exam01-practice-for-2015-soln

midterm practice

This practice exam is similar to previous midterms.
A list of study-topics:

```
data def'ns and racket/Java:
  data-defn for product type (struct/class)  (book, dvd, brick, paddle)
  data-defn for union type                   (stoplight-color, income-tax-entry, directory-response)
  data-defn for product type where one field is another product-type   (bookshelf, world)
  data-defn for union type where one branch is a product-type  (lib-item)
  data-defn for union type where one branch is a product-type  w/ recursion (list-of-X)
In all cases: be able to go from data-def'n to examples to template;
 for a particular function include test-cases, signature/purpose/header/stub,
    copy the template and do inventory-with-values; complete the body.

grammars: definitions: "parse", "ambiguous", "terminal", "non-terminal", "derivation"
          given a grammar and a target string, give a derivation/parse-tree
          Given a description of what we want, come up w/ a grammar rules.
scope
```

(Note that the problems on this study-guide do not comprehensively cover every single topic above.)

- For any question, you may use functions defined in previous questions, even if you didn't complete the previous question.
- Use the suggested abbreviations (in non-gray), to save writing.

1. (15pts) A friend is developing a new website that helps tag the *content* of pictures. Realizing that selfies are the bulk of photos these days, they decide that: A *selfie-info* has two pieces of info: the name of the person, and a description of the background. For example: “ibarland” and “Young Hall”.

   Give a data-definition to represent a selfie-info: a comment including types, along with the corresponding `define-struct`. Give two examples of the data (with different background descriptions); use `define` give a name to each example.

```
; A selfie-info is:
;  (make-slfi [string] [string])
(define-struct slfi (person place))
(define s1 (make-slfi "ibarland" "Young Hall"))
(define s1 (make-slfi "puppuluri" "Eiffel Tower"))
```

2. (10pts) Write the function `selfie-at?`, which takes in a selfie-info and a description of the background, and just returns whether that selfie-info has the given background description.

```
(check-expect ________________________________(selfie-at? s1 "Young Hall")____________________________  ______#
(check-expect ______________________________(selfie-at? s1 "Eiffel Tower")____________________________  ______#f

; selfie-at? : __selfie-info__, __string__ → __boolean__
; Returns  whether `_______s_______`s background descriptions are all equal to `__bkgrnd__`.
;
```

```
(define (selfie-at? s bkgrnd)
  (string=? (slfi-place s) bkgrnd))
```

Be sure to complete the signature, and also complete the description by explicitly mentioning your parameters.

Of course, your friend realizes that not *all* pictures are selfies. In addition, there are also ordinary non-selfies (just a picture of some background), as well as the next big thing: selfies that contain another picture within the photograph (*e.g.* ibarland in front of Young Hall, holding: a photo of puppuluri in front of The Bonnie holding a picture of the Eiffel Tower). We'll call this third type a “meta-picture-info”[1]. That is, for the purposes of your friend's awesome new website, a picture-info is one of three things:

- a background info, which has nothing but a description (*e.g.* “Young Hall”, or “RU students”), or
- a selfie-info (as above), or
- A meta-picture-info which has two things: a selfie-info, and another entire picture-info.

Examples of pictures

A photo of boring background	A selfie of ibarland in front of Young Hall	A meta-picture: dbraffit in front of Young Hall holding the preceding selfie.

These examples are not exhaustive — in particular, a meta-picture can contain *any* of the three types of picture-infos inside of it.

