Solutions for Problem Set 3

DPL Seminar, Summer 2001
Handout 7

Konstantine Arkoudas

August 2, 2001
Problem 1

Part A

(define (literal? P)
  (match P
    [(some-atom A) true]
    [(not (some-atom A)) true]
    (_ false))

(define (literal-disjunction? P)
  (match P
    [(or P1 P2) (& (literal-disjunction? P1) (literal-disjunction? P2))]
    (_ (literal? P)))))

(define (cnf? P)
  (match P
    [(and P1 P2) (& (cnf? P1) (cnf? P2))]
    (_ (literal-disjunction? P)))))

Part B

By a propositional rewrite rule we will mean any unary Athena function \( f \) that takes a proposition \( P \) and, if \( P \) is of a certain form, then \( f \) produces some other proposition \( Q \) (the “rewrite image” of \( P \)); otherwise \( f \) returns the sentinel ‘no-match’. The relevant rewrite rules for this problem are the following:

(define (dn-rule P)
  (match P
    [(not (and P1 P2)) (or (not P1) (not P2))]
    [(not (or P1 P2)) (and (not P1) (not P2))]
    (_ 'no-match))

(define (bicond-rule P)
  (match P
    [(iff P1 P2) (and (if P1 P2) (if P2 P1))]
    (_ 'no-match))

(define (cond-rule P)
  (match P
    [(if P1 P2) (or (not P1) P2)]
    (_ 'no-match))

(define (dn-rule P)
  (match P
    [(not (not Q)) Q]
    (_ 'no-match))

(define (dist-rule P)
  (match P
    [(or P1 (and P2 P3)) (and (or P1 P2) (or P1 P3))]
    [(or (and P1 P2) P3) (and (or P1 P3) (or P2 P3))]
    (_ 'no-match))

If a rule \( f \) is applicable to a proposition \( P \) (meaning that if we apply \( f \) to \( P \) we will get out some other proposition \( Q \) rather than ‘no-match’), then we say that \( P \) is a “candidate” for \( f \). For instance, the candidates of \( \text{dn-rule} \) above are all and only those propositions of the form \( \neg P \). The function \( \text{rtd} \) (“rewrite-top-down”) below will do most of the work: it takes a proposition \( P \) and a rewrite rule \( f \) and applies \( f \) to the topmost candidate subproposition(s) of \( P \); non-candidates are left unchanged:
(define (rtd P rule)
  (match (rule P)
    (((some-prop Q) Q)
      _ (match P
        ((not Q) (not (rtd Q rule)))
        (((some-prop-con pc) P1 P2) (pc (rtd P1 rule) (rtd P2 rule)))
        _ P)))))

The following function repeats this as many times as necessary in order to converge (to arrive at a “fixed point”):

(define (fix-rule-top-down P rule)
  (let ((new-P (rtd P rule)))
    (check ((equal? P new-P) P)
      (else (fix-rule-top-down new-P rule)))))

The following is a convenient generalization of fix-rule-top-down: it takes a list of rewrite rules instead of just one, and “fixes” each of them in turn:

(define (fix-rules-top-down P rules)
  (match rules
    ([P]
      (list-of rule rest) (fix-rules-top-down (fix-rule-top-down P rule) rest))))

The solution is now straightforward:

(define cnf-rewrite-rules [bicond-rule cond-rule dm-rule dm-rule dist-rule])
(define (cnf P) (fix-rules-top-down cnf-rewrite-rules))

Observe that this solution separates “logic” from “control”. The control part is encapsulated in rtd and fix-rule-top-down and has nothing whatsoever to do with the idea of conjunctive normal form. These functions can be reused with arbitrary rewrite rules, to achieve many different kinds of transformations (e.g., to convert propositions to disjunctive normal form). The only parts of the code that are specific to CNF are the five rules, dm-rule—dist-rule, each of which is very short and simple (all five have constant-time complexity).

