Meta Language Specification

Introduction

This specifies the *Meta* language, one of the two possible implementation targets for the CS:3820 Fall 2020 project. *Meta* is based on the functional programming language Scheme. The key novelty of the *Meta* language, derived from Scheme, is its focus on *metaprogramming*, or, to oversimplify, writing programs to write programs.

Artefacts

To complete this project, you will write the following.

A *Meta* Interpreter

First, you will write an interpreter for the *Meta* language. Your interpreter should compile to a stand-alone executable `meta` which, when invoked with a series of *Meta* files, reads, parses, and evaluates the contents of those files in order. For example, an interaction with your executable on Windows might look like the following:

```
 > cabal run meta .\lib\prelude.meta .\lib\define.meta .\lib\arith.meta .\lib\test.meta .\tests\even-odd.meta
 ()
 >
```

The `()` here is the result of evaluating the expression in `even-odd.meta`. The remaining files only contain definitions, and so did not produce any output.

I have provided a skeleton in `src/Main.hs` which reads a list of files from the command line, combines their contents, and passes it to a (currently undefined) `go` function. You do not need to rely on this skeleton, but it is hopefully informative regardless.

You may choose to add additional behavior to your interpreter to make it easier to develop and debug. The presence or absence of additional functionality will not impact your grade.

The *Meta* libraries

Second, you will write a number of libraries to make programming in *Meta* slightly more practical. I have provided some starting points—the file `lib/prelude.meta` defines several basic functions and macros. I have also provided a testing framework—`lib/test.meta`—although this depends on several libraries you will write, and a set of test cases.

The libraries you will write are detailed at the end of this document.

Syntax

The syntax of *Meta* is extremely simple. Expressions come in the following forms:

- *Boolean constants:* The Boolean constants are written `#t` for true, and `#f` for false. In general, *Meta* will consider any non-`#f` value to be “true”.
- *Integer constants:* The integer constants consist of sequences of digits, preceded by an optional `-`. For example: `151`, `2`, `-15`.
- *Symbols:* Symbols consist of any sequence of alphanumeric or symbolic characters, so long as they do not match one of the earlier classes, and do not contain the symbols `(` `)` `[` `]` `,` `'` `.`.
- *Combinations:* Combinations consist of sequences of expressions, separated by white space, and surrounded by parentheses or brackets. For example `(+ 1 2 3)`, `[else #f]`, and `()` are all combinations. The choice of parentheses or brackets is irrelevant as far as the language is concerned. As a special case (explained further below), a combiation may contain a (space-separated) `.` character before its final expression. For example, the following are valid combinations: `(1 . 2)`, `(x y . z)`. The `.` may *not* appear earlier in the combination, and must be preceded by at least one expression. The following are *not* valid combinations: `(. 1)`, `(1 .)`, `(.)`.

And in terms of the fundamentals of *Meta*, that’s it! However, for convenience, we also have a couple of syntactic abbrevations.

- *Quoted expressions:* The expression `'e`, where `e` is any expression, is interpreted as the combination `(quote e)`. For example, `'x` is interpreted as `(quote x)`, and `'()` is interpreted as `(quote ())`. Only a single expression is quoted: `(f 'a 1)` is interpreted as `(f (quote a) 1)`, not as `(f (quote a 1))`.
- *Unquoted expressions:* The expression `,e`, where `e` is any expression, is interpreted as the combination `(unquote e)`. The rules are otherwise identical to those for quoted expressions: `(1 ,x 2)` is interpreted as `(1 (unquote x) 2)`, not as `(1 (unquote x 2))`.

Comments begin with a `;`, and extend to the end of the line.