3. (10pts) Give a data-definition to represent meta-picture-info a comment including types, along with the corresponding `define-struct`. Then, provide two examples of meta-picture-infos.

```
; A meta-picture-info is:
; (make-metapict [slfi] [picture])
(define-struct metapict (slf inset))



; Two examples of the data:
(make-metapict s1 "air castles")
(make-metapict s1 s2)
(make-metapict s1 (make-metapict s2 "air castles"))
(make-metapict s1 (make-metapict s2 (make-metapict (make-selfie "Beyonce" "Eiffel Tower") "air castles")))
; Really, 3 would have been a more reasonable minimum, so that you exercise all three options in the 2nd field.

; One common glitch: people started using "background description" as a type-name, which is okay *if* it
; is accompanied by "data definition: a background description is a string."
```

4. (15pts) Write the function `all-at?`, which takes in a picture-info and a background description, and determines whether the picture-info's background descriptions (including any backgrounds nested pictures inside pictures) are equal to the given background description.

```
; all-at? : __picture-info__, __string__ → __boolean__
; Returns  whether `_______p_______`'s background descriptions (including any backgrounds nested
; inside) is equal to `__bkgrnd___`.
;
(define (all-at? p bkgrnd)
  (cond [(string? p) (string=? p bkgrnd)]
        [(slfi? p) (selfie-at? p bkgrnd)]
        [(metapicture? p) (and (selfie-at? (metapicture-slf p) bkgrnd)
                               (all-at? (metapicture-inset p) bkgrnd))]))
```

Be sure to complete the signature, and also complete the description by explicitly mentioning your parameters. Test cases are not required, but you are encouraged to make some to be sure you understand what the your function has to deal with.

5. (15pts) Write the function `photoshop-person-in`, which takes a picture-info and a name, and returns a "doctored" picture-info which is just like the input, except that all names anywhere inside the picture-info have been replaced with the new name.

   For example: if there is a picture of puppuluri in front of The Bonnie holding a selfie of ibarland in front of Porterfield, and we want to photoshop in Beyoncé, we would end up with: a selfie of Beyoncé in front of The Bonnie, holding a selfie of Beyoncé in front of Porterfield.

```
;photoshop-person-in : __string__, __picture__ → __picture__
; Returns a "doctored" picture-info which is just like `_______p_______`,
; except that all names anywhere inside it have been replaced with `__poseur__`.
;
(define (photoshop-person-in poseur p)
  (cond [(string? p) p]
        [(slfi? p) (make-slfi poseur (slfi-place p))]
        [(metapicture? p) (make-metapicture (photoshop-person-in poseur (metapicture-slf s))
                                            (photoshop-person-in poseur (metapicture-inset s)))]))


; Note: if you don't recur on the slfi, you can certainly cuplicate the work
; of branch two in branch three.  In that case, be sure to first extract the
; slfi from the metapicture, then get the  background from the slfi:
;
;       [(metapicture? p) (make-metapicture (make-slfi poseur (slfi-place (metapicture-slf s)))
;                                           (photoshop-person-in poseur (metapicture-inset s)))]))
```

Be sure to include the function signature, and complete the description by explicitly mentioning your parameters. Test cases are not required, but you are encouraged to make some to be sure you understand what the your function has to deal with.

Our new digital assistant, Sorri, occasionaly needs to give random bits of encouragement. We'll have her generate strings that fit the following grammar:

```
S → N CV NP.                        (S: "Sentence")
NP → ART MN                         (NP: "noun phrase")
MN → N | ADJ MN                     (MN: "modified noun")
ART → a | an | the                  (ART: "article")
CV → Vs | Ved | Ves                 (CV: "conjugated verb")
N → RU | student | joy | love | classmate | knowledge
V → enrich | love | inspire
ADJ → clever | elegant | lovely | true
```

(Note: not only are `N`, `V`, `ADJ` different from the sample exam, but `S` is also slightly different, starting with `N`, not `NP`, .)

