1 Introduction

In this lab assignment, we are going to implement the second phase of the compiler: *semantic analysis* and *type checking*. The goal of this phase is to produce an *Intermediate Representation* (IR) from the abstract syntax tree we generated in lab 2. The IR is a simplified form of the program with a much more limited set of operations that correspond more closely to the operations of the machine. Our IR retains the binding structure, and higher-order functions, in the AST. However, the IR will be much easier to analyze and optimize than the AST.

2 Definition of the IR

The IR is defined in the module `Fc_ir_type`. There are kinds of things in the IR: *types*, *atoms*, and *expressions*. Unlike most compilers, the IR is *typed*. This will mean that we can typecheck the IR to help validate our compiler.

2.1 Types

The IR types are a direct translation of the AST types. Here is the human-readable syntax for the types.

t	::=	any | char | int | float
	|	array[t; dim]
	|	$\{l_1 : t_1; \cdots; l_n : t_n\}$
	|	$(t_1, \ldots, t_n) \to t$
	|	v

The *any* type represents an aribtrary type. We'll use is temporarily to give a type to the **return** *e* expression. The *char*, *int*, and *float* types have the normal definitions. The *array[t; dim]* will be used to represent the type of arrays and pointers. The array (or pointer) refers to values of type t. If the *dim* parameter is present, it specifies the dimensions of the array.

The $\{l_1 : t_1; \cdots; l_n : t_n\}$ is the type of a `struct` with fields $l_1, \ldots, l_n$ of types $t_1, \ldots, t_n$. Functions have type $(t_1, \ldots, t_n) \to t$ (the number of arguments n may be 0). A function of the type takes arguments of type $t_1, \ldots, t_N$ and returns a value of type t.

Finally, a type may be a type identifier v, which refers to a `typedef` or `struct` definition.

The OCaml type `ty` represents an IR type.

```
type ty =
   (* TyAny is the result of a "return" statement *)
   TyAny

   (* Base types *)
 | TyChar
 | TyInt
 | TyFloat
 | TyArray of ty * int option

   (* Structures have fields *)
 | TyStruct of (var * ty) list
```

```
   (* Functions are not assignable values *)
 | TyFun of ty list * ty

   (* Type names *)
 | TyId of var
```

2.2 Atoms

In the IR, *atoms* represent *values*, either variables or constants. Strings are not constants, but numbers, variables, and the *nil* pointer are.

a ::= c | i | x | nil[t] | v

The `nil[t]` pointer is typed; it is a pointer to a value of type t.

```
type atom =
   AtomChar of char
 | AtomInt of int
 | AtomFloat of float
 | AtomNil of ty
 | AtomVar of var
```

2.3 Expressions

Most expressions are like very simplified ML `let` definitions. Here is the complete syntax and ML definition for the part we are working with.

type exp =
(* Declarations *)

(*__let type__ $v_1 = t_1$ __and__ ... __and__ $v_n = t_n$ __in__ e *)
LetTypes of types * exp

(* __let fun__ $v_1 : t(vars_1) = e_1$ *)
__and__ $v_2 : t_2(vars_2) = e_2$
$\vdots$
__and__ $v_n : t_n(vars_n) = e_n$
__in__ e
| LetFuns of funs * exp
(**Arithmetic* *)

(*__let__ $v : t = a$ __in__ e *)
| LetAtom of var * ty * atom * exp

(*__let__ $v : t = unop\ a$ __in__ e *)
| LetUnop of var * ty * unop * atom * exp

(*__let__ $v : t = a_1\ binop\ a_2$ __in__ e *)
| LetBinop of var * ty * binop * atom * atom * exp

(**Functions* *)

