Structure of Programming Languages, Assignment 2
Due: October 28, Monday, 3:30 pm

Ensure you understand the course policies for assignments.

hw2code.tar, available on the course website, contains several Caml files you need.

This assignment considers a small “language” for ML-style pattern matching with some twists. A “program” is a pattern \( p \) and a value \( v \). If \( p \) “matches” \( v \), then the result is a list of bindings \( b \). Else there is no result.

**Syntax definition:**

\[
\begin{align*}
  v & ::= \text{c} \mid (v, v) \mid s(v) \\
  p & ::= \cdot \mid x \mid c \mid (p, p) \mid s(p) \mid ... (p) \\
  b & ::= \cdot \mid (x, v), b \\
  (c \in \{0, 1, 2, \ldots\}) \\
  (s \text{ any nonempty string of English letters}) \\
  (x \text{ any nonempty string of English letters})
\end{align*}
\]

Values includes constants, pairs, and tagged values. The tag is any string (unlike in ML where type definitions must introduce constructors). Patterns include wildcard, variables, constants, pairs, tagged patterns, and the “descendent” pattern \(...(p)\).

**Informal semantics:**

- Pattern \( \cdot \) matches every value and produces the empty list of bindings (\( \cdot \)).
- Pattern \( x \) matches every value and produces the one-element binding list \((x, v), \cdot\) when matched with \( v \). Note \( x \) can be any variable.
- Pattern \( c \) matches only the value that is the same constant and produces the empty list of bindings.
- Pattern \((p_1, p_2)\) matches only pairs of values and only if \( p_1 \) and \( p_2 \) match the corresponding components of the pair. The result is the two binding lists from the nested matches appended together.
- Pattern \( s(p) \) matches only a tagged value where the tag is the same (i.e., the same string \( s \)) and \( p \) matches the corresponding value. The result is the result of the nested match.
- Pattern \(...(p)\) matches a value \( v \) if \( p \) matches any descendent of \( v \) in the abstract syntax tree, *including \( v \) itself*. Put another (very useful) way, it matches if \( p \) matches \( v \) or \(...(p)\) matches a child of \( v \) in the syntax tree. The result is the result of (any) successful match.
- Assume a pattern does not have any variable more than once; you do not need to check for this.

**Example:** Using the concrete syntax for the parser provided to you (note parentheses are necessary, the pattern and value must be on separate lines, and there can be no line breaks within the pattern or value):

\[
\begin{align*}
  \text{bar}((x, (...((18, z)), \cdot))) \\
  \text{bar}((42, (\text{foo}(17, (18, (0, 20)))), 19))
\end{align*}
\]

The only match produces a binding list where \( x \) maps to 42 and \( z \) maps to \((0, 20)\).

The formal large-step operational semantics for pattern-matching is given below. The judgment has have the form \( (p, v) \Downarrow b \), meaning \( p \) matches \( v \) producing \( b \). If \( p \) does not match \( v \) there must be no derivation for any \( b \). We use \( b_1 \Downarrow b_2 \) for the result of appending \( b_1 \) and \( b_2 \).
Large-step semantics: \((p, v) \Downarrow b\) (using infix @ for list append).

<table>
<thead>
<tr>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
</tr>
</thead>
<tbody>
<tr>
<td>((_, v) \Downarrow) \cdot</td>
<td>((x, v) \Downarrow (x, v)) \cdot</td>
<td>((c, c) \Downarrow) \cdot</td>
<td>((p_1, v_1) \Downarrow b_1)</td>
<td>((p, v) \Downarrow b)</td>
</tr>
<tr>
<td>((p, v) \Downarrow b)</td>
<td>((_, (p_1, v_1)) \Downarrow b)</td>
<td>((_, (p_2, v_2)) \Downarrow b)</td>
<td>((p_1, p_2, (v_1, v_2)) \Downarrow b_1 \oplus b_2)</td>
<td>((s(p), s(v)) \Downarrow b)</td>
</tr>
</tbody>
</table>

Note: You need to “pair up” with another student before starting the last (optional) problem.

Problems

1. Give examples of a \(p\) and \(v\) where multiple \(b\) are possible. That is, show the large-step semantics is nondeterministic.

2. (ML warm-up) Implement `string_of_valu` of type `Ast.valu -> string` for converting values to concrete syntax. Implement `string_of_binding_list` of type `(string * Ast.valu)` list -> string for converting a binding list to a string. The actual string is unimportant; we recommend putting each binding on a separate line and putting a “;” between the variable and the value. Note `print_ans` (provided) uses `string_of_binding_list`.

3. (ML large) Implement `large : Ast.pattern -> Ast.valu -> (string * Ast.valu)` list option
   A couple of rules are included in the implementation, finish the rest. Your code should largely correspond to the large-step inference rules above (hint: match on the pair \((p, v)\) with these differences:
   
   - Return `None` if there is no match and `Some` \(b\) if there is a match with binding-list \(b\). (Some is the constructor that matches `any` type.)
   - You may resolve the nondeterminism however you like, i.e., if there is more than one match your code should just “find one” and return it. You must always produce one if there is one.