Part C

Here we will try to adapt the above solution to a deductive setting. Since inference rather than plain computation is required, we can expect to work a bit harder. The role of rewrite rules will now be played by rewriters. By a “rewriter” we will understand a unary method \( M \) that takes a proposition \( P \), and if \( P \) is of a certain form then \( M \) derives an equivalence \( P \iff Q \), where \( Q \) is usually “simpler” than \( P \) in some appropriate sense. If the input \( P \) is not of the expected form then \( M \) returns the trivial theorem true. For instance, let negate-twice be as follows:

(define (negate-twice P)
  (suppose-aburd (not P)
    (!aburd P (not P)))))

and consider the following rewriter method:

(define (dm-rewriter P)
  (dmatch P
    ((not (not Q)) (!equiv (assume P (!dm P))
        (assume Q (!negate-twice Q))))
    _ (!claim true))))

\(^1\)This terminology is arbitrary. We could have called these “deductive rewrite rules”, but “rewriters” is shorter.
This method is a rewriter that takes any proposition of the form \( \neg \neg Q \) and derives the equivalence \( \neg \neg Q \iff Q \). For example:

\[
> (!dn-rewriter (not (not ?foo)))
\]

**Theorem:** (iff (not (not ?foo))

?foo)

If the input proposition is not of the form \( \neg \neg Q \), then **dn-rewriter** will simply derive \textit{true}:

\[
> (!dn-rewriter (or ?A ?B))
\]

**Theorem:** \textit{true}

Any proposition \( P \) that is appropriate for a rewriter \( M \)---i.e., any \( P \) such that \( (!M \ P) \) successfully produces a biconditional rather than \textit{true}---is called a \textit{candidate} for \( M \). For example, the candidates for **dn-rewriter** are all and only those propositions that are of the form \( \neg \neg Q \).

The analogue of \texttt{rtd} here is the method **rewrite-top-down** below, which takes a theorem \( P \) and a rewriter \( M \) and scans \( P \) top-down looking for a subproposition \( Q \) that is a candidate for \( M \). If and when such a \( Q \) is found, we apply \( M \) to it, producing a biconditional \( Q \iff Q' \), and then we replace every occurrence of \( Q \) inside \( P \) by \( Q' \), using the method **replace-equiv** from the second problem set. Thus we essentially view the equivalence \( Q \iff Q' \) produced by the rewriter \( M \) as a reduction rule \( Q \rightarrow Q' \). If no subproposition of \( P \) turns out to be a candidate for \( M \), then **rewrite-top-down** simply reiterates \( P \) using **claim**. The top-down scanning of \( P \) is accomplished by the auxiliary method **find-candidate**, which takes a rewriter \( M \) and a list of propositions \( L \) and scans \( L \) from left to right looking for a candidate for \( M \). If and when such a candidate \( Q \) is found, the resulting biconditional \( (!M \ Q) \) is returned; if \( L \) does not contain any candidates for \( M \), we return **true**. Thus the aforementioned “scanning” of \( P \) can be achieved by calling **find-candidate** on \( M \) and a list of all the subpropositions of \( P \), appearing in top-down order. Such a list is obtained by the function **sub-props**:

\[
\begin{align*}
\text{(define (sub-props P)} \\
\text{  (match P)} \\
\text{    (\( \text{(not Q)} (\text{add P (sub-props Q)}) \))} \\
\text{    (\( \text{((some-prop-con pc) P1 P2)} (\text{join P} (\text{sub-props P1}) (\text{sub-props P2)}) \))} \\
\text{    (_ [P])))} \\
\end{align*}
\]

**Theorem:** \textit{true}

Convergence is achieved by deductive counterparts of the previous fixed-point functions:

\[
\begin{align*}
\text{(define (fix-rewriter-top-down M P)} \\
\text{  (dlet (new (!rewrite-top-down M P)})} \\
\text{  (dcheck (\text{equal? new P} (!\text{claim new})} \\
\text{    (else (!\text{fix-rewriter-top-down M new})})))} \\
\end{align*}
\]

**Theorem:** \textit{true}

Convergence is achieved by the previous fixed-point functions:

\[
\begin{align*}
\text{(define (fix-rewriters-top-down rewriters P)} \\
\text{  (dmatch rewriters)} \\
\text{    (\[M] (!\text{fix-rewriter-top-down M P})} \\
\text{    (\( \text{\( \text{(list-of M rest)} (\text{fix-rewriters-top-down rest (!\text{fix-rewriter-top-down M P})}) \)} \))} \\
\end{align*}
\]

\[3\]
All we now have to do is define the necessary rewriters, which we do below. We use a few auxiliary methods which are straightforward exercises in propositional reasoning: \texttt{dn} for disjunctive syllogism (this was defined in the solutions to the first problem set); \texttt{dm} for De Morgan’s, which takes any theorem of the form \( \neg(P \land Q) \), or \( \neg(P \lor Q) \), \( \neg P \land \neg Q \) or \( \neg P \lor \neg Q \) and produces the appropriate conclusion; and \texttt{cases}, which takes two theorems of the form \( P \Rightarrow Q \) and \( \neg P \Rightarrow Q \) and derives \( Q \).