Examples

The following are examples of valid *Meta* expressions:

```
1
-14
(+ 1 -14)
(lambda (x y) (if x y #f))
(lambda args (if (fst args) '(1 2) '(3 4)))
(foldr + 0 '(1 2 3))
''(1 2 ,x ,,y)
```

The following are not:

```
((x)
(, 1)
'
```

Semantics

Meta is an eagerly-evaluated functional programming language. The three challenges in implementing *Meta* are:

- *Quoting: Meta* includes quoting and unquoting of expressions. For example, the expression `(+ 1 2)` would normally be interpreted as calling function `+` with arguments `1` and `2`, and so would evalute to `3`. *Quoting* that expression would delay its evaluation: the expression `'(+ 1 2)` would evaluate to a three-element list, with the elements being the symbol `+` and the numbers `1` and `2`. *Unquoting* is the reverse: the expression `'(+ 1 ,(+ 2 3))` would evaluate to a three-element list, with the elements being the symbol `+`, the number `1`, and the number `5`—the latter resulting from evaluting the *unquoted* expression `(+ 2 3)`.
- *Macros: Meta* includes macros, or programs which generate programs. This allows *Meta* to have a very small core language, with most language features added by macros. Quoting and unquoting will be particularly useful in defining macros. We will see more examples of macros later in the specification.
- *Scope and recursion: Meta* is a statically-scoped langauge, and includes recursive function definition. These are also the choices made in Haskell, so the intuitive meaning of *Meta* programs should not be surprising to you. However, these will introduce some implementation challenges, to be discussed later in the specification.

Values

The core idea in the semantics of *Meta* is to reduce expressions—like `(+ 1 2)`, or `((lambda (x y) (+ x y) 3 4)`—to values—like `3` or `7`. To make this idea precise, we need to start by defining values themselves.

1. Numbers (`1`, `-14`, `615`) are values.
2. Boolean constants (`#t`,`#f`) are values.
3. Quoted symbols (`'x`, `(quote x)`) are values.
4. *Pairs* `(cons a b)` are values if `a` and `b` are values. The empty list `nil` is a value. We say that a pair value is a list if it is `nil`, or if it is of the form `(cons a b)` and `b` is a list. We will write lists `'(e1 e2 e3)` for `(cons e1 (cons e2 (cons e3 nil)))`. We will write `'(e1 e2 . e3)` for `(cons e1 (cons e2 e3)`
5. Function *closures* `#<function: env args body>` and macro closures `#<macro: env args body>` are values. In each case, `env` is an environment, and `args` must be either:

- A list of symbols `'(x1 x2` … `xn)`
- A "list" of symbols, but with the final element being a variable instead of the empty list. `(x1 x2` … `xn . x)`
- A single symbol `x`.

If `args` does not match one of these cases, then `(lambda args body)` or `(macro args body)` is *ill-formed*, and is *not* a value.

In cases 3 and 4, we used *Meta* syntax to describe the forms of quoted symbols, pairs and lists. You may prefer a different internal representation of values within your implementation. In understanding this specification, you should be careful to distinguish between expressions and the values they may be intended to represent. For example, `(cons (+ 1 2) 3)` is not a value, because `(+ 1 2)` is not a value. However, it is a valid *Meta* expression, and we hope it would evaluate to `(cons 3 3)`, which is a value.

In case 5, we write `#<...>` to denote a value that does not directly correspond to any *Meta* syntax. Function and macro closures are the results of evaluating *Meta* expressions; however, the `env` environment does not correspond to anything that can be represented in *Meta* directly. We will talk more about closures when discussion the `lambda` and `macro` special forms below.

Evaluation

We can now define *Meta's* evaluation process—its algorithm for reducing expressions to values. For evaluation, we have one important piece of context: a mapping of symbols (i.e., variables) to values. I will call this the *environment*. Evaluation may fail. For the purposes of this specification, I will assume that failing evaluation at any point in an expression causes the evaluation of the entire expression to fail.

The first rules of evaluation are relatively simple:

1. Integer and Boolean constants evaluate to themselves. The empty list constant `nil` evaluates to itself.
2. Symbols evaluate to their mapping in the environment; if they are not in the environment, their evaluation fails.
3. Combinations `(e e1 ... en)` evaluate according to the rules below. (This rule does *not* apply to combinations with "dotted tails" `(e e1 e2 ... e[n-1] . en)`.)
4. Anything else is an error.

Combinations

Much of the complexity of evaluation has been deferred to the rules for combinations. Again, there are several core cases and a number of special cases, described later. A combination `(e e1... en)` is evaluated as follows.

