Computer Science 131, Spring 2001
Assignment 8: Implementing Subtyping

Out: Wednesday, March 28, 2001

Due: Wednesday, April 4, 11:00am

To complete this assignment, retrieve the files `setup8.sml`, `assign8.sml` and `assign8-tests.sml`. Instead of running a special version of the SML/NJ compiler, start by loading the file `setup8.sml`. (You should not need to ever look at this file.)

Your task is to complete the functions in the `assign8.sml` file and turn this in. As always, your file should contain no syntax or type errors. The submission process is the same as before (`cs131submit assign8.sml`).

For this assignment we will add floating-point values to the NQSML language, as shown in Figure reffig:absyn.

The datatype in Figure 2 should be familiar; the changes from assignment 5 have all been marked. You can also look at some of the pre-defined expressions in the `assign8-tests.sml` file to see their abstract syntax and the SML representations of this abstract syntax. (These types are pre-defined for you in `setup8.sml`. You will *not* need to type this in as part of your `assign8.sml` file.)

The `setup8.sml` file includes the type `'a env` (environments associating values of type `'a` with variable names) and the following items:

```
empty  : 'a env
extend : 'a env * varname * 'a -> 'a env
lookup : 'a env * varname -> 'a
```

The setup file also supplies functions

```
ppty  : ty -> unit
ppexp : exp -> unit
pptyenv : ty env -> unit
```

for printing types, expressions, and type environments respectively.

$v ::=$	n	(integer constants)
	c	(floating-point constants)
	`tt`	
	`ff`	
	`fix` f(`(`x`:`t_1`)`)`:`t_2 `is` e_2	
	$\langle v_1, v_2 \rangle$	
	`nil[`t`]`	
	v_1 `::` v_2	
$e ::=$	v	
	x	
	e_1 `+` e_2	(integer addition)
	e_1 `F+` e_2	(floating-point addition)
	e_1 `<` e_2	(integer comparison)
	e_1 `F<` e_2	(floating-point comparison)
	`toFloat(`e`)`	(integer-to-floating-point conversion)
	`if` e_1 `then` e_2 `else` e_3	
	`let` x `be` e_1 `in` e_2	
	e_1 e_2	
	$\langle e_1, e_2 \rangle$	
	`fst` e	
	`snd` e	
	e_1 `::` e_2	
	`case` e_1 `of nil =>` e_2 `|` x`::`y `=>` e_3	
$t ::=$	`Int`	
	`Float`	
	`Bool`	
	t_1`->`t_2	
	t_1`*`t_2	
	t `list`	

Figure 1: Abstract Syntax for This Assignment

```
type varname = string
datatype ty  = Int_t
             | Float_t                                          (* ! *)
             | Bool_t
             | Arrow_t of ty * ty
             | Times_t of ty * ty
             | List_t of ty
datatype aop = Plus | Minus | Times | FPlus | FMinus | FTimes  (* ! *)
datatype cop = Less | Equal | Fless | FEqual                   (* ! *)
datatype exp = Num of int
             | Float of real                                   (* ! *)
             | Bool of bool
             | Var of varname
             | Arith of exp * aop * exp
             | Compare of exp * cop * exp
             | ToFloat of exp                                  (* ! *)
             | If of exp * exp * exp
             | Let of varname * exp * exp
             | Fix of varname * varname * ty * ty * exp
             | Apply of exp * exp
             | Pair of exp * exp
             | Fst of exp
             | Snd of exp
             | Nil of ty
             | Cons of exp * exp
             | Listcase of exp * exp * (varname * varname * exp)
```

Figure 2: Datatype Representation of Programs

1 The Subtype Relation

The subtype relation for this language can be defined by the following collection of inference rules:

$$\frac{}{\texttt{Bool} \preceq \texttt{Bool}} \tag{1}$$

$$\frac{}{\texttt{Int} \preceq \texttt{Int}} \tag{2}$$

$$\frac{}{\texttt{Int} \preceq \texttt{Float}} \tag{3}$$

$$\frac{}{\texttt{Float} \preceq \texttt{Float}} \tag{4}$$

$$\frac{t_1 \preceq u_1 \qquad t_2 \preceq u_2}{t_1 \texttt{*} t_2 \preceq u_1 \texttt{*} u_2} \tag{5}$$

$$\frac{u_1 \preceq t_1 \qquad t_2 \preceq u_2}{t_1 \texttt{->} t_2 \preceq u_1 \texttt{->} u_2} \tag{6}$$

$$\frac{t \preceq u}{t\ \texttt{list} \preceq u\ \texttt{list}} \tag{7}$$

Write the function

```
subtype : ty * ty -> bool
```

which determines whether the first type is a subtype of the second. Your code should have one case for each inference rule.

Side Note: this presentation of the subtyping language has been carefully constructed to be reflexive and transitive, without explicitly having inference rules for reflexivity or transitivity. This means that for given types t_1 and t_2, there is at most one proof that t_1 is a subtype of t_2. This makes checking for subtypes pretty trivial. Annoyingly, for certain completely reasonable type systems, one can show there exists *no* algorithm for correctly determining whether one type is a subtype of another. For example, there is a subtyping-like relation between signatures in the SML module system. If SML modules were ordinary values (so that signatures became a kind of type, and one could have pairs of modules, etc.) then subtyping — and typechecking — become provably undecidable.

