CS 5510 Homework 2

Due: Wednesday, September 9th, 2015 11:59pm

Part 1 — Maximum

Start with the interpreter with environments, and add a `max` operator that takes two numbers and returns the larger of them.

Since you must change the `ExprC` datatype, and since different people may change it in different ways, you must update the `parse` function, which accepts an S-expression and produces an `ExprC` value.

Some examples:

```
(test (interp (parse '{max 1 2})
              mt-env
              (list))
      2)
(test (interp (parse '{max {+ 4 5} {+ 2 3}})
              mt-env
              (list))
      9)
```

Note: The fact that quoting an identifier produces a `symbol` instead of an `s-exp` is inconvenient, because it means that `(parse 'x)` doesn't work. You could use `(parse (symbol->s-exp 'x))`, but a more readable trick is to use a *backquote* insted of a normal quote: ``(parse `x)``. A backquote always produces an `s-exp`. It also allows an escape back out of quoting, but we don't need that feature, yet.

Part 2 — Hiding Variables

Add an `unlet` form that hides the nearest visible binding (if any) of a specified variable, but lets other bindings through. For example,

```
{let {[x 1]}
  {unlet x
    x}}
```

should raise a "free variable" exception, but

```
{let {[x 1]}
  {let {[x 2]}
    {unlet x
      x}}}
```

should produce `1`.

As before, you must update the `parse` function.

Some examples:

```
(test/exn (interp (parse '{let {[x 1]}
                           {unlet x
                            x}})
                  mt-env
                  (list))
          "free variable")
(test (interp (parse '{let {[x 1]}
                        {+ x {unlet x 1}}})
              mt-env
```

```
                (list))
        2)
  (test (interp (parse '{let {[x 1]}
                          {let {[x 2]}
                            {+ x {unlet x x}}}})
                mt-env
                (list))
        3)
  (test (interp (parse '{let {[x 1]}
                          {let {[x 2]}
                            {let {[z 3]}
                              {+ x {unlet x {+ x z}}}}}})
                mt-env
                (list))
        6)
  (test (interp (parse '{f 2})
                mt-env
                (list (parse-fundef '{define {f z}
                                       {let {[z 8]}
                                         {unlet z
                                           z}}})))
        2)
```

Part 3 — Functions that Accept Multiple Arguments

Extend the interpreter to support multiple or zero arguments to a function, and multiple or zero arguments in a function call.

For example,

```
  {define {area w h} {* w h}}
```

defines a function that takes two arguments, while

```
  {define {five} 5}
```

defines a function that takes zero arguments. Similarly,

```
  {area 3 4}
```

calls the function `area` with two arguments, while

```
  {five}
```

calls the function `five` with zero arguments.

At run-time, a new error is now possible: function application with the wrong number of arguments. Your `interp` function should detect the mismatch and report an error that includes the words “wrong arity”.

To support functions with multiple arguments, you’ll have to change `fdC` and `appC` and all tests that use them. When you update the `parse` function, note that `s-exp-match?` supports `...` in a pattern to indicate zero or more repetitions of the preceding pattern. *Beware of putting the multi-argument application pattern too early in* `parse`*, since that pattern is likely to match other forms.* In addition, you’ll need to update the `parse-fundef` function that takes one quoted `define` form and produces a `FunDefC` value.

Some examples:

```
  (test (interp (parse '{f 1 2})
                mt-env
                (list (parse-fundef '{define {f x y} {+ x y}})))
        3)
  (test (interp (parse '{+ {f} {f}})
                mt-env
                (list (parse-fundef '{define {f} 5})))
        10)
  (test/exn (interp (parse '{f 1})
                    mt-env
                    (list (parse-fundef '{define {f x y} {+ x y}})))
            "wrong arity")
```

A function would be ill-defined if two of its argument names were the same. To prevent this problem, your `parse-fundef` function can optionally detect this problem and report a “bad syntax” error. For example, `(parse-fundef '{define {f x x} x})` could report a “bad syntax” error, while `(parse-fundef '{define {f x y} x})` should produce a `FunDefC` value.

Remember that `plai-typed` provides the following useful functions:

- `map`— takes a function and a list, and applies the function to each element in the list, returning a list of results. For example, if `sexps` is a list of S-expressions to parse, `(map parse sexps)` produces a list of `ExprC`s by parsing each S-expression.
- `map2`— like `map`, but takes a function of two arguments followed by two lists.

Last update: Friday, September 4th, 2015
mflatt@cs.utah.edu