\begin{verbatim}
(define (dm-rewriter P)
  (match P
    ((not (and P1 P2)) (equiv (assume P (!dm P))
        (assume-let (hyp (or (not P1) (not P2)))
            (!dm hyp))))
    ((not (or P1 P2)) (equiv (assume P (!dm P))
        (assume-let (hyp (and (not P1) (not P2)))
            (!dm hyp))))))

(define (bicond-rewriter P)
  (dmatch P
    ((iff P1 P2) (dlet ((L1 (assume P
        (both (assume P1 (!mp (!left-iff P) P1))
            (assume P2 (!mp (!right-iff P) P2)))))
        (L2 (assume-let ((hyp (if P1 P2) (if P2 P1))))
            (equiv (!left-and-hyp) (!right-and-hyp))))
            (!equiv L1 L2)))))

(define (cond-rewriter P)
  (dmatch P
    ((if P1 P2) (dlet ((L1 (assume P
        (dlet ((L (suppose-absurd-let hyp (or (not (or P1) P2))))
            (dlet ((conj ((and (not (not P1)) (not P2)) BY (!dm hyp)))
                (!absurd (!mp P (!left-and-conj))
                    (!right-and-conj)))))
            (!dm L)))))
        (L2 (assume-let ((hyp (or (not P1) P2)))
            (assume P1
                (!ds hyp (!negate-twice P1))))))
            (!equiv L1 L2))))))

(define (dm-rewriter P)
  (dmatch P
    ((not (not Q)) (equiv (assume P (!dn P))
        (assume Q (!negate-twice Q))))))

(define (distrib-rewriter-1 P)
  (dmatch P
    ((or P1 (and P2 P3)) (dlet ((L1 (assume P
        (!cd P (assume P1
            (both (either P1 P2)
                (both P1 P3)))
            (assume-let (hyp (and P2 P3))))
            (both (either P1 (!left-and-hyp))
                (either P1 (right-and-hyp)))))))
        (L2 (assume-let ((hyp (and (or P1 P2) (or P1 P3)))
            (!cases
                (assume P1
                    (either P1 (and P2 P3)))
                (assume (not P1)
                    (either P1 (!both (!ds (!left-and-hyp) (not P1)))
                        (!ds (!right-and-hyp) (not P1)))))
            (!equiv L1 L2))))))))
(define (distrib-rewriter-2 P)
  (dmatch P
    ((or (and P1 P2) P3)
      (dlet ((L1 (assume P
        (and P (assume-let ((Q (and P1 P2)))
          (!both (left-and Q) P3)
          (right-and Q) P3)))
        (assum P3
          (both (left P1 P3)
            (right P2 P3))))
        (L2 (assume-let ((hyp (and (or P1 P2) (or P2 P3)))
            (cases
              (assume P3 (left (and P1 P2) P3))
              (assume (not P3)
                (left (and (left-and hyp) (not P3))
                  (and (left-and hyp) (not P3)))
                P3))))))
      (equiv L1 L2))))
    ((or (P1 P2 P3)) (distrib-rewriter-1 P))
    ((or (and P1 P2) P3) (distrib-rewriter-2 P))))

(define cnf-rewriters [bicond-rewriter cond-rewriter de dn-rewriter distrib-rewriter])

(define (cnf-derive P)
  (fix-rewriters-top-down cnf-rewriters P))

Problem 2

Part A

What makes naive so inefficient is that the input propositions have an enormous amount of redundant structure which naive fails to exploit. For instance, let

\[ P = (\text{make-and-tree \text{true} \text{true}}) = (\text{and} (\text{and} \ \text{true} \ \text{true}) (\text{and} \ \text{true} \ \text{true})). \]

Here naive will derive the proposition true 4 times and the proposition (and true true) twice. As the height of the conjunctions increases, the repetition grows exponentially. In general, for a tree of height \( n \) (i.e., a proposition of the form (make-and-tree true \( n \))), naive will derive true \( 2^n \) times, it will derive (and true true) \( 2^{n-1} \) times, (and (and true true) (and true true)) \( 2^{n-2} \) times, and so on. For instance, for \( n = 16 \), naive will derive (and true true) 32,768 times. By contrast, fast will derive (and true true) only once. The savings become magnified as the height of the tree increases. In fact trm will perform exactly \( n \) applications of both whereas naive will perform \( 2^n \) of them (where \( n \) is the height of the conjunction).