(*__let__ $v : t = f(args)$ __in__ e *)

```
| LetApply of var * ty * var * atoms * exp

(*f(args) *)
| TailCall of var * atoms

(*return a *)
| Return of atom

(*Allocation *)

(*let string v = <string> in e *)
| LetString of var * string * exp

(*Conditional *)

(*if a1 relop a2 then e1 else e2 *)
| IfThenElse of relop * atom * atom * exp * exp

(*Array/pointer operations *)

(*a1[a2] <- a3; e *)
| SetSubscript of atom * ty * atom * atom * exp

(*let v : t = a1[a2] in e *)
| LetSubscript of var * ty * atom * atom * exp

(*let v : t = &a1[a2] in e *)
| LetAddrOfSubscript of var * ty * atom * atom * exp

(*Struct projection *)

(*a1.v <- a2; e *)
| SetProject of atom * var * ty * atom * exp

(*let v1 : t = a.v2 in e *)
| LetProject of var * ty * atom * var * exp

(*let v1 : t = &a.v2 in e *)
| LetAddrOfProject of var * ty * atom * var * exp

(*IR1 only: Variable operations (not functional) *)

(*let v : t = copy(a) in e *)
| LetCopy of var * ty * atom option * exp

(*v <- a; e *)
| SetVar of var * ty * atom * exp

(*let v1 : t = &v2 in e *)
| LetAddrOfVar of var * ty * var * exp

(*IR2 only: ignore these for now *)
| Memcpy of atom * ty * atom * exp
| LetAlloc of var * ty * exp
| LetClosure of var * ty * var * atom list * exp
```

2.3.1 Type definitions

The **let type** $v_1 = t_1$ **and** ... **and** $vn = tn$ **in** e expressions defines types $v_1, \ldots, v_n$ in expression e. Types are defined at the source level with a `typedef`, or a `struct` definition.

2.3.2 Simple expressions

The **let** $v : t = a$ **in** e expression binds to the value of the atom a in expression e.

The **let** $v : t = unop\, a$ **in** e expression performs a unary operation. The valid unary operators are arithmetic and logical negation, as well as a few explicit coercions.

```
type unop =
   UMinusOp     (* Arithmetic negation on int or float *)
 | UNotOp       (* Logical negation on int *)
 | UToChar      (* char coercion: from char, int, or float *)
 | UToInt       (* int coercion: from char, int, or float *)
 | UToFloat     (* float coercion: from char, int, or float *)
```

The **let** $v : t = a_1\ binop\ a_2$ **in** e performs a binary operation. In most case, both operands must have the same type (you can't add an `int` and a `float` for example).

```
type binop =
   PlusOp       (* Addition: int/int, float/float, pointer/int *)
 | MinusOp      (* Subtraction: int/int, float/float, pointer/int, pointer/pointer *)
 | MulOp        (* Multiplication: int/int, float/float *)
 | DivOp        (* Division: int/int, float/float *)
 | RemOp        (* Remainder: all the rest are int/int only *)
 | BAndOp       (* Bitwise-and *)
 | BOrOp        (* Bitwise-or *)
 | LslOp        (* Logical shift left *)
 | LsrOp        (* Logical shift right *)
 | AsrOp        (* Arithmetic shift right *)
 | XorOp        (* Bitwise xor *)
```

Strings are not atoms, they require an explicit string allocation **let string** v = `<string>` **in** e. The variable v has type `TyArray (TyChar, None)`.

2.3.3 Functions

The **let fun** expression defines mutually-recursive functions.

$$
\begin{array}{ll}
\textbf{let fun} & v_1 : t_1(vars_1) = e_1 \\
\textbf{and} & v_2 : t_2(vars_2) = e_2 \\
\vdots & \\
\textbf{and} & v_n : t_n(vars_n) = e_n \\
\textbf{in} & e
\end{array}
$$

Function v_i must have type t_i. The $vars_i$ is the list of formal parameters for the function, and e_i is the body of the function. The functions are bound in the bodies e_i as well as the rest of the program e.

Functions can be called in two ways. In the **let** $v : t = f(args)$ **in** e expression, variable v is bound to the result of calling f on arguments *args*, in the expression e.

A *tail-call* has the form $f(args)$: the function f is called on arguments *args* and the value is returned as the result of the current expression.

The **return** a can be used to return a value from a function. Note that, by construction, the body of a function must either end in a tail-call or a **return**.

