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 cpsTranslate.ml to wangyl@cs.cornell.edu by 5PM on that day.

1. Recursive types (20 pts.)

   Once we add recursive types to the simply-typed lambda calculus, we see that the untyped lambda calculus is a subset of the simply-typed lambda calculus in which every variable is declared to have the type $\Lambda = \mu X.X \to X$. An arbitrary lambda calculus expression can be desugared to the typed lambda calculus with recursive types as follows:

$$\begin{aligned} \mathcal{D}[\![x]\!] &= x \\ \mathcal{D}[\![\lambda x\ .\ e]\!] &= (\mathsf{fold}_\Lambda\ (\lambda x : \Lambda.\ \mathcal{D}[\![e]\!])) \\ \mathcal{D}[\![e_0\ e_1]\!] &= ((\mathsf{unfold}\ \mathcal{D}[\![e_0]\!])\ \mathcal{D}[\![e_1]\!]) \end{aligned}$$

   In the untyped lambda calculus, a rec operator is not necessary because we can write a closed expression Y that has the same effect. Using the desugaring above, we can obtain a version of Y for the typed lambda calculus with recursive types. For this problem, we will assume that the language has lazy (normal order) evaluation, so either form of the Y operator that we saw earlier in lecture will work.

   However, the desugared Y operator is not generally useful in the typed lambda calculus because it takes fixed points only for functions from $\mu X.X \to X$ to the same type. For a given type τ, we can define an operator Y_τ that takes fixed points over τ. Write the appropriate typed lambda expression for Y_τ. Make sure that your expression can be proved to have the correct type using the typing rules. To make this straightforward, assume that the fold and unfold operators are annotated with the type of the value that they produce, and have the following typing rules:

$$\frac{\Gamma \vdash e : \tau\{\mu X.\tau/X\}}{\Gamma \vdash \mathsf{fold}_{\mu X.\tau}\ e : \mu X.\tau} \qquad \frac{\Gamma \vdash e : \mu X.\tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau\{\mu X.\tau/X\}}$$

2. Type-safety of a stack language (40 pts.)

   In this problem, you will show that a simple stack language is type-safe by proving preservation and progress lemmas. Consider the following simple stack language, Subscript, inspired by PostScript:

$$\begin{array}{rcl} p & ::= & c \mid c\ p \\ c & ::= & \mathsf{skip} \mid n \mid \oplus \mid \{p\}_\sigma \mid \mathsf{app} \mid \mathsf{ifz} \mid \\ & & /x\ \mathsf{def} \mid x \mid \mathsf{fold}_{\mu X.\tau} \mid \mathsf{unfold} \\ v & ::= & n \mid \{p\}_\sigma \\ S & ::= & [\,] \mid v :: S \\ \tau & ::= & \mathsf{int} \mid \sigma_1 \to \sigma_2 \mid X \mid \mu X.\tau \\ \sigma & ::= & \diamond \mid \tau :: \sigma \\ n & \in & \mathbb{Z} \\ x & \in & \mathit{Var} \\ X & \in & \mathit{TyVar} \end{array}$$

   A program p is a sequence of one or more commands made up of integer pushes, binary arithmetic operations, command sequences, function pushes, function applications, variable definition and lookup, type folding and unfolding operations, and conditionals. The binary arithmetic operations pop two argument integers from the stack and push the resulting integer onto the stack. Pushing a function onto the stack requires both the commands to be executed on function application and an input stack

$$\frac{}{\langle n, S\rangle \longmapsto \langle \mathsf{skip}, n :: S\rangle} \qquad \frac{}{\langle \oplus, n_0 :: n_1 :: S\rangle \longmapsto \langle \mathsf{skip}, (n_1 \oplus n_0) :: S\rangle}$$

$$\frac{}{\langle \{p\}_\sigma, S\rangle \longmapsto \langle \mathsf{skip}, \{p\}_\sigma :: S\rangle} \qquad \frac{}{\langle \mathsf{app}, \{p\}_\sigma :: S\rangle \longmapsto \langle p, S\rangle}$$

$$\frac{}{\langle \mathsf{fold}_{\mu X.\tau}, v :: S\rangle \longmapsto \langle \mathsf{skip}, v :: S\rangle} \qquad \frac{}{\langle \mathsf{unfold}, v :: S\rangle \longmapsto \langle \mathsf{skip}, v :: S\rangle}$$

