What to turn in

Turn in the written parts of the assignment during class on the due date. For the programming part, you should mail your version of the file cps-translate.sml to `schong@cs.cornell.edu` by 5PM on that day.

Problem 1: Semantics of Exceptions

Consider extending the language F with exceptions and call this language F_{exc}:

$$e ::= \ldots \mid \textsf{throw}\ s\ e_v \mid \textsf{try}\ e_v\ \textsf{catch}\ (s\ x)\ e_h$$

There are many ways of writing the semantics of exceptions. In class we provided a denotational semantics of exceptions using Continuation Passing Style. In this problem you will explore alternative approaches for expressing the semantics of such constructs, including an operational semantics and a different denotational semantics for F_{exc}.

Informally, the expression $\textsf{try}\ e_v\ \textsf{catch}\ (s\ x)\ e_h$ returns the evaluation of e_v, unless e_v evaluates to $\textsf{throw}\ s\ v$. In that case, it returns the evaluation of the exception handler e_h with x bound to v. Note that v should be a value, and that v should not be a throw expression. The expression $\textsf{throw}\ s\ e_v$ first attempts to evaluate e_v to a value. If an exception $\textsf{throw}\ s'\ v'$ is raised during the evaluation of e_v, then that exception "takes priority," and replaces the original exception.

Denotational Semantics

Instead of using continuations, we will use sum domains to denote values of F_{exc}. More precisely, if an expression evaluates to an exception $\textsf{throw}\ s\ v$, then it will produce a pair, consisting of a symbol s, denoting the exception handler, and the value v, to be passed as an argument to the exception handler. Otherwise it will be a single value, such as an integer, a boolean, a pair, or a function, as appropriate.

(a) Write the domain equations for the denotational semantics of F_{exc} and then write the signature of the denotation function $\mathcal{C}[\![e]\!]$. You must provide the equations that define the domains *Value*, *Function*, *Env*, etc. You may assume the existence of a domain *Sym*, which will be the domain of the denotations of the exception symbols s. You may find it helpful to define a domain *Exception*.

(b) Write the denotations of the following constructs of F_{exc}. Make sure your denotations agree with the domain equations in part (a).

$\textsf{true}$	$e_0\ e_1$	$\langle e_0, e_1\rangle$	$\textsf{throw}\ s\ e_v$
x	$\lambda x.\, e$	$\textsf{let}\ x\ = e_0\ \textsf{in}\ e_1$	$\textsf{try}\ e_v\ \textsf{catch}\ (s\ x)\ e_h$

Operational Semantics

(c) Finally, we want to write a large step operational semantics for F_{exc}, but without using pairs. In other words, we require that the evaluation of throw constructs should not yield pairs of values. Write the large-step operational semantics for the following constructs:

$\textsf{throw}\ s\ e$	$e_0\ e_1$	$\textsf{let}\ x\ = e_0\ \textsf{in}\ e_1$
$\textsf{try}\ e_v\ \textsf{catch}\ (s\ x)\ e_h$	$\langle e_0, e_1\rangle$	

$$\frac{\langle e_0, \sigma\rangle \to \langle e_0', \sigma\rangle}{\langle e_0 \ op\ e_1, \sigma\rangle \to \langle e_0' \ op\ e_1, \sigma\rangle} \qquad \frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma\rangle}{\langle v \ op\ e_1, \sigma\rangle \to \langle v \ op\ e_1', \sigma\rangle}$$

$$\frac{}{\langle v_0 \ op\ v_1, \sigma\rangle \to \langle v, \sigma\rangle}\ v = v_0 \ op\ v_1 \qquad \frac{}{\langle x, \sigma\rangle \to \langle \sigma(x), \sigma\rangle}$$

$$\frac{\langle e_0, \sigma\rangle \to \langle e_0', \sigma'\rangle}{\langle e_0; e_1, \sigma\rangle \to \langle e_0'; e_1, \sigma'\rangle} \qquad \frac{}{\langle \mathbf{skip}; e_1, \sigma\rangle \to \langle e_1, \sigma\rangle}$$

$$\frac{\langle e_0, \sigma\rangle \to \langle e_0', \sigma'\rangle}{\langle x := e_0, \sigma\rangle \to \langle x := e_0', \sigma'\rangle} \qquad \frac{}{\langle x := v, \sigma\rangle \to \langle \mathbf{skip}, \sigma[x \to v]\rangle}$$

