CS 321: Programming Languages — Homework 6

1 Mutable structs

Structs are pieces of compound data which accessed using *field names*, which map to *field values*.

Add `struct` and `get` forms for records, and also add a `set` form that modifies the value of a record field:

```
<RFAE> = <num>
       | {+ <RFAE> <RFAE>}
       | {- <RFAE> <RFAE>}
       | {fun {<id>} <RFAE>}
       | {<RFAE> <RFAE>}             ;; function application
       | <id>
       | {struct {<id> <RFAE>}*}     ;; all fields must be distinct
       | {get <RFAE> <id>}
       | {set <RFAE> <id> <RFAE>}
       | {seqn <RFAE> <RFAE>}
```

- A `struct` form allocates a new record.
- A `get` form accesses from the record produced by the sub-expression the value of the field named by the identifier.
- A `set` form changes the record produced by the first sub-expression. Specifically, the field named by the identifier becomes the value of the second sub-expression. The original (now clobbered) value must be the result of the `set` expression.

Each of `struct`, `get`, and `set` evaluate all of their sub-expressions before doing the corresponding operations.

Also extend `parse` to support the new operations; note that the fields must be distinct, but that our tests do not supply any `struct` expressions with duplicate fields.

Define the usual `interp-expr`, which takes a (parsed) expression and interprets it with an empty initial deferred substitution and empty initial store, and returns either a number, `'function`, or `'struct` (without returning the store).

Your implementation must not use mutation at the PLAI level. (I.e., no PLAI boxes.)

Some examples (these are just a few to think about; this does not constitute anything near a comprehensive test suite):

```
(test/exn (interp-expr (parse '{struct {z {get {struct {z 0}} y}}}))
          "unknown field")

(test (interp-expr (parse '{{fun {r}
                              {get r x}}
                            {struct {x 1}}}))
      1)

(test (interp-expr (parse '{{fun {r}
                              {seqn
                                {set r x 5}
                                {get r x}}}
                            {struct {x 1}}}))
      5)

(test (interp-expr (parse '{set {struct {x 42}} x 2}))
      42)

(test (interp-expr (parse '{{{{{fun {g}
                                   {fun {s}
                                     {fun {r1}
                                       {fun {r2}
                                         {+ {get r1 b}
                                            {seqn
                                              {{s r1} {g r2}}
                                              {+ {seqn
                                                   {{s r2} {g r1}}
                                                   {get r1 b}}
                                                 {get r2 b}}}}}}}}
                                {fun {r} {get r a}}}            ; g
                              {fun {r} {fun {v} {set r b v}}}} ; s
                             {struct {a 0} {b 2}}}             ; r1
                            {struct {a 3} {b 4}}}))            ; r2
      5)
```

2 Eliminating old bindings

If you used the same strategy we did in class for implementing the store, this program:

```
{with {b {struct {x 1}}}
  {with {f {fun {f}
             {seqn {set b x 2}
                   {f f}}}}
    {f f}}}
```

will (eventually) consume all of the memory available because the store grows without bound. Adjust your implementation so that the store's size is proportional only to the number of `struct` expressions that were evaluated, not to the number of `set` operations.

Alternatively, here's a version of the above program that does not use `with`, but is otherwise the same.

```
{{fun {b} {{fun {f} {f f}}
           {fun {f} {seqn {set b x 2}
                          {f f}}}}}
 {struct {x 1}}}
```

To help you experiment with your interpreter, use this function:

```
; size : any -> number?
; computes a (very rough!) approximation of
; the size a PLAI object takes in memory
(define (size s)
  (cond
    [(struct? s)
     (size (struct->vector s))]
    [(vector? s)
     (for/fold ([tot 0])
               ([ele (in-vector s)])
       (+ tot (size ele)))]
    [(pair? s)
     (+ 1 (size (car s)) (size (cdr s)))]
    [else 1]))
```

Something like this:

```
; interp : RFAE? DefSub? Store? -> Value*Store?
(define (interp a-rfae ds store)
  (printf "size ~a\n" (size store))
  (type-case RFAE a-rfae
    ...))
```

3 Conveniences

To make your life easier when testing your interpreter, you may implement a compiler from an extended language (e.g., including `with` and/or multi-argument functions) to `RFAE`.

We will not be testing it; our tests will only cover the `RFAE` language above. This would be strictly for your convenience.

4 Errors

There are five different kinds of errors that can occur (at runtime) in this language and for each error in the input program, your interpreter must signal an error that includes one of the following phrases:

- `"free identifier"`
- `"expected function"`
- `"expected number"`
- `"expected record"`
- `"unknown field"`

As before, each operation in the language that evaluates its sub-expressions immediately must evaluate them from left to right and must evaluate all of them before checking any error conditions.

Your parser will be supplied only well-formed RFAE programs, according to the grammar above.

5 Handin instructions

Provide a definition of `interp-expr : RFAE -> number or 'function or 'struct`, as above.

Provide a definition of `parse : s-expression -> RFAE`, as above.

Have the 8 rules from the Provost's website (see homework 1 for more details).

Submit your code via Canvas.

Your submission must include your test cases; submissions without test cases will get a grade of 0.