MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Problem Set 4

Problem 1: Flavors of Naming

Do exercise 7.12 on pages 237–238 of the course notes.

Problem 2: Compiling FLEX to FLAT

In this problem, you will design and implement an algorithm for translating one language into another. The translation process considered here is actually used in compilers for languages with lexically scoped first-class procedures. The goal of this problem is to develop a deeper understanding of first-class procedures and lexical scoping.

Warning: this problem is difficult! Some suggestions:

- Please, do not leave this problem until the last minute.
- Do not attempt to code the solution until you have fully fleshed out your design. Doing otherwise is a recipe for disaster. This problem is more about thinking and design than actual code—our solution is about 60 lines of code and we do not expect yours to be significantly longer (and it may certainly be shorter).
- We strongly suggest that you work with others on the design phase and even the coding phase. However, writeup should be done individually. Be sure to acknowledge others that you work with.

The two languages used in this problem, FLEX and FLAT, are both variants of call-by-value FL. The kernel syntax for FLEX appears in figure 1; FLAT's syntax is given in figure 2. In both languages, the `primop` construct inherits its usual meaning from FLK. The primitive operators O supported in both languages are the same as in FLK.

FLEX is just call-by-value FLK without the `rec` construct. FLEX also inherits the syntactic sugar of FL, except for `letrec` and `program`, which are not supported.

FLAT is very similar to FLEX. They are the same except for the following differences:

- In FLAT, `proc` expressions can't have any free variables. This is expressed in the grammar by the restriction:

  $$\mathit{Free\text{-}Ids}[\![\texttt{(proc } I \ E\texttt{)}]\!] = \emptyset.$$

  This restriction effectively makes all FLAT procedures independent of each other. In FLAT, it is possible to straightforwardly "lift" all `proc` expressions in any program to the top-level. Such a lifting is not possible in FLEX because `proc` expressions may have free variables that make them dependent on their lexical position in the program.
- In FLAT, the `let` expression is a primitive form of the language — it does not desugar into other forms.
- FLAT supports strict, immutable arrays called *tuples*:
  - `(tuple` E_1 `...` E_n`)` creates an n-tuple whose n components are the values of $E_1 \ldots E_n$.

$$
\begin{array}{rcll}
E & ::= & L & \text{[literal]} \\
 & | & I & \text{[variable-reference]} \\
 & | & \texttt{(primop } O\ E^*\texttt{)} & \text{[primitive-application]} \\
 & | & \texttt{(proc } I\ E\texttt{)} & \text{[abstraction]} \\
 & | & \texttt{(call } E_p\ E_a\texttt{)} & \text{[application]} \\
 & | & \texttt{(pair } E_l\ E_r\texttt{)} & \text{[pairing]} \\
 & | & \texttt{(if } E_b\ E_t\ E_f\texttt{)} & \text{[branch]}
\end{array}
$$

Figure 1: FLEX Kernel Syntax

$$
\begin{array}{rcll}
E & ::= & L & \text{[literal]} \\
 & | & I & \text{[variable-reference]} \\
 & | & \texttt{(primop } O\ E^*\texttt{)} & \text{[primitive-application]} \\
 & | & \texttt{(proc } I\ E\texttt{)} & \text{[abstraction]} \\
 & | & \texttt{(call } E_p\ E_a\texttt{)} & \text{[application]} \\
 & | & \texttt{(pair } E_l\ E_r\texttt{)} & \text{[pairing]} \\
 & | & \texttt{(if } E_b\ E_t\ E_f\texttt{)} & \text{[branch]} \\
 & | & \texttt{(let ((}I\ E\texttt{)}^*\texttt{)} \ E_b\texttt{)} & \text{[let]} \\
 & | & \texttt{(tuple } E^*\texttt{)} & \text{[tuple]} \\
 & | & \texttt{(tuple-ref } E\ N\texttt{)} & \text{[tuple-ref]} \\
 & | & \texttt{(tuple? } E\texttt{)} & \text{[tuple-pred]} \\
 & | & \texttt{(tuple-length } E\texttt{)} & \text{[tuple-length]} \\
 & | & \texttt{(tuple-append } E_1\ E_2\texttt{)} & \text{[tuple-append]}
\end{array}
$$

$$\textit{Free-Ids}[\![\texttt{(proc } I\ E\texttt{)}]\!] = \emptyset$$

Figure 2: FLAT Kernel Syntax

- `(tuple-ref` E N`)` evaluates E to an n-tuple t and returns the Nth component of t, where indices are 0-based. It is an error if E does not evaluate to an n-tuple or if N not an integer between 0 and $n-1$, inclusive. For example:

```
(define tup
  (tuple (primop + 1 2)
         (primop * 3 4)
         (primop = 7 8)))

(tuple-ref tup 0) ⇒ 3
(tuple-ref tup 1) ⇒ 12
(tuple-ref tup 2) ⇒ false
(tuple-ref tup 3) ⇒ error
```

- `(tuple?` E`)` is a predicate that indicates whether the value of E is a tuple.
- `(tuple-length` E`)` evaluates E to an n-tuple and returns its length. It is an error if E does not evaluate to an n-tuple.
- `(tuple-append` E_1 E_2`)` evaluates E_1 and E_2 to n-tuples and returns a new n-tuple consisting of the elements of the first tuple followed by the elements of the second tuple. It is an error if either E_1 or E_2 does not evaluate to an n-tuple.