$$\frac{}{\langle \mathsf{ifz}, \{p_0\}_\sigma :: \{p_1\}_\sigma :: 0 :: S\rangle \longmapsto \langle p_1, S\rangle} \qquad \frac{}{\langle \mathsf{ifz}, \{p_0\}_\sigma :: \{p_1\}_\sigma :: n :: S\rangle \longmapsto \langle p_0, S\rangle}\ (n \neq 0)$$

$$\frac{}{\langle /x\ \mathsf{def}, v :: S\rangle \longmapsto \langle \mathsf{skip}, S\rangle} \qquad \frac{}{\langle /x\ \mathsf{def}\ p, v :: S\rangle \longmapsto \langle p\{v/x\}, S\rangle}$$

$$\frac{}{\langle \mathsf{skip}\ p, S\rangle \longmapsto \langle p, S\rangle} \qquad \frac{\langle c, S\rangle \longmapsto \langle c', S'\rangle}{\langle c\ p, S\rangle \longmapsto \langle c'\ p, S'\rangle}\ (c \neq \mathsf{skip}\ \wedge\ c \neq /x\ \mathsf{def})$$

Figure 1: Operational Semantics

type (analogous to declaring the type of the argument to a function in the typed-lambda calculus). The type of a function is $\sigma_1 \to \sigma_2$, where σ_1 and σ_2 correspond to the types of the input and output stacks, respectively. Function application pops a function from the stack and executes the commands of the function. The command $/x$ def pops a value from the stack and inserts into the environment the binding of the value to the symbol x. Bindings are statically scoped; a binding is in scope until the end of the current function or until the end of the program. The command x pushes the value bound to x in the current environment onto the stack. Folding and unfolding operations do not affect the values on the stack but do affect the type of the top element of the stack. The command ifz pops two functions $\{c_2\}$ and $\{c_1\}$ from the stack and then pops an integer. c_1 is executed if the integer is 0; otherwise c_2 is executed. For example, here is a program which computes the factorial of 5, storing the result as the only value on the stack:

$$\begin{array}{l}
\{\ \ /f\ \mathsf{def}\ /n\ \mathsf{def} \\
\quad n \\
\quad \{1\}_\diamond \\
\quad \{n\ n\ 1\ -\ f\ f\ \mathsf{unfold}\ \mathsf{app}\ *\}_\diamond \\
\quad \mathsf{ifz} \\
\}_{(\mu X.X :: \mathsf{int} :: \diamond \to \mathsf{int} :: \diamond) :: \mathsf{int} :: \diamond} \\
\mathsf{fold}_{\mu X.X :: \mathsf{int} :: \diamond \to \mathsf{int} :: \diamond} \\
/\mathit{fact}\ \mathsf{def} \\
5 \\
\mathit{fact} \\
\mathit{fact}\ \mathsf{unfold}\ \mathsf{app}
\end{array}$$

Note how the recursive call to the factorial function is accomplished by passing a copy of the function to itself as an argument on the stack. Recursive types are required in order to give a type to a function which expects a function with the same type on its input stack.

The operational semantics for Subscript are given in Figure 1. A configuration is a pair $\langle p, S\rangle$ consisting of a program and a stack. The initial configuration for a program p is given by $\langle p, [\,]\rangle$. The final configurations are of the form $\langle \mathsf{skip}, S\rangle$. Note that the rule for $/x$ def p handle substituting the value on the top of the stack for x in the remainder of the program p. The other rule for $/x$ def only applies for programs which consist only of the definition. In Figure 2 we define the substitution rules.

