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 translate.sml to jcheney@cs.cornell.edu by 5PM on that day.

1. Recursive types (10 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:

   $$\mathcal{D}[\![x]\!] = x$$

   $$\mathcal{D}[\![\lambda x\ .\ e]\!] = (\mathsf{fold}_\Lambda\ (\lambda x : \Lambda.\ \mathcal{D}[\![e]\!]))$$

   $$\mathcal{D}[\![e_0\ e_1]\!] = ((\mathsf{unfold}\ \mathcal{D}[\![e_0]\!])\ \mathcal{D}[\![e_1]\!])$$

   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 (20 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, STACK:

   $$\begin{array}{rcl} n & \in & \mathbb{Z} \\ i & \in & \mathbb{N} \\ c & ::= & \mathsf{skip} \mid n \mid \oplus \mid \{c\}_\sigma \mid \mathsf{app} \mid \mathsf{dup} \mid \mathsf{pop} \mid \mathsf{swap}_i \mid \\ & & \mathsf{fold}_{\mu X.\tau} \mid \mathsf{unfold} \mid \mathsf{ifz}\ c_1\ c_2 \mid c_1\ ;\ c_2 \\ v & ::= & n \mid \{c\}_\sigma \\ S & ::= & \cdot \mid v :: S \\ X & \in & TyVar \\ \tau & ::= & \mathsf{int} \mid \sigma_1 \to \sigma_2 \mid X \mid \mu X.\tau \\ \sigma & ::= & \cdot \mid \tau :: \sigma \end{array}$$

   A program c is a sequence of commands made up of skips, integer pushes, binary arithmetic operations, function pushes, function applications, stack duplication, pop, and swap operations, type folding and

Figure 1: Operational Semantics

$$\frac{}{\langle n, S\rangle \to \langle \mathsf{skip}, n :: S\rangle} \qquad \frac{}{\langle \oplus, n_1 :: n_2 :: S\rangle \to \langle \mathsf{skip}, (n_1 \oplus n_2) :: S\rangle}$$

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

$$\frac{}{\langle \mathsf{dup}, v :: S\rangle \to \langle \mathsf{skip}, v :: v :: S\rangle} \qquad \frac{}{\langle \mathsf{pop}, v :: S\rangle \to \langle \mathsf{skip}, S\rangle}$$

$$\frac{}{\langle \mathsf{swap}_i, v_0 :: v_1 :: \cdots :: v_{i-1} :: v_i :: S\rangle \to \langle \mathsf{skip}, v_i :: v_1 :: \cdots :: v_{i-1} :: v_0 :: S\rangle}$$

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

$$\frac{}{\langle \mathsf{ifz}\ c_1\ c_2, 0 :: S\rangle \to \langle c_1, S\rangle} \qquad \frac{}{\langle \mathsf{ifz}\ c_1\ c_2, n :: S\rangle \to \langle c_2, S\rangle}\ n \neq 0$$

$$\frac{\langle c_1, S\rangle \to \langle \mathsf{skip}, S'\rangle}{\langle c_1\ ;\ c_2, S\rangle \to \langle c_2, S'\rangle} \qquad \frac{\langle c_1, S\rangle \to \langle c_1', S'\rangle}{\langle c_1\ ;\ c_2, S\rangle \to \langle c_1'\ ;\ c_2, S'\rangle}\ c_1' \neq \mathsf{skip}$$

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 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 dup duplicates the top value on the stack, pop pops the top value on the stack, swap_i swaps the top value on the stack with the i^{th} value on the stack (the top value on the stack is the 0^{th} value). Folding and unfolding operations do not affect the values on the stack, but do effect the type of the top element of the stack. The command ifz c_1 c_2 pops an integer from the stack and exectutes c_1 if the integer is 0 and executes c_2 otherwise. For example, here is a program which computes the factorial of 5, storing the result as the only value on the stack:

```
5 ;
{swap1 ;dup ;ifz (pop ; pop ; 1)
                 (dup ; 1 ; swap1 ; − ; swap1 ; swap2 ; dup ; unfold ; app ; ∗)
}(μX.X::int::·→int::·)::int::· ;
fold_μX.X::int::·→int::· ;
dup ;
unfold ;
app
```

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 STACK are given in Figure 1. A configuration is a pair $\langle c, S\rangle$ consisting of a command and the stack. The initial configuration for a program c is given by $\langle c, \cdot\rangle$. The final configurations are of the form $\langle \mathsf{skip}, S\rangle$.