Loosely put, a method call of the form (!trm P M) can be understood as follows: “First derive \( P \), unless it’s already in the assumption base, and then do whatever \( M \) does”. The trick is to use recursion to make “whatever \( M \) does” have the desired effect. Perhaps the most important point in the definition of trm is the dbegin deduction:

\[ \text{(dbegin (!both P1 P2) (!M)).} \]
By Athena's semantics, in a deduction of the form \( \text{dbegin } D_1 D_2 \) the conclusion of \( D_1 \) will be available within \( D_2 \), so in this case the conclusion of \( (\text{!both } P_1 P_2) \) will be available during the evaluation of \( (\text{!M}) \). Now when \( \text{M} \) happens to be a method thunk of the form \( \text{method } () \ (\text{!trm } \cdots) \), this means that the conclusion of \( \text{both} \) will be available during the recursive evaluation of \( (\text{!trm } \cdots) \).

In general, tail-recursive method calls provide the simplest way to “thread” the assumption base monotonically in such a way that intermediate conclusions are retained in the assumption base for future use. When we need to traverse branching structures such as \text{and} trees, which require more than one recursive call in the body of the method, we often take advantage of Athena's lexical scoping and use higher-order method continuations in order to preserve tail recursion. Another illustration of this technique can be found in the solution of the next problem.

Part B

The given solution is inefficient because \text{conjoin} repeatedly calls \text{detach} for every atom in the list \text{leaves}. We thus have to scan \text{P} multiple times in order to detach every single leaf separately. But \text{detach} is linear in the size of \text{P} (its total number of nodes), since even in the average case we still have to examine half of the nodes of \text{P} in order to locate the desired leaf. So, on average, this solution will require at least \( l \cdot \frac{n}{2} \) steps, where \( l \) is the number of leaves and \( n \) the total number of nodes of \text{P}.

Moreover, the given solution does not exploit repetitions: if an atom has multiple leaf occurrences, it will be detached every single time anew.

The trick is to scan \text{P} only once, detaching every subproposition of it along the way, and so that all such subpropositions become retained in the assumption base for subsequent use. Once every leaf of \text{P} has been derived we can call the method \text{fast-conjoin} below, which simply runs through all the leaves and chains them in one big right-associated conjunction:

```
(define (fast-conjoin props)
  (dmatch props
     (P) ((claim P))
     ((list-of rest) (both L ((fast-conjoin rest)))))
```

Note that the only difference is that, in \text{conjoin}, the left argument to \text{both} is a deduction (the method call \( (\text{!detach } L \ P) \)), whereas here it is simply the leaf \text{L} itself. So while \text{conjoin} deductively derives every leaf, here the leaves are already expected to be in the assumption base.

How can we detach the leaves of \text{P} and retain all of them in the assumption base so that they can be later used by \text{fast-conjoin}? The answer again is to thread the assumption base monotonically using tail-recursive method calls. Specifically, we define a method \text{detach-all} that takes a conjunction \text{P} along with a method thunk \text{M}, detaches every single subproposition of \text{P} from top to bottom, and then finally invokes \text{M} in an assumption base that is guaranteed to contain every detached conjunct (hence including the leaves). The only wrinkle is that, because conjunctions are branching tree structures, a single tail recursive call will not suffice; we have to be a bit more clever about scanning \text{P} from top to bottom in a tail-recursive way. We can accomplish that by taking advantage of Athena's lexical scoping to properly augment the given method thunk at each step, directly recursing on the left subtree and stacking up a method continuation for doing the same thing recursively on the right subtree on top of the given thunk:

```
(define (detach-all P thunk)
  (dmatch P
     ((and _ _) (detach-all (!left-and P)
                   (method () (detach-all (!right-and P) thunk))))
     (_ (tthunk))))
```

The solution is now immediate:

```
(define (rassoc-2 P)
  (detach-all P (method () (fast-conjoin (get-leaves P)))))
```
The complexity of this method is \( l + n \), since every node of \( P \) will be visited exactly once during the descent of \texttt{detach-all}, and then every leaf will be also be visited once by \texttt{fast-conjoin}. Therefore, although for certain degenerate cases the previous version of \texttt{rassoc} will be faster (consider a proposition of the form (\texttt{(and ?A ?Q)}, where \( ?A \) is a very large conjunction with \( ?A \) at all the tips), in most cases \texttt{rassoc-2} will perform better.