Figure 3 gives the typing rules for a subset of the commands. A judgment of the form $\Gamma; \sigma \vdash p : \sigma'$ asserts that if the program p is executed with a typing context Γ and a stack of type σ, then, if and when the program terminates, it will terminate with a stack of type σ'. In the rule for app,

$$\begin{aligned}
x\{v/x\} &= v && \\
y\{v/x\} &= y && x \neq y \\
\{p\}_\sigma\{v/x\} &= \{p\{v/x\}\}_\sigma && \\
c\{v/x\} &= c && \text{for all other commands } c \\
(/x\ \mathsf{def}\ p)\{v/x\} &= /x\ \mathsf{def}\ p && \\
(/y\ \mathsf{def}\ p)\{v/x\} &= /y\ \mathsf{def}\ p\{v/x\} && x \neq y \\
(c\ p)\{v/x\} &= (c\{v/x\}\ p\{v/x\}) && (c \neq /x\ \mathsf{def})
\end{aligned}$$

Figure 2: Substitution Rules

$$\frac{}{\Gamma;\sigma \vdash \mathsf{skip} : \sigma} \qquad \frac{}{\Gamma;\sigma \vdash n : \mathsf{int} :: \sigma} \qquad \frac{}{\Gamma;\sigma \vdash x : \Gamma(x) :: \sigma}$$

$$\frac{}{\Gamma;(\sigma_1 \to \sigma_2) :: (\sigma_1 \mathbin{@} \sigma) \vdash \mathsf{app} : (\sigma_2 \mathbin{@} \sigma)}$$

$$\frac{}{\Gamma;(\tau\{\mu X.\tau/X\}) :: \sigma \vdash \mathsf{fold}_{\mu X.\tau} : (\mu X.\tau) :: \sigma} \qquad \frac{}{\Gamma;(\mu X.\tau) :: \sigma \vdash \mathsf{unfold} : (\tau\{\mu X.\tau/X\}) :: \sigma}$$

$$\frac{}{\Gamma;(\tau :: \sigma) \vdash /x\ \mathsf{def} : \sigma} \qquad \frac{\Gamma;(\tau :: \sigma) \vdash /x\ \mathsf{def} : \sigma \quad \Gamma[x \mapsto \tau];\sigma \vdash p : \sigma'}{\Gamma;(\tau :: \sigma) \vdash /x\ \mathsf{def}\ p : \sigma'}$$

$$\frac{\Gamma;\sigma \vdash c : \sigma'' \quad \Gamma;\sigma'' \vdash p : \sigma'}{\Gamma;\sigma \vdash c\ p : \sigma'}\ (c \neq /x\ \mathsf{def})$$

Figure 3: Typing Rules for Commands

the notation $\sigma_a \mathbin{@} \sigma_b$ stands for the concatenation of the stack type σ_b to the stack type σ_a (*e.g.*, $(\tau_1 :: \cdots :: \tau_a :: \diamond) \mathbin{@} (\tau'_1 :: \cdots :: \tau'_b :: \diamond) = \tau_1 :: \cdots :: \tau_a :: \tau'_1 :: \cdots :: \tau'_b :: \diamond$).

In order to prove the soundness of this type system, we need to relate typing judgments to configurations. Figure 4 gives the typing rules for stack values and stacks. A judgment of the form $\Gamma \vdash S : \sigma$ asserts that the stack S has the stack type σ in typing context Γ. We write $\vdash S : \sigma$ as shorthand for $\emptyset \vdash S : \sigma$.

Now, we can assert that a configuration is well-typed:

$$\frac{\Gamma;\sigma \vdash p : \sigma' \quad \Gamma \vdash S : \sigma}{\Gamma;\sigma \vdash \langle p, S\rangle : \sigma'}$$

In particular, a program is well-typed if its initial configuration is well-typed, *i.e.*, if $\emptyset;\diamond \vdash \langle p, [\,]\rangle : \sigma$.

Using this notation, we can state the soundness of the Subscript type system as follows:

$$\diamond \vdash \langle p, [\,]\rangle : \sigma \ \wedge\ \langle p, [\,]\rangle \longmapsto^* \langle p', S'\rangle \implies (p' = \mathsf{skip} \ \vee\ \exists\langle p'', S''\rangle.\ \langle p', S'\rangle \longmapsto \langle p'', S''\rangle)$$

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma;\sigma_1 \vdash p : \sigma_2}{\Gamma \vdash \{p\}_{\sigma_1} : \sigma_1 \to \sigma_2} \qquad \frac{\Gamma \vdash v : \tau\{\mu X.\tau/X\}}{\Gamma \vdash v : \mu X.\tau} \qquad \frac{\Gamma \vdash v : \mu X.\tau}{\Gamma \vdash v : \tau\{\mu X.\tau/X\}}$$

$$\frac{}{\Gamma \vdash [\,] : \diamond} \qquad \frac{\Gamma \vdash v : \tau \quad \Gamma \vdash S : \sigma}{\Gamma \vdash v :: S : \tau :: \sigma}$$

Figure 4: Typing Rules for Values and Stacks

As usual, we will need to strengthen our assumptions to prove the preservation and progress lemmas. Prove soundness by completing the following problems.