4. (Formal small) Give a formal small-step operational semantics for pattern-matching. Your judgment should have the form \(p; v; b \rightarrow p'; v'\) \(b'\). We are “done” when \(p\) is \(\_\). Otherwise, if \(p\) matches \(v\) there should be a rule that simplifies \(p\) or \(v\) or both by turning them into \(p'\) and \(v'\). The binding list \(b'\) is either \(b\) or something added onto \(b\). A result for the “whole program” \(p\) and \(v\) is \(a'\) where \(p; v; \cdot \rightarrow^{*} a; v; b'\).
   If \(p\) and \(v\) do not match, there must be no way to derive \(p; v; \cdot \rightarrow^{*} a; v; b'\). Hints:
   
   - Have a total of 9 rules, 8 of which are axioms. The non-axiom is given below, as well as some of the axioms, define the rest of the axioms.
   - Almost all the axioms produce the same \(b\) they start with.
   - A couple axioms will turn a pattern into \(\_\). This is similar to IMP’s assign rule where we turn an assignment into a skip.

Note: These “hints” are perhaps more for “checking your work” than guiding you.

<table>
<thead>
<tr>
<th>S3</th>
<th>S4</th>
<th>S5</th>
</tr>
</thead>
<tbody>
<tr>
<td>((_, p); (v_1, v_2); b \rightarrow p; v_2; b)</td>
<td>(p_1; v_1; b \rightarrow p_1'; v_1'; b')</td>
<td>((s(p); s(v); b \rightarrow p; v; b)</td>
</tr>
<tr>
<td>((p_1, p_2); (v_1, v_2); b \rightarrow (p_1', p_2); (v_1', v_2); b')</td>
<td>((_, (p_1, p_2); (v_1, v_2); b \rightarrow (p_1', p_2); (v_1', v_2); b')</td>
<td>((_, (p_1, p_2); (v_1, v_2); b \rightarrow (p_1', p_2); (v_1', v_2); b')</td>
</tr>
</tbody>
</table>
5. (ML small) Complete the ML implementation of the small-step semantics by implementing:

```markdown
small_step: Ast.pattern -> Ast.valu -> (string * Ast.valu) list ->
(Ast.pattern * Ast.valu * (string * Ast.valu) list) list
```

What makes the ML version difficult is turning the nondeterministic choice of step-sequences into explicit search, but some of this is done for you: `iter` maintains a stack of program states left to consider; `small_step` takes one state and returns a list of states. Hint: This list could contain 0, 1, 2, or 3 states. Hint: Think about how recursion interacts with multiple next states.

6. (Pseudo-Denotational) In ML, implement `denote`, which takes a pattern and produces a `Ast.valu->(string * Ast.valu) list option`. (For example, `PWild -> (fun v -> Some [])` matches a wildcard and `PVar s -> (fun v -> Some [(s,v)])` matches a variable.) This translation must always terminate and produce an ML function that when run does not use the `Ast.pattern` type. (It does use the `Ast.valu` type.) Hint: Descendent patterns are the most interesting.

7. Challenge problem (optional): (Equivalence Proofs) Find another student. Do one of the following yourself (as with other problems, getting help is fine but it is your work) while the other student does the other. If you cannot find a partner, it is ok to do just one of these.

(a) Prove: If \( p_1; v_1; b_1 \rightarrow^n ; v_2; b_2 \), then there exists a \( b_3 \) such that \( (p_1, v_1) \Downarrow b_3 \).

(b) Prove: If \( (p, v) \Downarrow b \), then for all \( b' \), there exists a \( v' \) such that \( p; v; b \rightarrow^* ; v'; b' @ b \).

After the other student has a “full draft,” review their proof and provide useful feedback. Using the feedback you get, improve your proof. Turn in two copies of your proof; before and after feedback. And turn in the feedback you gave your partner.

**Hints for 7a:**

- The proof structure is similar to a proof we did in class.
- Note you are not asked to prove that \( b_3 \) has any connection to \( b_1 \) or \( b_2 \). Doing so appears much more difficult; it is a challenge problem.
- Use induction on \( n \).
- Use this lemma, which you must prove: If \( p; v; b_1 \rightarrow p'; v'; b_2 \) and \( (p', v') \Downarrow b_3 \), then there exists a \( b'_3 \) such that \( (p, v) \Downarrow b'_3 \).
- The proof for the lemma will have cases for each rule in the small-step semantics and you will show a derivation exists using the large-step semantics.

**Hints for 7b:**

- The proof structure is similar to a proof we did in class.
- Note the theorem uses \( b \) twice; you are proving the small-step semantics produces the same binding list (plus \( b' \)).
- Note the theorem is stronger than what we actually care about (namely that \( p; v; b \rightarrow^* ; v'; b \)). You will need the stronger claim in one case of the proof.
- Use induction on the assumed derivation.
- Use this lemma, which you should prove (it’s not difficult): If \( p; v; b \rightarrow^n p'; v'; b' \), then \( (p, p_2); (v, v_2); b \rightarrow^n (p', p_2); (v', v_2); b' \).
What to turn in:

- Caml source code in a file called `hw2.ml`. Change only the functions you are asked to implement. More helper functions are fine, but the sample solution has none. Do not modify other files.
- Hard-copy (written or typed) answers to nonprogramming problems.
- The name of your “partner.” (for challenge problem only).