1. If `e` is a symbol identifying a *special form* (`define`, `lambda`, `macro`, `quote`, `unquote`, `if`), the combination is evaluated following the rules in the next section.

Otherwise, we evaluate `e` to a value `v` and proceed as follows:

2. If `v` is a function closure `#<function: env args body>`, then:
   a. Evalute each of the arguments `e1`…`en` to values `v1`…`vn`.
   b. Evalute `body` in the environment `env` *extended* with the bindings of `v1`…`vn` to `args`. (See below for details of binding.)
3. If `v` is a macro closure `#<macro: env args body>`, then:
   a. Evaluate `body` in the environment `env` *extended* with the bindings of `v1`…`vn` to `args`. (See below for details of binding.) Note that the arguments are **unevaluated**, giving a new expression `f`.
   b. Evaluate expression `f`.
4. If `e` evaluates to an intrinsic function (`eq?`, `add`, `sub`, `mul`, `div`, `cons`, `fst`, `snd`, `number?`, `pair?`, `list?`, `function?`, `eval`):
   a. Evaluate each of the arguments `e1`…`en` to values `v1`…`vn`
   b. Compute a result according to the rules in the following section.
5. Anything else is an error.

Binding

Recall that the arguments list `args` in a `lambda` or `macro` may have three forms. Suppose that the arguments to a function or macro are `e1`…`en`. Then:

- If `args` is a list `'(x1 x2`…`xm)`, and `m = n` (that is: there are the same number of arguments as parameters), then we associate `x1` to `e1`, `x2` to `e2`, and so forth.
- If `args` is a "list" `'(x1 x2`…`xm . x)`, and `m <= n` (that is: there are at least as many arguments as the initial list of parameters), then we associate `x1` to `e1`, `x2` to `e2`, and so forth until we have associated `xm` to `em`. Any remaining arguments, *as a list*, are associated to `x`. If there are no more arguments, `x` is associated to the empty list `nil`.
- If `args` is a single symbol `x`, then we associate `x` to the arguments `'(e1 e2`…`en)`, as a list.

Examples

We consider several examples of evaluating combinations; in each case, we assume a starting environment `env`:

- The expression `(lambda () 3)` evaluates to `#<function: env () 3>`.

- The expression `((lambda () 3))` is a combination: for the rules above, we have `e` = `(lambda () 3)` and no arguments.
  - We evaluate `e` giving the function closure `#<function: env () 3>`
  - Following rule 2, we then evaluate the arguments. There are none.
  - We bind the arguments to the parameters. There are no arguments and no parameters, so this step trivially succeeds.
  - We then evaluate the body of the function `3`. `3` is already a value, so it evaluates to `3`.

  To sum up: `((lambda () 3))` evaluates to `3`.
- The expression `((lambda (x y) (add x y)) 2 3)` is a combination: for the rules above, we have `e` = `(lambda (x y) (add x y))`, `e1` = `2`, and `e2` = `3`.
  - We evaluate `e`, getting the function closure `#<function: env (x y) (add x y)>`
  - We evalute the arguments: `2` evaluates to `2`, and `3` evaluates to `3`.
  - We bind the arguments to the parameters. We associate `x` to `2` and `y` to `3`.
  - We evaluate the body `(add x y)`. This is a combination: for the rules above, we have `e` = `add`, `e1` = `x`, `e2` = `y`.
    - We evaluate `add`; I will assume that this evalutes to the corresponding intrinsic function.
    - We evalute the arguments. Argument `x` is a symbol, which we look up in the environment to get `2`. Argument `y` is a symbol, which we look up in the environment to get `3`.
    - The intrinsic `add` with arguments `2` and `3` gives `5`.

  To sum up, `((lambda (x y) (add x y)) 2 3)` evaluates to `5`.

More examples are available in the `tests` directory of the project distribution.

Special forms

Special forms are built-in features of *Meta* that don't follow the evaluation rules for functions. We have four special forms in *Meta*: `define` is used at the top level to introduce new macros, functions, and constants; `lambda` and `macro` construct function and macro closures; `quote` and `unquote` implement quoting; and, `if` gives meaning to conditionals. This section describes their evaluation rules.