Note that although \texttt{detach-all} and \texttt{trm} (from problem 2) use similar techniques, the two methods are in a sense inverses of each other, in that they proceed in opposite directions: \texttt{trm} is bottom-up and synthetic, meaning that it builds the entire conjunction starting from the leaves all the way to the top; whereas \texttt{detach-all} is top-down and analytic, meaning that it breaks up the conjunction to smaller and smaller components, starting from the top and ending at the leaves.

Also note that there are several other ways to solve this problem efficiently. Perhaps the most direct solution is the following:

\begin{verbatim}
(define (rassoc-once P)
  (dmatch P
    ((and (and P1 P2) P3)
      (let ((left (rassoc-once P1))
              (right (rassoc-once P3)))
        (both (left-and-right) (both (right-and P))))))

(define (rassoc-3 P)
  (dmatch P
    ((and (and _ _) _) (!rassoc-3 (!rassoc-once P)))
    ((and _ _) (!both (rassoc-3 (left-and P)))
     (rassoc-3 (right-and P)))
    (_ (!claim P))))
\end{verbatim}

This solution, however, has little to do with assumption-base threading and method continuations, which were the theme of this problem.

Finally, it is interesting to observe that the “deductive rewrite” machinery we developed for the CNF problem can be reused here. All we have to do is define an appropriate rewriter:

\begin{verbatim}
(define (rassoc-rewriter P)
  (dmatch P
    ((and (and P1 P2) P3) (!equiv (assume P (!rassoc-once P))
                              (assume-let ((hypo (and P1 (and P2 P3)))
                                           (both (both (left-and hyp)
                                                     (right-and hyp)))
                                           (right-and (right-and hyp))))))
    (_ (!claim true))))

(define (rassoc-4 P)
  (!fix-rewriter-top-down rassoc-rewriter))
\end{verbatim}

This is probably the “easiest” solution because we don’t have to be concerned at all about control, which is already taken care of by \texttt{fix-rewriter-top-down}. All we had to do was define the constant-time \texttt{rassoc-rewriter}, which is a simple exercise in propositional logic. By contrast, both \texttt{rassoc-2} and \texttt{rassoc-3} are control-heavy, with several recursive calls. On the other hand, \texttt{rassoc-4} is much less efficient than \texttt{rassoc-2}. This is a manifestation of a general phenomenon that recurs time and again, both in software and in proof engineering: the tension between easy-to-put-together-but-inefficient solutions that are obtained by combining general-purpose off-the-shelf tools, and custom-made solutions which are more difficult to formulate but also more efficient because they exploit case-specific knowledge.
Problem 3

In this and the following problem we will need numbers, a few classic list-manipulating functions, and a negation function:

(domain Num)

(use-numerals (0 ... ) Num)

(define-numeric-operations)

(define (error msg)
  (begin
    (print (join "\n" msg "\n"))
    (halt)))

(define (map f L)
  (match L
    ([] []
      (list-of x rest) (add (f x) (map f rest)))
    (_ false)))

(define (member? x L)
  (match L
    ([] []
      (list-of (val-of x) _) true)
    (_ false)))

(define (for-some L pred)
  (match L
    ([] false)
    (list-of x rest) (|| (pred x) (for-some rest pred))))

(define (filter L test?)
  (match L
    ([] []
      (list-of x rest) (check ((test? x) (add x (filter rest test?)))
                      (else (filter rest test?))))
    (_ false)))

(define (prefix L1 L2)
  (match [L1 L2]
    ([] []
      (list-of x rest) (list-of x rest2) (prefix rest1 rest2)
                      (_ false)))

(define (nth-element n list)
  (match [n list]
    ([] []
      (error "Invalid position.")
    ([1 (list-of x _)] x)
    ([n (list-of _ rest)] (nth-element (minus n 1) rest))))

(define (remove-duplicates L)
  (match L
    ([] []
      (split L1 (list-of x (split L2 (list-of x L3))) (remove-duplicates (join L1 [x] L2 L3))
                      (_ L)))

We will also need a negation function:

(define (~ b)
  (match b
    ([true false]
      (false true))
    (_ false)))
Part A

(define (prepend i pos-list)
  (map (function (pos) (add i pos)) pos-list))

(define (fold dom-list n accum)
  (match dom-list
    ([[] accum])
    ([list-of dom rest] (fold rest (plus n 1) (join accum (prepend n dom)))))

