CIS 624 Structure of Programming Languages, Assignment 2
Due: Thursday (Week 5)

For this assignment you may optionally work with a partner. Submit a single solution and clearly indicate both partners' names in the solution.

Source code: `hw2code.tar`, available on the course website, contains several OCaml files you need and test pattern programs (filenames begin with test). There is also an executable of the solution on ix.cs.uoregon.edu. For example, in a terminal session on ix, you can check an example with the command

```
/home/users/norris/public/cis624/hw2/hw2 test
```

where test is a file containing a valid pattern program as described in this assignment.

This assignment considers a small language for OCaml-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:

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

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

Informal semantics description:

- Pattern _ (underscore character serving as wildcard) 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):

```
bar((x,(...((18,z)),_)))
bar((42,(foo((17,(18,(0,20)))),19)))
```

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 $b_1 @ b_2$ for the result of appending b_1 and b_2.

Large-step semantics: $(p, v) \Downarrow b$ (using infix @ for list append).

L1
$$\frac{}{(\_, v) \Downarrow \cdot}$$

L2
$$\frac{}{(x, v) \Downarrow (x, v), \cdot}$$

L3
$$\frac{}{(c, c) \Downarrow \cdot}$$

L4
$$\frac{(p_1, v_1) \Downarrow b_1 \qquad (p_2, v_2) \Downarrow b_2}{((p_1, p_2), (v_1, v_2)) \Downarrow b_1 @ b_2}$$

L5
$$\frac{(p, v) \Downarrow b}{(s(p), s(v)) \Downarrow b}$$

L6
$$\frac{(p, v) \Downarrow b}{(...(p), v) \Downarrow b}$$

L7
$$\frac{(...(p), v_1) \Downarrow b}{(...(p), (v_1, v_2)) \Downarrow b}$$

L8
$$\frac{(...(p), v_2) \Downarrow b}{(...(p), (v_1, v_2)) \Downarrow b}$$

L9
$$\frac{(...(p), v) \Downarrow b}{(...(p), s(v)) \Downarrow b}$$

Note: In this assignment you need to "pair up" with another student. Submit a single solution, clearly indicating the names of both students. You can work alone if you prefer, just indicate in your submission that it was individual work. For the electronic portions of the solution, please use your Bitbucket git repository.

Required: Problems 1 - 5 are worth 100 points and have the following point distribution: 1 (10), 2(10), 3(30), 4 (20), 5 (30).

Optional: Problem 6 or Problem 7 (if present) is worth 10 points of extra credit each; only one extra credit problem (the higher score) will be added to your total if you decide to do both (so maximum with extra credit is 110).

Problems

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

2. (OCaml 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. (OCaml 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 $p; v; b \to p'; v'; b'$. We are "done" when p is _ (underscore designating wildcard pattern). 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 b' where $p; v; \cdot \to^* \_; v'; b'$. If p and v do not match, there must be no way to derive $p; v; \cdot \to^* \_; v'; b'$. 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 _ (wildcard). 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.

S1
$$\frac{}{(\_, p); (v_1, v_2); b \to p; v_2; b}$$

S2
$$\frac{p_1; v_1; b \to p_1'; v_1'; b'}{(p_1, p_2); (v_1, v_2); b \to (p_1', p_2); (v_1', v_2); b'}$$

S3
$$\frac{}{s(p); s(v); b \to p; v; b}$$

S4
$$\frac{}{...(p); v; b \to p; v; b}$$

S5
$$\frac{}{...(p); (v_1, v_2); b \to ...(p); v_1; b}$$

5. (OCaml small) Complete the OCaml implementation of the small-step semantics by implementing:

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

The Ast types are defined in the provided ast.ml file. What makes the OCaml 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 (**optional, 10 extra points**)

In OCaml, 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 OCaml function that when run does not use the `Ast.pattern` type. (It does use the `Ast.valu` type.)

What to turn in:

- The semantic rules for problem 4 (only the new ones, no need to copy the provided ones). If submitting through your repository, name the file problem4.pdf (or .txt, etc)
- OCaml 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.
- The name of your partner in hw2.ml.
- **Students working in pairs should submit a single solution using one of the students' repositories.** Both students (if working as a team) should each have a `hw2/team.txt` file containing both names in their respective bitbucket repositories. Students working individually do not need to provide the team.txt file.