Closures

The expressions `(lambda args body)` and `(macro args body)` evaluate to functions and macros respectively. The complexity arises in their treatment of the environment. Consider an expression like the following

```
((lambda (x)
   (lambda (y) (+ x y))) 1)
```

This is a combination; in the function position, we have a function `(lambda (x) (lambda (y) (+ x y)))` that takes one argument, and returns a function of one argument. In the argument position, we have the constant `1`. The result is the (the result of evaluating) the function `(lambda (y) (+ x y))`. In this resulting expression, what value do we expect for variable `x`? Hopefully `1`. But this is an artifact of the environment at the time `(lambda (y) (+ x y))` is defined. By the time we get to using this function, `x` may have a different meaning, or no longer have a meaning at all.

To make sure that `x` has the meaning we expect, when we evaluate a function or macro term, we save not just the arguments and body but *also* the environment in which the function or macro was defined. This packaging of a function (or macro) body with its environment is called a *closure*. We'll see much more about closures as we continue our discussion of programming languages in the second half of the semester.

How is the packaged environment used? We've already seen this in rules 2 and 3 for combinations above. When we evaluate the body of a closure, we don't use the environment at the calling site; we switch to the value stored in the closure.

Definitions

The expression `(define name body)` adds an association from `name` to the evaluation of `body` to the environment. This special form should *only* appear at the top-level. It does not return anything (i.e., you should not see any output from your interpreter for each `define`). You do not have to check for `define` only appearing at the top level; no test cases will include invalid placement of `define`s.

The `define` special form is the feature of *Meta* that introduces *recursion*. For example, we could write the `length` function in *Meta* as follows:

```
(define length
  (lambda (lst)
     (if (null? lst)
         0
         (+ 1 (length (snd lst))))))
```

Because *Meta* is call-by-value, the only things that can be recursive (without immediately diverging—that is, looping forever) are functions and macros. However, in combination with the rules for closures above, this introduces a problem. The body of the definition will evaluate to a closure `#<function: env (lst) (if ...)>`, where `env` captures the environment in which the body was evaluated. But, for the recursive call to `length` to work, we need `env` to contain an association from `length` to that closure `#<function: env (lst) (if ...)>`. It seems like we need to implement recursion in terms of recursion!

There are two approaches to untying this knot: one is more straight-forward, but labor intensive; the other relies on mutual recursion in Haskell to implement recursion in *Meta*.

- The more straightforward approach takes two steps. In the first step, we evaluate the body of the definition, with the name bound to an arbitrary value. (You can even make it something that returns an error in Haskell—because *Meta* is call-by-value, any definition like `(define ones (cons 1 ones))` can't return a *Meta* value.) Then, after evaluating the body, replace all occurrences of the initial binding with a new binding that refers to the result of evaluating the body. Even if the body doesn't return a closure, it may *contain* closures, and those will need to be updated. So, if you take this approach, you will have to pass over the entire result value.
- The more elegant approach would rely on *mutual* recursion in Haskell. Consider the following pair of Haskell definitions:

  ```
  evens = 0 : map (1 +) odds
  odds  = 1 : map (1 +) (tail evens)
  ```

  Here, we have defined `evens` in terms of `odds`, and `odds` in terms of (the `tail` of) `evens`. Each of these lists is quite long, but you can look at any finite prefix of either of them. Similarly, when you evaluate the body of a `define`, you can use mutual recursion to evaluate it in an environment that refers to the result of that evaluation. The complexity here arises from the model of evaluation: your reference to the result may have to assume that evaluation succeeded. But, as in the use of an undefined value in the previous case, this is fine: the evaluation of the `define` would fail anyway.

Whichever approach you take, the result should be the same: you should be able to define functions recursively, like the example of `length` above.

Conditionals

The expression `(if cond cons alt)` evaluates to (the result of evaluating) `alt` if `cond` evaluates to `#f`, and evaluates to (the result of evaluating) `cons` otherwise.