Note that function declarations have a `bool` flag. This flag is `true` iff the function is a "user-defined" or "global" function. There is a more precise definition: a "global" function can be callsed only with `LetApply`; a "local" function can only be called in a `TailCall`. The **return** expression returns the value to the nearest "global" function.

2.3.4 Conditional

The conditional **if** a_1 *relop* a_2 **then** e_1 **else** e_2 requires a comparison. Both operands a_1 and a_2 must have the same type, either `char`, `int`, `float`, or else they must be `pointers` to values of the same type. The *relop* includes the standard comparisons.

```
type relop =
   EqOp    (* == *)
 | NEqOp   (* != *)
 | LeOp    (* <= *)
 | LtOp    (* < *)
 | GtOp    (* > *)
 | GeOp    (* >= *)
```

Note that the result of the `if` can't be assigned in a `let` definition.

2.3.5 Array and struct operations

There are the standard operations on arrays. The dereference operation is

$$\textbf{let } v : t = a_1[a_2] \textbf{ in } e.$$

The address of an element of the array can be taken with the addressing operation.

$$\textbf{let } v : t = \&a_1[a_2] \textbf{ in } e.$$

An element can be assigned with the `SetSubscript` operation.

$$a_1.v < -a_2; e.$$

Structs are the same as arrays, but the elements are labeled with variables.

3 Your task

Your task for this lab is to produce the IR from the AST. There is template code in the `Fc_ir_ast` module. You are to implement the `make_expr` function. We'll talk about *how* to implement this function in class. Here is a brieft description of some of the files you will need to use.

3.1 The `Symbol` module

You will need to generate lots of new variables for temporary values. The symbol functions are defined in the `Symbol` module.

```
val new_symbol : symbol -> symbol
val new_symbol_string : string -> symbol
```

These functions generate *new* symbols that are guaranteed to be unique, so you can use them without fear of shadowing other variables that have been bound.

As we build the compiler, it will often be useful to rename *all* the variables in a program so that they are different. This is called "standardize apart" (AI terminology). The `Fc_ir_standardize` module contains a function to do just that.

```
val standardize_expr : exp -> exp
```

3.2 Exceptions

It is important to generate sensible error messages in your compiler. The `Fc_ir_exn_type` module defines some useful exceptions. A `sem_error` represents the kind of error that occurred; you may add to this list if you want.

```
type sem_error =
   UnboundVar of var
 | UnboundType of var
 | UnboundLabel of var
 | TypeError of ty * ty
 | TypeError1 of ty
 | BinopTypeError of ty * ty
 | NotAConstant of Fc_ast_type.expr
 | NotAddressible of Fc_ast_type.expr
 | NotAPointer of ty
 | NotAScalar of ty
 | NotAStruct of ty
 | NotAFunction of ty
 | NotAStructPointer of ty
 | NotAPrefixValue of ty
 | CantInitialize of ty
 | ArityMismatch of int * int
 | IRLevel of int
 | InternalError of string
 | SubscriptOutOfBounds of int
 | NotImplemented
```

It is also extremely important to print out sensible position information when an error happens. The position type `'a exn_loc` allows you to add some debugging information to the position that is printed out on an error message.

```
type 'a exn_loc =
   ExnAtom of 'a
 | ExnString of string * 'a exn_loc
 | ExnInt of int * 'a exn_loc
```

The 'a type parameter represents the position in the input file. The ExnString and ExnInt functions allow you to add a extra information.