Not all tuple operations may be needed in writing your translator, but we provide them for completeness. All tuple operations except for tuple creation could have been primitives, but we have made them special forms so that FLAT expressions are more readable.

Your assignment in this problem is to design and implement a semantics-preserving program translation that translates FLEX expressions to FLAT expressions.

Your solution is required to be "semantics-preserving" in the following sense. Your solution must produce expressions that compute the same result as the original expression in all cases where that result is not a procedure or a data structure containing a procedure. (Your translator must always halt no matter what its input is, so it cannot simply evaluate the input expression.)

Why is this worth doing? Because this particular program transformation is actually used in real compilers. The resulting programs contain only `proc` expressions with no free variables; therefore all the `proc` expressions can be treated as top-level procedures. This transformation is used when compiling Pascal to assembly language, and when compiling Scheme to C.

There are several important things to keep in mind about this problem:

- To design your translation, you must think about how a compiler represents procedure values (closures). In a lexically scoped language, how are procedure values represented? What does a Pascal compiler pass when a procedure value is passed as an argument? How is such a procedure value called? When it is called, how does the body of the procedure access its environment?

- The purpose of this translation is to take a language with first-class procedure expressions and translate it into a more restricted language. You should not restrict yourself to translating only `proc` expressions. You can translate anything and everything in the language, as you see fit.

- Your translator only needs to handle *closed expressions*, that is, expressions that have no free variables. In particular, you do *not* have to worry about top-level bindings of standard identifiers like `+`, `left`, etc. The `primop` construct makes it possible to write interesting expressions without depending on the bindings of standard identifiers. For example, your translator must handle the top-level expression `(primop + 1 2)`, but it does not need to handle the top-level expression `(+ 1 2)`.

- The translation of an expression should not require examining the result of translating subexpressions.

- The FLAT code produced by your translator need not be particularly efficient. You needn't worry about optimizing the output code.

- The translator itself does not need to be particularly efficient. When writing the translator, emphasize clarity rather than efficiency. Make the translator as simple as possible.

- Your translator must return a FLAT expression for every closed FLEX expression. In particular, your translator must always terminate, even when applied to FLEX expressions that might not terminate.

- It is crucial to handle the case where a procedure is returned outside the scope of the declaration of one of the variables referenced in its body. For example, consider:

```
(let ((make-subtracter (proc n (proc x (primop - x n)))))
  (let ((dec (make-subtracter 1)))
    (dec 5)))
```

  When `make-subtracter` is applied to `1`, it returns a procedure created by the expression `(proc x (primop - x n))`. The returned procedure somehow "remembers" that the `n` in the body means `1`. Implementing this "remembering" behavior is at the heart of the problem.

- You need to correctly transform the following FLEX expressions:

```
(primop procedure? (proc x x))   ⟹FLEXTFLAT ⇒FLAT  true
(primop procedure? 1)            ⟹FLEXTFLAT ⇒FLAT  false
```

Tools. Did we mention tools? Yes, well, we're providing the following pieces of software, all conveniently located in the file `ps4.scm`:

- Definition of the abstract syntax: since the languages are so similar, we will use the same `exp` datatype for both.
- A function for computing the free variables of FLEX and FLAT expressions: `free-vars`.
- Restriction checker, verifies that a FLAT expression contains only `proc` expressions which have no free variables: `non-scoped?`.
- Two language evaluators: `flex-eval` and `flat-eval`.
- The other usual stuff: `flex-parse`, `flat-parse`, `flex-unparse`, `flat-unparse`, and two top-levels: `flex-repl` and `flat-repl`.
- Two testing procedures, `test-translate` and `test-loop`, that use your `translate` procedure. `test-translate` will prompt for a single FLEX expression and produce the resulting FLAT expression from your translator. `test-loop` will perform the same task and prompt for another FLEX expression.
- A lifter procedure `lift` that when applied to any FLAT expression, will lift all `proc` expressions to the top-level. You can use this to see what happens in a compiler when the procedures are lifted to the top-level. The top-level is represented by the FLAT program (`program` $(I\ E)^*\ E_b$). The bindings for the identifiers are mutually recursive so that `procs` within the body of other `procs` are correctly handled.

  You can use `lift-loop-on-flex` to try lifting FLEX expressions, but it will complain if it is not safe to lift the `procs` in the expression. You can use `lift-loop-with-translate` to enter FLEX expressions, have them translated by your translator, and then lifted. If all goes well, you should see a program that binds some `procs` followed by some calls involving those `procs`.

You should submit the following results:

a. A brief explanation of how you translate `proc` abstractions, stating why your approach works.

b. A formal description of your translation rules. By this we mean something along the lines of the HOOK to FL translator of Chapter 9 (figure 9.19 on page 356 of the course notes). That is, you should formally define a translation function $\mathcal{T}$ that maps FLEX expressions (and, potentially, other arguments) to FLAT expressions. The rules should be the ones followed by your translation program, but written in a form that makes your translation as clear as possible.

c. The code for your translation procedure, `translate`, and any auxiliary routines you define. `translate` should map *parsed* FLEX expressions to *parsed* FLAT expressions.

d. Anything else you decide to change. (We don't expect that other changes will be necessary.)

e. Test cases showing the behavior of your `translate` procedure. You should make sure your `translate` procedure is semantics-preserving. You can use the language implementations to help test this, but we only want to see *unparsed* expressions before and after translation.