6. (10pts) Give a parse tree for “`knowledge enriches the elegant true student.`”.

```
                       _______S___________________
                      /       |         \         \
                    N        CV         NP         .
                    |       /  \       /  \
               knowledge   V   es    Art    MN
                           |         |     /  \
                         enrich     the  ADJ   MN
                                          |   /  \
                                     elegant  ADJ  MN
                                               |    |
                                             true   N
                                                    |
                                                  student
```

```
We weren't asked for a (left-most) derivation, and it'd be a lot more to hand-write,
but typing it isn't so bad:


S ⇒ N CV NP .
  ⇒ knowledge CV NP .
  ⇒ knowledge Ves NP .
  ⇒ knowledge enriches NP .
  ⇒ knowledge enriches ART MN .
  ⇒ knowledge enriches the MN .
  ⇒ knowledge enriches the ADJ MN .
  ⇒ knowledge enriches the elegant MN .
  ⇒ knowledge enriches the elegant ADJ MN .
  ⇒ knowledge enriches the elegant true MN .
  ⇒ knowledge enriches the elegant true N .
  ⇒ knowledge enriches the elegant true student .
```

-0.5 for leaving off the period.
-1 for not having `S` as the root of the tree (Some people had no root, and just jumped directly to all of `N CV NP.`.)
-0.5 for repeating the `es`; we don't repeat any of the non-terminals:

```
   CV
  / \
 V  es
 |
enrich
```

7. (10pts) Which of the following are derivable from *S*?
   a. [✓ T] / F?: *S* $\Rightarrow^*$ `knowledge inspires a clever joy.`
   b. T / [✓ F]?: *S* $\Rightarrow^*$ `clever joy inspires knowledge.`
   c. T / [✓ F]?: *S* $\Rightarrow^*$ `student`
   d. [✓ T] / F?: *S* $\Rightarrow^*$ `love loves a clever love.`
   e. [✓ T] / F?: *S* $\Rightarrow^*$ `RU inspireed a true clever true elegant true lovely true true knowledge.`
8. (1pt) Your friend notices that the word “love” can be used both as a subject, verb, and object, in Sorri's grammar. Does this mean that the grammar ambiguous? (Circle one: Yes / No)

   Solution: No.
   A grammar is ambiguous if one string has two different parse trees. But that's not the same as "two nonterminals can each generate the same string".

9. Recall: the scope of identifiers introduced with `let` is just the body of the `let`, while the scope of identifiers introduced with `let*` is the body of the `let` *and* all following right-hand-sides of the `let*`.

   Recall: a variable-use's *binding-occurrence* is the place where that variable is defined.

   - (5pts) For *each* variable-*use* of `a` below, draw an arrow from it to its binding occurrence[2].
   - (10pts) Fill in the blank, with the result of evaluating the expression.

   In all cases, presume we have:

```
(define a 1)
(define b 2)
```

   a.

```
(let {[z 100]
      [a 101]
      }
   (+ a b z))

; evaluates to: ____203_____
```

   In `(+ a b z)`, `a` and `z`'s binding-occurrences are the `let`'s left-hand-sides, and `b`'s binding occurrence is the global `define`.

   b.

```
(let {[z 100]
      [a 101]
      [b (+ a 10)]
      }
   (+ a b z))

; evaluates to: ____212_____
```

   In `[b (+ a 10)]`, `a`'s binding-occurrence is the global `define`. In `(+ a b z)`, all of `a`, `b`, and `z`'s binding-occurrences are the `let`'s left-hand-sides.

   c.

```
(define (foo a)
  (let {[z 100]
        [a 101]
        [b (+ a 10)]
        }
     (+ a b z)))

(foo 1001)

; evaluates to: ____1212____
```

In `[b (+ a 10)]`, `a`'s binding-occurrence is parameter in `(foo a)`. In `(+ a b z)`, all of `a`, `b`, and `z`'s binding-occurrences are the `let`'s left-hand-sides.

d.

```
(let* {[z 100]
       [a 101]
       [b (+ a 10)]
       }
    (+ a b z))

; evaluates to: _____312_____
```

In `[b (+ a 10)]`, `a`'s binding-occurrence is the preceding line `[a 101]`. In `(+ a b z)`, all of `a`, `b`, and `z`'s binding-occurrences are the `let*`'s left-hand-sides.

---

[1] The opening credits of Modern Family use a similar concept of people holding a painting of other people holding …, in a linear fashion with a base-case. ↩

[2] The binding-occurrence itself is not a *use* of the variable. ↩

most recent semester