Quoting

The `quote` and `unquote` special forms implement quoting. The basic idea of quoting is as follows:

- `(quote 1)` evaluates to `1`, and similarly for the other integer and Boolean constants.
- `(quote x)` evaluates to `(quote x)`, and similarly for other symbols.
- `(quote ())` and `(quote nil)` evaluate to `nil`
- `(quote (cons a b))` evaluates to `(cons (quote a) (quote b))`. For example, consider the term `(quote (1 x 2))`. The argument is equivalent to `(cons 1 (cons x (cons 2 nil)))` Evaluation proceeds as follows:
  - `(quote (cons 1 (cons x (cons 2 nil))))`
  - `(cons (quote 1) (quote (cons x (cons 2 nil))))`
  - `(cons 1 (cons (quote x) (quote (cons 2 nil))))`
  - `(cons 1 (cons (quote x) (cons (quote 2) (quote nil))))`

    - `(cons 1 (cons quote x) (cons 2 nil))`

- `(quote (unquote e))` evaluates to the result of evaluating `e`. For example, consider the term `(quote (x (unquote (add 1 2))))`. The argument is equivalent to `(cons x (cons (unquote (add 1 2)) nil))`. Evaluation proceeds as follows:
    - `(quote (cons x (cons (unquote (add 1 2)) nil)))`
    - `(cons (quote x) (quote (cons (unquote (add 1 2)) nil)))`
    - `(cons (quote x) (cons (quote (unquote (add 1 2))) (quote nil)))`
    - `(cons (quote x) (cons (add 1 2) nil))` and, referring back to an earlier example:
    - `(cons (quote x) (cons 3 nil))`

You may have noticed that there is a case missing from the above description: what happens with `(quote (quote e))`? This is the added complexity which makes the above rules only the basic idea of quoting. Quoting and unquoting *nest* in *Meta*. What this means is that we must restate the previous rules not in terms of a single application of the `quote` special form, but in terms of levels of quotation.

- At quotation level 0, evaluation is normal.
- At quotation level `n > 0`:
    - Integer and Boolean constants evaluate to themselves
    - Symbols `x` evaluate to `(quote x)`
    - `()` and `nil` evaluate to `nil`
    - `(cons a b)` evaluates to `(cons c d)`, where `a` evaluates to `c` and `b` evaluates to `d` (at quotation level `n`)
    - `(quote e)` evalutes to `(quote f)` where `e` evaluates to `f` at quotation level `n + 1`
    - `(unquote e)` evaluates to `f` where `e` evaluates to `f` at quotation level `n - 1`

Intrinsic functions

In addition to the special forms, *Meta* also has a number of *intrinsic* (or built-in) functions. Unlike special forms, these functions obey the normal evaluation rules: all of their arguments are evaluated before the function is applied. However, they provide primitive functionality that otherwise could not be expressed with *Meta* programs. Each intrinisic function is defined for a specific number of arguments; calling it with a different number of arguments should report an error.

- `(eq? e1 e2)`: if `e1` and `e2` evaluate to identical values, and neither contains a function or macro closure, then this function returns `#t`. Otherwise, it returns `#f`.
- `(add e1 e2)`: if `e1` and `e2` evaluate to numbers, this function returns their sum. Otherwise, it should report an error.
- `(sub e1 e2)`: if `e1` and `e2` evaluate to numbers, this function returns their difference. Otherwise, it should report an error.

- `(mul e1 e2)`: if `e1` and `e2` evaluate to numbers, this function returns their product. Otherwise, it should report an error.
- `(div e1 e2)`: if `e1` and `e2` evaluate to numbers, this function returns their integer division (i.e., the floor of their quotient, or whatever the Haskell `div` function gives you). Otherwise, it should report an error.
- `(cons e1 e2)`: returns the pair of (the evaluations of) `e1` and `e2`.
- `(fst e)`: if `e` evaluates to a pair `(cons v1 v2)`, returns `v1`. Otherwise, it should report an error.
- `(snd e)`: if `e` evaluates to a pair `(cons v1 v2)`, returns `v2`. Otherwise, it should report an error.
- `(number? e)`: if `e` evalutes to a numeric value, returns `#t`; otherwise, returns `#f`.
- `(pair? e)`: if `e` evalutes to a pair value, returns `#t`; otherwise, returns `#f`.
- `(list? e)`: if `e` evalutes to a list value, returns `#t`; otherwise, returns `#f`.
- `(function? e)`: if `e` evalutes to a function or macro closure, returns `#t`; otherwise, returns `#f`.
- `(eval e)`: returns the result of evaluting `e`, as if it were an expression. For example, `'(add 1 2)`, because it is quoted, evalutes to the list `(cons (quote add) (cons 1 (cons 2 nil)))`. However, `(eval '(add 1 2))` evaluates to *the evaluation of* `(add 1 2)`, that is, `3`.