$$\frac{\langle e_0, \sigma\rangle \to \langle e_0', \sigma'\rangle}{\langle \mathbf{if}\, e_0 \,\mathbf{then}\, e_1 \,\mathbf{else}\, e_2, \sigma\rangle \to \langle \mathbf{if}\, e_0' \,\mathbf{then}\, e_1 \,\mathbf{else}\, e_2, \sigma'\rangle}$$

$$\frac{}{\langle \mathbf{if\ true\ then}\, e_1 \,\mathbf{else}\, e_2, \sigma\rangle \to \langle e_1, \sigma\rangle} \qquad \frac{}{\langle \mathbf{if\ false\ then}\, e_1 \,\mathbf{else}\, e_2, \sigma\rangle \to \langle e_2, \sigma\rangle}$$

$$\frac{}{\langle \mathbf{while}\, e_0 \,\mathbf{do}\, e_1, \sigma\rangle \to \langle \mathbf{if}\, e_0 \,\mathbf{then}\, e_1; \mathbf{while}\, e_0 \,\mathbf{do}\, e_1 \,\mathbf{else\ skip}, \sigma\rangle}$$

Figure 1: Operational Semantics for Problem 2(a)

Problem 2: Typing Imperative Constructs and Aliases

In this problem we will construct a type system for a language with imperative features, pointers, and explicit types that capture aliasing relations.

(a) We start with a base language similar to **IMP**. In contrast to **IMP**, we don't have different syntactic categories for commands, integers, and booleans. We represent all of them as expressions and leave the job of distinguishing between them to the type-checker. The syntax of expressions, types, and values in our base language is:

$$\begin{array}{rcl} e & ::= & n \mid x \mid \mathbf{true} \mid \mathbf{false} \mid e_0 + e_1 \mid e_0 < e_1 \mid e_0 \wedge e_1 \\ & & \mid \mathbf{skip} \mid e_0; e_1 \mid x := e \mid \mathbf{if}\, e_0 \,\mathbf{then}\, e_1 \,\mathbf{else}\, e_2 \mid \mathbf{while}\, e_0 \,\mathbf{do}\, e_1 \\ \tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \\ v & ::= & n \mid \mathbf{true} \mid \mathbf{false} \end{array}$$

Figure 1 presents the axioms in the small-step operational semantics for this language. The semantics consist of evaluations of the form $\langle e, \sigma\rangle \to \langle e', \sigma'\rangle$.

We would like to write down typing rules that correspond to this semantics, i.e. such that well-typed expressions don't get in stuck configurations. Here are some of the typing rules:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma[x \to \tau] \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{int} \quad \Gamma \vdash e_1 : \mathsf{int}}{\Gamma \vdash e_0 + e_1 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_0 : \mathsf{int} \quad \Gamma \vdash e_1 : \mathsf{int}}{\Gamma \vdash e_0 < e_1 : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_0 : \mathsf{bool} \quad \Gamma \vdash e_1 : \mathsf{bool}}{\Gamma \vdash e_0 \wedge e_1 : \mathsf{bool}}$$

Write the typing rules for skip, sequencing, assignments, conditionals, and while loops.

(b) Since states are mappings in *Var* $\to$ *Value*, and typing contexts are mappings in *Var* $\to$ *Type*, we must define a correspondence between typing contexts and states which agree with those contexts. We say that a state σ satisfies a typing context Γ, and write $\sigma \models \Gamma$, if:

$$\begin{array}{ll} \forall x \in \mathrm{dom}(\Gamma): & (\Gamma(x) = \mathsf{int} \quad\; \Rightarrow \sigma(x) \in \mathbb{Z}) \;\; \wedge \\ & (\Gamma(x) = \mathsf{bool} \Rightarrow \sigma(x) \in \{\mathbf{true}, \mathbf{false}\}) \wedge \\ & (\Gamma(x) \neq \mathsf{unit}) \end{array}$$

Using this definition, formally write and prove the Preservation Lemma.

(c) So far the language doesn't allow us to assign types to variables. We therefore extend the language with explicit type declarations for variables:

$$e \;\;::=\;\; \ldots \;\mid\; x\!:\!\tau$$

Informally, we want to allow variables to be used only after they have been declared, and we want each variable to be declared at most once. We allow declarations only for integer and boolean types. Finally, we require that integer variables are initialized to 0 and booleans are initialized to **false** at such declarations.