(define (dom E)
  (match E
    ([[some-symbol root] (some-list children)] (join [] (fold (map dom children) 1 [])))))

Part B

(define (label E pos)
  (match [E pos]
    ([[some-symbol c] (some-list args)] [] c)
    ([[some-symbol c] (some-list args)] (list-of n pos')
      (label (nth-element n args) pos'))))

Problem 4

Part A

By a node of an expression E we will mean a two-element list [l p] consisting of a label l and a position p in (dom E) such that (label E pos) = l. We define free-occurrence? as the complement of bound-occurrence?. The functions bound-occurrences and fvar are given as bound-var-occurrences and free-vars below. Analogous functions free-var-occurrences and bound-vars are also given.

(define (leaf? E)
  (member? (label E []) [NumExp VarExp TrueExp FalseExp]))

(define (nodes E)
  (map (function (pos) [(label E pos) pos]) (dom E)))

(define (var-node? node)
  (match node
    ([lab _] (meta-id? lab))))

(define (bound-occurrence? E node)
  (match node
    ([x pos] (& (var-node? node)
        (equal? (label E pos) x)
        (for-node (dom E)
          (function (pos')
            (& (prefix pos' pos)
              (member? (label E pos') [LamExp FixExp LetExp])
              (equal? x (label E (join pos’ [l]))))))))))

(define (bound-var-occurrences E)
  (filter (nodes E) (function (node)
    (& (var-node? node)
      (bound-occurrence? E node))))
(define (free-occurrence? E node) (~ (bound-occurrence? E node)))

(define (free-var-occurrences E)
  (filter (nodes E) (function (node)
    (# (var-node? node)
      (free-occurrence? E node))))

(define (vars E)
  (remove-duplicates (map head (filter (nodes E) var-node?))))

(define (free-vars E)
  (remove-duplicates (map head (free-var-occurrences E))))

(define (bound-vars E)
  (remove-duplicates (map head (bound-var-occurrences E))))

Part B

(define (morph f)
  (function (E))
  (match E
    (((some-symbol c) (some-list args))
      (check (~ (member? c [PairExp AppExp CondExp LeftExp RightExp])) (f E))
      (else (make-term c (map (morph f) args))))))

(define (naive-sub v rep)
  (function (base)
    (match base
      ((VarExp (val-of v)) rep)
      ((LamExp (val-of v) _) base)
      ((LamExp x E) (LamExp x (naive-sub E v rep)))
      ((LetExp (val-of v) E1 E2) (LetExp v (naive-sub E1 v rep) (naive-sub E2 v rep)))
      ((FixExp (val-of v) _) base)
      ((FixExp x E) (FixExp x (naive-sub E v rep)))
      (_ (check ((leaf? base) base)
        (else (morph (naive-sub v rep)) base))))))

(define (make-fresh-id)
  (string->id (var->string (fresh-var))))

(define (rename E)
  (match E
    ((LamExp x E') (let ((new-id (make-fresh-id)))
      (LamExp new-id (rename ((naive-sub x (VarExp new-id)) E')))))))

(define (rename E)
  (match E
    ((FixExp x E') (let ((new-id (make-fresh-id)))
      (FixExp new-id (rename ((naive-sub x (VarExp new-id)) E')))))))

(define (rename E)
  (match E
    ((LetExp x E1 E2) (let ((new-id (make-fresh-id)))
      (LetExp new-id (rename E1) (rename ((naive-sub x (VarExp new-id)) E2))))))

(define (rename E)
  (match E
    ((leaf? E) E)
    (else (morph rename E))))))

(define (sub-base v rep)
  ((naive-sub v rep) (rename base)))
Part C

(define (left p)
match p
  ((PairExp e _) e)))

(define (right p)
match p
  ((PairExp _ e) e)))

(define (eval E)
match E
  ((VarExp x) (error (join "Error: unbound variable: " (id->string x))))
  ((PairExp E1 E2) (PairExp (eval E1) (eval E2)))
  ((LeftExp E') (left (eval E'))) ((right (eval E')))
  ((AppExp E1 E2) (match (eval E1)
    ((LamExp x E) (eval (sub E x (eval E2))))))
  ((LetExp x E1 E2) (eval (sub E2 x (eval E1))))
  ((FixExp x E') (eval (sub E' x E')))
  ((CondExp E1 E2 E3) (match (eval E1)
    (TrueExp (eval E2))
    (FalseExp (eval E3))))
  ( _ E)))