Meta Libraries

In addition to your interpreter, you should write the following libraries:

`lib/arith.meta`

Your interpreter should provide basic intrinsic operations on integers—addition, subtraction, &c. This library will wrap them to provide more usable versions. The library defines four functions, as follows:

- `+`: the `+` function takes any number of arguments, including 0. It returns the sum of its arguments. If it received no arguments, it returns `0`.
- `-`: the `-` function takes any number of arguments greater than 0. If given one argument, it returns its negation. If given arguments `e1`, `e2`, …, `en`, it returns (((`e1` - `e2`) - `e3`) … - `en`)
- `*`: the `*` function takes any number of arguments, including 0. It returns the product of its arguments. If it received no arguments, it returns `1`.
- `/`: the `/` function takes any number of arguments greater than 0. If given one argument, it returns its reciprocal. If given arguments `e1`, `e2`, …, `en`, it returns (((`e1` / `e2`) / `e3`) … / `en`) Given that we have limited ourselves to integers, this function is nigh-on useless. I am including it only for consistency, and for (hopefully) some easy test cases to pass.

`lib/bool.meta`

Your interpreter provides a conditional for Boolean values. This library will provide two additional Boolean “functions”; however, to provide the expected short-circuiting behavior, they must be implemented as macros. Recall that in *Meta* all values that are not `#f` are considered to be “true”.

- `and`: the `and` macro takes any number of arguments, including 0. If given 0 arguments, it returns `#t`. If any of its arguments evaluate to `#f`, then it should return `#f`. *It should not evaluate any arguments after the first false argument*. Finally, if none of its arguments are evaluate to `#f`, then it should return the value of the final argument. For example, given

  ```
  (define quarter
    (lambda (x) (and (not (eq? x 0)) (/ x 4))))
  ```

  The expression `(quarter 0)` should evaluate to `#f` and should *not* signal a divide-by-zero error.

- `or`: the `or` macro takes any number of arguments, including 0. If given 0 arguments, it returns `#f`. Otherwise, it returns the *first* argument to evaluate to true, if there is one, and `#f` otherwise. For example, `(or 1 (/ 4 0))` should evaluate to `1`, and should not signal a divide-by-zero error.

- `cond`: writing long `if`-`else` chains is annoying. The `cond` macro provides a more compact representation. An expression like

  ```
  (cond [d1 e1]
        [d2 e2]
        [d3 e3])
  ```

  should evaluate to the first `ei` such that `di` evaluates to true. Optionally, one of the `di` may be the symbol `else`; in that case, if none of the previous `dj` have evaluated to true, the `cond` should evaluate to `ei`. If none of them are true, the `cond` should evaluate to `#f`.

`lib/define.meta`

The `define` special form provides the underpinning of top-level definition in *Meta*, while `lambda` provides local definition. This library will provide several macros to make definition more convenient.

- `defun`: The `defun` macro combines `define` and `lambda`: `(defun (name x y z) body)` is equivalent to `(define name (lambda (x y z) body))`. The `defun` macro should support dotted tails just as in lambdas. For example, `(defun (list . args) args)` is equivalent to `(define list (lambda args args))`.

- `defmacro`: The `defmacro` combines `define` and `macro`; it follows the behavior of `defun`.

- `let*`: `lib/prelude.meta` provides a `let` macro to make local definition more intuitive. However, repeatedly nesting `let`s is still inconvenient. The `let*` macro compresses a series of local

definitions into a single form: `(let* [(x1 e1) (x2 e2) ... (xn en)] e)` is equivalent to `(let [x1 e1] (let [x2 e2] ... (let [xn en] e) ...))`