2 Meets and Joins

In any set with an ordering, one can ask whether a pair of types has a least upper bound (join) or a greatest lower bound (meet). The least upper bound of two types t_1 and t_2 (written $t_1 \sqcup t_2$ when it exists) must satisfy the following two conditions:

1. $t_1 \preceq t_1 \sqcup t_2$ and $t_2 \preceq t_1 \sqcup t_2$
2. For every type u, if $t_1 \preceq u$ and $t_2 \preceq u$ then $t_1 \sqcup t_2 \preceq u$.

That is, $t_1 \sqcup t_2$ is an upper bound for both t_1 and t_2, and among all such upper bounds it is minimal.

The greatest lower bound $t_1 \sqcap t_2$ of two types similarly satisfies

1. $t_1 \sqcap t_2 \preceq t_1$ and $t_1 \sqcap t_2 \preceq t_2$
2. For every type u, if $u \preceq t_1$ and $u \preceq t_2$ then $u \preceq t_1 \sqcap t_2$.

For example, in this type system

- `Int` $\sqcup$ `Float` = `Float`
- `Int` $\sqcap$ `Float` = `Int`
- `Int->Int` $\sqcup$ `Float->Float` = `Int->Float`
- `Int->Int` $\sqcap$ `Float->Float` = `Float->Int`
- `Int` and `Float->Int` have no meet or join.

Define the functions

```
meet : ty * ty -> ty option
join : ty * ty -> ty option
```

which return the meet and join of the given types if they exist, and which return `NONE` otherwise.

3 Typechecking with Subtyping

The simplest way to extend the typing rules for the extended language would be to add rules

$$\frac{}{\Gamma \vdash c : \texttt{Float}} \tag{8}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Float} \qquad \Gamma \vdash e_2 : \texttt{Float}}{\Gamma \vdash e_1 \,\texttt{F+}\, e_2 : \texttt{Float}} \tag{9}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Float} \qquad \Gamma \vdash e_2 : \texttt{Float}}{\Gamma \vdash e_1 \,\texttt{F<}\, e_2 : \texttt{Bool}} \tag{10}$$

(with similar rules for floating-point multiplication and subtraction, and floating-point equality)

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{toFloat(}e\texttt{)} : \texttt{Float}} \tag{11}$$

plus the subsumption rule

$$\frac{\Gamma \vdash e : t \qquad t \preceq u}{\Gamma \vdash e : u} \tag{12}$$

Unfortunately, it is not immediately obvious how to do typechecking because the rules are no longer syntax-directed; expressions no longer have unique types (or unique proofs of well-typedness.)

Fortunately, this system has the property that every expression has a most-precise (minimal with respect to subtyping) type, called its *principal type*. (Languages without principal types are really hard to typecheck.) Then:

- It follows that $e : t$ if and only if the principal type of e is a subtype of t.
- Although the integer `3` can be shown to have both the type `Int` and the type `Float`, its principal type is uniquely `Int`.
- The principal type of e_1 `F+` e_2 is always `Float` (though when typechecking, one must verify that the principal types of e_1 and e_2 are *subtypes* of `Float`).
- The principal type of $\langle e_1, e_2 \rangle$ is t_1`*`t_2 where t_1 is the principal type of e_1 and t_2 is the principal type of e_2.
- The principal type of `if` e_1 `then` e_2 `else` e_3 is $t_2 \sqcup t_3$ where t_2 and t_3 are the principal types of t_1 and t_2 respectively. (Of course when typechecking one still must check that the principal type of e_1 is a subtype of `Bool`.)
- What is the principal type of the application $e_1\, e_2$? Well, if the principal type of e_1 is t`->`u and the principal type of e_2 is t_2, then the principal type of the application is u. To typecheck the expression, however, one must check that $t_2 \preceq t$ (that is, checking whether e_2 is an acceptable argument) instead of checking that $t_2 = t$.
- In general, every place where you previously used equality between types in the absence of subtyping, you will now either use subtyping or check whether the two types have a join, as appropriate.

Write a function

```
principal : ty env * exp -> ty
```

that computes the principal type of the given expression, and raises an exception otherwise. You may want to work by modifying your `typeof` function from Assignment 5.

4 "Written" portion

Put the answers to the following questions in a comment at the end of your `assign8.sml` file. *Do not hand them in separately.*

For the first 4 parts, give the most general conditions on t_1 and t_2 for the given subtyping judgments to be safe in an ML-like language: $t_1 \preceq t_2$ or $t_1 \succeq t_2$ or $t_1 = t_2$. Give a brief and clear explanation.

1. t_1 `foo` $\preceq t_2$ `foo`, defined by `type 'a foo = 'a list * 'a list`.
2. t_1 `bar` $\preceq t_2$ `bar`, defined by `type 'a bar = 'a -> 'a`
3. t_1 `baz` $\preceq t_2$ `baz`, defined by `type 'a baz = ('a -> int) -> int`. (Careful.)
4. t_1 `cont` $\preceq t_2$ `cont`. (What operations are there specific to continuations?)
5. If $t_1 \preceq t_2$ then we know that neither t_1 `ref` nor t_2 `ref` is a subtype of the other. However, is it safe to assign a value of type t_1 to a t_2 `ref` or a value of type t_2 to a t_1 `ref`? Again, give a brief explanation of your answer.