The actual exception has two forms. In this stage, you will be using the SemException.

```
exception SemException of Fc_ast_type.pos exn_loc * sem_error
```

That is, the SemException exception takes an annotated AST position, and an error. Normally, you manipulate the position information with the _pos functions in the Fc_ir_exn module.

```
val atom_pos : 'a -> 'a exn_loc
val int_pos : int -> 'a exn_loc -> 'a exn_loc
val string_pos : string -> 'a exn_loc -> 'a exn_loc
```

Don't worry about the extra positional information too much; it is just there to help you debug your compiler. Normally, you would use string_pos to add a comment to a position when a function is entered. These comments may be printed by using the -debug_pos flag on the compiler. Here's an example, where the Fc_ir_ast.make_expr, make_apply_expr, and coerce_fun functions use string_pos to add their name to the position.

```
<tehachapi 1013>cat test.c
int fibx(int i)
{
    if(i == 0 || i == 1)
        1;
    else
        fib(i - 1) + fib(i - 2);
}
<tehachapi 1014>./fcc test.c
test.c:6:chars 8-18:unbound variable: fib
Exit 1
<tehachapi 1015>./fcc -debug_pos test.c
test.c:6:chars 8-18:
   /Fc_ir_ast
   /make_apply_expr
   /coerce_fun:
   unbound variable: fib
Exit 1
```

3.3 Type and variable environments

As you build the IR, you will need to keep track of the typedefs and the types of variables in the program. The Fc_ir_env module contains definitions of these environments.

```
(*
 * Environments are abstract.
 *)
type tenv
type venv

(*
 * Type environment operations.
 *)
val empty_tenv : tenv
```

```
val tenv_add : tenv -> var -> ty -> tenv
val tenv_lookup : 'a raise_exn -> tenv -> 'a -> var -> ty
val tenv_expand : 'a raise_exn -> tenv -> 'a -> ty -> ty

(*
 * Variable environment operations.
 *)
val empty_venv : venv
val venv_add : venv -> var -> ty -> venv
val venv_lookup : 'a raise_exn -> venv -> 'a -> var -> ty
val venv_lookup_expand : 'a raise_exn -> tenv -> venv -> 'a -> var -> ty
```

An environment is essentially a functional table. If you add a `var` and it's type `ty` to a variable environment with the `venv_add` function, the `venv_lookup` function can be used to get the type of the variable later. The type environment works the same way.

The `raise_exn` argument say what kind of exception to raise if the variable is not found. Here are some standard definitions to return a normal exception.

```
let raise_exn pos v =
   SemException (pos, v)

let tenv_lookup = tenv_lookup raise_exn  (* tenv -> pos -> var -> ty *)
let tenv_expand = tenv_expand raise_exn  (* venv -> pos -> ty -> ty *)
let venv_lookup = venv_lookup raise_exn  (* venv -> pos -> var -> ty *)
let venv_lookup_expand = venv_lookup_expand raise_exn
```

3.4 Typechecking IR

The IR you generate should typecheck. The `Fc_ir_check` and `Fc_ir_typeof` modules define type checkers. The `check_expr` function returns the type of an expression, if the expression type checks. Otherwise, it raises an exception. You will find this function to be extremely useful; you can turn on type checking by using the `-check_ir` function to `fcc`. Don't use it to perform type checking on the AST; you should do that yourself when you generate the IR. In a correct compiler, the `check_expr` function would never fail.

```
val check_expr : tenv -> venv -> exp -> ty
```

4 What to turn in

You should turn in your entire compiler in your `submit/lab3` directory on `mojave` (you should probably keep a copy in your CS account too). IF you are working in a group, only one of you should do the submission, and you should send mail to `cs134-admin` to tell us who the group is. If you had to modify anything in the `mllib` directory, include your changes.

In addition, you should include the following.

- A `README` file explaining what you did, how it works, and whether you had any problems.
- A `DIFF` file generated using the command `cvs diff` in the `fc_ir` directory. If you like, you can insert this into the `README` file, with brief explanations of what you changed.
- The files `test1.ir`, `test2.ir`, and `test3.ir` generated using your compiler with the `-print_ir` option.

```
% ./fcc -check_ir -print_ir test1.c > test1.ir
% ./fcc -check_ir -print_ir test2.c > test2.ir
% ./fcc -check_ir -print_ir test3.c > test3.ir
```

- Program output of the test runs. Use the following commands.

```
% ./fcc -eval test1.c -- test1 10 > test1.out
% ./fcc -eval test2.c -- test2 > test2.out
% ./fcc -eval test3.c -- test3 > test3.out
```