(a) Typing Rules for Commands: Complete the typing rules for commands by giving inference rules for $\{p\}_{\sigma_1}$, $\oplus$, and ifz.

(b) Lemma: Prove $\Gamma; \sigma_a \vdash p : \sigma_b \implies \Gamma; (\sigma_a \,@\, \sigma) \vdash p : (\sigma_b \,@\, \sigma)$. This lemma will be useful for completing the proof of preservation. (Note: It would be reasonable to add the following typing rule to the typing rules for commands:

$$\frac{\Gamma; \sigma_a \vdash p : \sigma}{\Gamma; (\sigma_a \,@\, \sigma) \vdash p : (\sigma_b \,@\, \sigma)}$$

However, this rule complicates proofs of preservation and progress, because the derivation of a typing judgment is no longer syntax-directed. Likewise, inclusion of this rule complicates the implementation of a type-checker. Further, because the lemma can be proved without this typing rule, the set of typing rules is kept to a minimum.)

(c) Substitution Lemma: Prove $\Gamma; \sigma \vdash p : \sigma' \;\wedge\; \Gamma(x) = \tau \;\wedge\; \Gamma \vdash v : \tau \implies \Gamma; \sigma \vdash p\{v/x\} : \sigma'$.

(d) Preservation: Prove $\sigma \vdash \langle p, S\rangle : \sigma' \;\wedge\; \langle p, S\rangle \longmapsto \langle p', S'\rangle \implies \exists \sigma''.\ \sigma'' \vdash \langle p', S'\rangle : \sigma'$.

(e) Progress: Prove $\sigma \vdash \langle p, S\rangle : \sigma' \implies (p = \mathsf{skip} \;\vee\; \exists \langle p'', S''\rangle.\ \langle p, S\rangle \longmapsto \langle p'', S''\rangle)$.

(f) Strong Normalization: Is Subscript strongly normalizing (with a suitable definition of normal form)? Why or why not? (You do not need to prove your claim, but some justification is expected.)

3. Closure Conversion + CPS = Assembly code (40 pts.)

In this problem you will translate a language similar to the simple target language from Problem 6, HW3 into a CPS language similar to the one from Problem 4 on the prelim. This CPS language is very close to assembly language, so you will be exploring some of the transformations needed for compilation to assembly.

The key idea of the target CPS 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 syntax for the source language is:

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

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 LambdaSimpleP.parse, which should make it easy to write test cases for your translation and turn them into LambdaSimpleAst.expr's.

The target language is given by:

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

Here, $\langle x_1, \ldots, x_n\rangle$ represents a "tuple" of n elements; if $y = \langle x_1, \ldots, x_n\rangle$ then $y[n] \to x_n$. The statement ($\mathsf{halt}\ x$) 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.

There is a strong correspondence between the expressions e and statements s of the CPS language and the instructions of an ordinary machine language. For example, instead of writing

$$\begin{array}{l}\mathsf{let}\ x_1 = x_2 + x_3\ \mathsf{in}\\ \mathsf{let}\ f = \lambda y.\ y\ *\ y\ \mathsf{in}\\ f\ x_1\end{array}$$

we could write

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

Implement a translation from the source language to the CPS 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 are allowed to modify only the file "cpsTranslate.ml". To see some of the parsing and translation functions being used, look in "main.ml" for some examples. You may choose to perform both CPS and closure conversion in one pass, or separate them into two passes. However, note that the evaluator does not know how to handle lambda terms that have free variables. If you want to do CPS conversion first, you can modify "cpsEvaluation.ml" just for debugging purposes. By changing empty_env to env on line 97, you will not have this restriction for closures any more, so you can test your result for CPS separately. Remember that this is only for debugging purposes. Your program should run without this modification in the end to satisfy closure property.

Your translation should be a function cps_translate : LambdaSimpleAst.expr→CpsAst.stm. Given a source language expression which evaluates to a base value n, evaluating the result of your CPS translator using CpsEvaluation.evaluate should also result in n.