Write the corresponding small-step operational semantics and the new typing rules that support type declarations.

(d) We next add pointers and pointer operations in the language. We use a syntax similar to the C language consisting of the address-of operator & and the dereference operator $*$. But we also allow programmers to express *extended pointer types*, where they can declare explicitly the set of possible target variables. For instance, a declaration $x\!:\!\mathsf{ptr}\ \{y, z\}\ \mathsf{int}$ declares a pointer to an integer which may point to y or z, but to no other variable. This declaration must agree with the declarations for y and z, i.e. y and z must be declared as integers. The new syntax is:

$$\begin{array}{lcl} e & ::= & \ldots \;\mid\; *\,e \;\mid\; \&x \;\mid\; *\,x := e \\ \tau & ::= & \ldots \;\mid\; \mathsf{ptr}\ S\ \tau \\ S & ::= & \emptyset \;\mid\; S \cup \{x\} \\ v & ::= & \ldots \;\mid\; \&x \end{array}$$

For instance, the following piece of code will not type-check, because it will make `u` point to `z`, which is not allowed by the type of `u`:

```
x : ptr{u} ptr{y} int;
u : ptr{y} int;
y : int;
z : int;
 x := &u; /* okay */
*x := &y; /* okay */
*x := &z; /* type error */
```

Note that we allow variables to occur in type declarations before they are themselves declared. The occurrences of variables in pointer type declarations are not regarded as uses of those variables.

Define a set of typing rules which ensure that pointer variables do not point to any variables other than the declared ones. You may find helpful to define a subtype-like relation between extended pointer types, since a pointers whose types have smaller target sets can be used whenever pointer with larger target sets can be used.

(e) The types defined in part (d) don't allow us to type recursive declarations such as $x : \mathsf{ptr}\ \{x\}\ \mathsf{int}$. We therefore add recursive types and explicit fold and unfold operators:

$$\begin{array}{lcl} e & ::= & \ldots \;\mid\; \mathsf{fold}\ [\mu T\ .\ \tau]\ e \;\mid\; \mathsf{unfold}\ e \\ \tau & :: & \ldots \;\mid\; \mu T\ .\ \tau \;\mid\; T \end{array}$$

Consider the following statements:

```
 x := &y;
*x := &x;
 x := **x;
 y := **y;
```

Write the type declarations for `x` and `y`. Then insert explicit fold and unfold operators in the above assignments to make them well-typed.

Problem 3: Closure Conversion + CPS = Assembly Code

In this problem you will translate a simple call–by–value language into a target language that is very close to assembly language. Hence, you will be exploring some of the transformations needed for compilation to assembly.

The key feature of the target language is that lambda terms in the language cannot have any free variables other than the argument variable itself. This makes sense because in a low-level assembly language one can only refer to local registers, not to variables from containing lexical scopes. The process of removing free variables from lambda terms is known as *closure conversion* or *lambda lifting* (because once your lambda terms have no free variables, they can be lifted to the top level of the program).

The source language here is a strict, call–by–value λ-calculus language. To simplify writing examples, it also adds built–in boolean constants and an if expression. Its syntax is:

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \\ e & ::= & x \mid n \mid \#t \mid \#f \mid \textsf{iszero}\ e \mid \textsf{if}\ e_0\ e_1\ e_2 \mid \textsf{fn}\ x\ e \mid e_1\ e_2 \mid op\ e_1\ e_2 \end{array}$$

The operators $+$, $*$, and $-$ are strictly binary, and iszero is unary, producing one of $\#t$ or $\#f$. We have provided a parser for this language, the function LambdaSimpleParser.parse, which should make it easy to write test cases for your translation and turn them into LambdaSimpleAST.exp's.

The syntax of the target language is:

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \\ e & ::= & x \mid \lambda x.\, s (\text{where } FV[\![s]\!] \subseteq x) \mid n \mid x_1\ op\ x_2 \mid \textsf{halt} \mid \langle x_1, ..., x_n \rangle \mid x[n] \\ s & ::= & x_1\ x_2 \mid \textsf{let}\ x = e\ \textsf{in}\ s \mid \textsf{ifz}\ x\ s_1\ s_2 \end{array}$$

Here, $\langle x_1, ..., x_n \rangle$ represents a "tuple" of n elements; if $y = \langle x_1, ..., x_n \rangle$ then $y[i] \rightarrow x_i$. The application of x to halt (e.g. let $h =$ halt in $h\ x$) terminates execution, returning the value bound to x. Applications are restricted to be of one variable to another variable, and abstractions $\lambda x.\, s$ are restricted to terms s with at most one free variable, necessarily x. The ifz statement tests whether the variable x is zero and executes s_1 if so, s_2 otherwise. Note that the target language does not have boolean values. We suggest that you represent source–language booleans as integers; for example, translate $\#t$ as 0 and $\#f$ as 1. This will simplify translation of iszero! However, you can choose another representation if you like.

