`.

```
't5 = 't4 -> 't7                't1 = 't6 -> 't5
't5 = int -> 't6                't2 = 't7 list
't6 = bool                      't3 = bool * 't4
```

Exercise 10: Factoring (5 points).

The type definitions `Ast.expr` and `TypedAst.annotated_expr` are nearly identical. Moreover, there are a number of simple helper functions that look very similar.

Describe how you might create a single type and helper function that make the definitions of `expr`, `annotated_expr`, `annotate`, `strip`, and `subst_expr` all one-liners.

Be sure to make it possible to create an `expr` without any reference to types (that is, don't just define `expr` as a synonym for `annotated_expr`).

Exercise 11: Feedback (0 points).

Let us know what parts of the assignment you enjoyed, what parts you didn't enjoy, and any other comments that would help us improve the assignment for the future.

What to submit

You should submit the following files on CMS:

- `streams.ml` containing your stream implementations
- `eval.ml` containing your interpreter
- `infer.ml` containing your type checker
- `written.txt` or `written.pdf` containing your responses to the written questions
- `logs.txt` containing your source control logs