Figure 2 gives the typing rules for a subset of the commands. A judgement of the form $\sigma \vdash c : \sigma'$ asserts that if the command c is executed with a stack of type σ, then, when and if the command terminates, it will terminate with a stack of type σ'. In the rule for app, the notation $\sigma_A \mathbin{@} \sigma_B$ stands

Figure 2: Typing Rules for Commands

$$\frac{}{\sigma \vdash \mathsf{skip} : \sigma} \qquad \frac{}{\sigma \vdash n : \mathsf{int} :: \sigma}$$

$$\frac{\sigma_1 \vdash c : \sigma_2}{\sigma \vdash \{c\}_{\sigma_1} : (\sigma_1 \to \sigma_2) :: \sigma} \qquad \frac{}{(\sigma_1 \to \sigma_2) :: (\sigma_1 \mathbin{@} \sigma) \vdash \mathsf{app} : (\sigma_2 \mathbin{@} \sigma)}$$

$$\frac{}{\tau_0 :: \tau_1 :: \cdots :: \tau_{i-1} :: \tau_i :: \sigma \vdash \mathsf{swap}_i : \tau_i :: \tau_1 :: \cdots :: \tau_{i-1} :: \tau_0 :: \sigma}$$

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

$$\frac{\sigma \vdash c_1 : \sigma'' \quad \sigma'' \vdash c_2 : \sigma'}{\sigma \vdash c_1 \; ; c_2 : \sigma'}$$

Figure 3: Typing Rules for Values and Stacks

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

$$\frac{}{\cdot : \cdot} \qquad \frac{v : \tau \quad S : \sigma}{v :: S : \tau :: \sigma}$$

for the concatenation of the stack type σ_B to the stack type σ_A (e.g., $\tau_1 :: \cdots :: \tau_a :: \cdot \mathbin{@} \tau'_1 :: \cdots :: \tau'_b :: \cdot = \tau_1 :: \cdots :: \tau_a :: \tau'_1 :: \cdots :: \tau'_b :: \cdot$).

In order to prove the soundness of this type system, we need to relate typing judgements to configurations. Figure 3 gives the typing rules for stack values and stacks. A judgement of the form $S : \sigma$ asserts that the stack S has the stack type σ.

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

$$\frac{\sigma \vdash c : \sigma' \quad S : \sigma}{\sigma \vdash \langle c, S\rangle : \sigma'} \; .$$

In particular, a program is well-typed if it's inital configuration is well-typed, i.e., if $\cdot \vdash \langle c, \cdot\rangle : \sigma$.

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

$$\cdot \vdash \langle c, \cdot\rangle : \sigma \;\wedge\; \langle c, \cdot\rangle \to^* \langle c', S'\rangle \Rightarrow (c' = \mathsf{skip} \;\vee\; \exists \langle c'', S''\rangle . \langle c', S'\rangle \to \langle c'', S''\rangle).$$

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 $\oplus$, dup, pop, and ifz c_1 c_2.

(b) Lemma: Prove $\sigma_A \vdash c : \sigma_B \Rightarrow (\sigma_A \mathbin{@} \sigma) \vdash c : (\sigma_B \mathbin{@} \sigma)$. 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:

$$\frac{\sigma_A \vdash c : \sigma}{(\sigma_A \mathbin{@} \sigma) \vdash c : (\sigma_B \mathbin{@} \sigma).}$$

However, this rule complicates proofs of preservation and progress, becuase the derivation of a typing judgement 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) Preservation: Prove $\sigma \vdash \langle c, S\rangle : \sigma' \ \wedge \ \langle c, S\rangle \rightarrow \langle c', S'\rangle \Rightarrow \exists \sigma''.\sigma'' \vdash \langle c', S'\rangle : \sigma'$.

(d) Progress: Prove $\sigma \vdash \langle c, S\rangle : \sigma' \Rightarrow (c = \mathsf{skip} \ \vee \ \exists \langle c'', S''\rangle.\langle c, S\rangle \rightarrow \langle c'', S''\rangle)$.

(e) Strong Normalization: Is STACK 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 (20 pts.)

In this problem you will translate a language similar to the simple target language from Problem 4, HW3 into a CPS language even simpler than the one from Problem 3 on the prelim. This simpler CPS language is very close to assembly language, so you will be exploring some of the transformations needed for compilation to assembly.

The key simplification 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).

Unlike the target language of HW3, the source language here is a strict, call-by-value language. To simplify writing examples, it also adds built-in boolean constants and an if expression. (Recall that the target language in HW3 required encoding booleans and if as function values.)

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. Note that this differs from the zero? operator in the target language of HW3, which produced lambda terms that were *your translations* of #t and #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 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 $y[n] \rightarrow x_n$. The statement (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.

Combining the translation you will work out in this problem with a pass that converts the lazy target language of HW3 into a strict language yields a compiler from a fairly high-level lazy language to a low-level language that's only a stone's throw away from assembly or machine language. Of course, this

compiler may not necessarily produce the most efficient code, since it encodes simple data structures like pairs and tagged cases as function calls.

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 "cps-translate.sml". To see some of the parsing and translation functions being used, look in "top.sml" 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.

Optional: We have provided the lazy-to-strict translation in the function fromLambdaAST in case you want to try out the whole chain of translations from HW3 on down. To make this work you will need to be able to translate the zero? expression from the HW3 target language, which returned lambda terms representing true and false. We have included an expression (zero? e) in the source language and (zero? x) in this target language only so you can translate this expression; if you want to implement it (which is not required), you simply need to provide a CPS translation for (zero? e), which should be straightforward. Your translation should be a function cps_translate : LambdaSimpleAst.exp→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.