Note that the statements s in the target language are essentially continuations. They contain the "rest of the program" to execute. Therefore we can refer to the target language as a CPS language. For example, the expression $(+\ (*\ 4\ 3)\ 5)$ would need to be translated into the target language as something like:

let $x_1 = 4 * 3$ in
let $x_2 = x_1 + 5$ in
s

where x_1 and x_2 are new temporary variables, and s is the rest of the computation that needs to happen after the expression $(+\ (*\ 4\ 3)\ 5)$ has been evaluated.

Note that there is a strong correspondence between the expressions e and statements s of the CPS language and the instructions of an ordinary machine language, as shown by the following example[1]:

let $x_1 = x_2 + x_3$ in
let $f = \lambda y.\, y * y$ in
$f\ x_1$

```
F:  POP Y            ADD X1 X2 X3
    MUL Y Y Y        PUSH X1
    PUSH Y           CALL F
    RET
```

(a) Define the translation function $\mathcal{T}[\![e]\!]$, where e is an expression of the source language, and the result is a statement of the target language.

The main hurdle to overcome is that whereas the source language has no restrictions on non-local variables, the target language only allows function bodies to have one free variable. You will need to find some way of packing a function's real argument, its return continuation, and any other necessary information into the single argument to the CPS function. In addition, note that source–language function values capture the values of their free variables (lexically), whereas target–language function values cannot. Therefore, you will need to translate source–language function values to something other than a target–language function value!

You may choose to perform both CPS and closure conversion in one pass (i.e. your translation function does both at the same time), or separate them into two passes (i.e. your translation function does one conversion first, takes the result, and performs the other conversion on the result).

(b) Implement your translation function(s) from part (a). You are allowed to modify only the file "cps-translate.sml". To see some of the parsing and translation functions being used, look in "top.sml" for some examples. If you chose to perform CPS and closure conversion in two separate passes, note that the evaluator does not know how to handle lambda terms that have free variables.

Your translation should be a function cps_translate : LambdaSimpleAst.exp $\rightarrow$ CPS.stm. Given a source language expression which evaluates to a base value n, evaluating the result of your CPS translator using CPS.evaluate should also result in n. The file "cps-ast.sml" contains a couple of functions that you may find useful: newvar and eqvar.

Problem 4: Widening Operators

(a) Consider a complete lattice $(L, \sqsubseteq, \sqcup, \bot, \top)$. Show that all of the following are upper bound operators. Determine which of them are also widening operators and explain your answer.

$$\lambda(l_1, l_2).\, \top$$

$$\lambda(l_1, l_2).\, \begin{cases} l_1 & \text{if } l_2 \sqsubseteq l_1 \\ \top & \text{otherwise} \end{cases}$$

$$\lambda(l_1, l_2).\, \begin{cases} l_2 & \text{if } l_1 = \bot \\ l_1 & \text{if } l_2 \sqsubseteq l_1 \wedge l_1 \neq \bot \\ \top & \text{otherwise} \end{cases}$$

$$\lambda(l_1, l_2).\, \begin{cases} l_1 \sqcup l_2 & \text{if } l_1 \sqsubseteq l' \vee l_2 \sqsubseteq l_1 \\ \top & \text{otherwise} \end{cases}$$

where l' is a fixed element in the lattice L.

[1]The target language example is not entirely valid. In particular, $\lambda y.\, y*y$ is not a valid expression, as $y*y$ is not a statement. To be in proper Continuation Passing Style, the function would also need to accept a continuation, and flow of control should go to this continuation after the evaluation of the function body. A significant part of problem (a) is determining how to translate abstraction and application from the source language to the target language.

None-the-less, the connection between the target language and machine code should be apparent.

(b) Find sufficient conditions on l' such that the operator involving l' is a widening operator.