Coursera Programming Languages Course
Section 6 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

Datatype-Programming Without Datatypes

In ML, we used datatype-bindings to define our own one-of types, including recursive datatypes for tree-based data, such as a little language for arithmetic expressions. A datatype-binding introduces a new type into the static environment, along with constructors for creating data of the type and pattern-matching for using data of the type. Racket, as a dynamically typed language, has nothing directly corresponding to datatype-bindings, but it *does* support the same sort of data definitions and programming.

First, some situations where we need datatypes in ML are simpler in Racket because we can just use dynamic typing to put any kind of data anywhere we want. For example, we know in ML that lists are polymorphic but any particular list must have elements that all have the same type. So we cannot directly build a list that holds "string *or* ints." Instead, we can define a datatype to work around this restriction, as in this example:

```
datatype int_or_string = I of int | S of string

fun funny_sum xs =
    case xs of
        [] => 0
      | (I i)::xs' => i + funny_sum xs'
      | (S s)::xs' => String.size s + funny_sum xs'
```

In Racket, no such work-around is necessary, as we can just write functions that work for lists whose elements are numbers or strings:

```
(define (funny-sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (funny-sum (cdr xs)))]
        [(string? (car xs)) (+ (string-length (car xs)) (funny-sum  (cdr xs)))]
        [#t (error "expected number or string")]))
```

Essential to this approach is that Racket has built-in primitives like `null?`, `number?`, and `string?` for testing the type of data at run-time.

But for recursive datatypes like this ML definition for arithmetic expressions:

```
datatype exp = Const of int | Negate of exp | Add of exp * exp | Multiply of exp * exp
```

adapting our programming idioms to Racket will prove more interesting.

We will first consider an ML function that evaluates things of type `exp`, but this function will have a different return type than similar functions we wrote earlier in the course. We will then consider two different approaches for defining and using this sort of "type" for arithmetic expressions in Racket. We will argue the second approach is better, but the first approach is important for understanding Racket in general and the second approach in particular.

Changing How We Evaluate Our Arithmetic Expression Datatype

The most obvious function to write that takes a value of the ML datatype `exp` defined above is one that evaluates the arithmetic expression and returns the result. Previously we wrote such a function like this:

```
fun eval_exp_old e =
    case e of
        Const i => i
      | Negate e2  => ~ (eval_exp_old e2)
      | Add(e1,e2) => (eval_exp_old e1) + (eval_exp_old e2)
      | Multiply(e1,e2) => (eval_exp_old e1) * (eval_exp_old e2)
```

The type of `eval_exp_old` is `exp -> int`. In particular, the return type is `int`, an ML integer that we can then add, multiply, etc. using ML's arithmetic operators.

For the rest of this module, we will instead write this sort of function to *return an exp*, so the ML type will become `exp -> exp`. The result of a call (including recursive calls) will have the form `Const i` for some int `i`, e.g., `Const 17`. Callers have to check that the kind of `exp` returned is indeed a Const, extract the underlying data (in ML, using pattern-matching), and then themselves use the `Const` constructor as necessary to return an `exp`. For our little arithmetic language, this approach leads to a moderately more complicated program:

```
exception Error of string
fun eval_exp_new e =
    let
        fun get_int e =
            case e of
                Const i => i
              | _ => raise (Error "expected Const result")
```

```
in
    case e of
        Const _ => e (* notice we return the entire exp here *)
      | Negate e2  => Const (~ (get_int (eval_exp_new e2)))
      | Add(e1,e2) => Const ((get_int (eval_exp_new e1)) + (get_int (eval_exp_new e2)))
      | Multiply(e1,e2) => Const ((get_int (eval_exp_new e1)) * (get_int (eval_exp_new e2)))
end
```

This extra complication has little benefit for our simple type `exp`, but we are doing it for a very good reason: Soon we will be defining little languages that have *multiple kinds of results*. Suppose the result of a computation did not have to be a number because it could also be a boolean, a string, a pair, a function closure, etc. Then our `eval_exp` function needs to return some sort of one-of type and using a subset of the possibilities defined by the type `exp` will serve our needs well. Then a case of `eval_exp` like addition will need to check that the recursive results are the right kind of value. If this check does not succeed, then the line of `get_int` above that raises an exception gets evaluated (whereas for our simple example so far, the exception will never get raised).

Recursive Datatypes Via Racket Lists

Before we can write a Racket function analogous to the ML `eval_exp_new` function above, we need to define the arithmetic expressions themselves. We need a way to *construct* constants, negations, additions, and multiplications, a way to *test* what kind of expression we have (e.g., "is it an addition?"), and a way to *access* the pieces (e.g., "get the first subexpression of an addition"). In ML, the datatype binding gave us all this.

In Racket, dynamic typing lets us just use lists to represent any kind of data, including arithmetic expressions. One sufficient idiom is to use the first list element to indicate "what kind of thing it is" and subsequent list elements to hold the underlying data. With this approach, we can just define our own Racket functions for constructing, testing, and accessing:

```
; helper functions for constructing
(define (Const i) (list 'Const i))
(define (Negate e) (list 'Negate e))
(define (Add e1 e2) (list 'Add e1 e2))
(define (Multiply e1 e2) (list 'Multiply e1 e2))
; helper functions for testing
(define (Const? x) (eq? (car x) 'Const))
(define (Negate? x) (eq? (car x) 'Negate))
(define (Add? x) (eq? (car x) 'Add))
(define (Multiply? x) (eq? (car x) 'Multiply))
; helper functions for accessing
(define (Const-int e) (car (cdr e)))
(define (Negate-e e) (car (cdr e)))
(define (Add-e1 e) (car (cdr e)))
(define (Add-e2 e) (car (cdr (cdr e))))
(define (Multiply-e1 e) (car (cdr e)))
(define (Multiply-e2 e) (car (cdr (cdr e))))
```

(As an orthogonal note, we have not seen the syntax `'foo` before. This is a Racket *symbol*. For our purposes here, a symbol `'foo` is a lot like a string `"foo"` in the sense that you can use any sequence of characters,

but symbols and strings are different kinds of things. Comparing whether two symbols are equal is a fast operation, faster than string equality. You can compare symbols with `eq?` whereas you should not use `eq?` for strings. We could have done this example with strings instead, using `equal?` instead of `eq?`.)

We can now write a Racket function to "evaluate" an arithmetic expression. It is directly analogous to the ML version defined in `eval_exp_new`, just using our helper functions instead of datatype constructors and pattern-matching:

```
(define (eval-exp e)
  (cond [(Const? e) e] ; note returning an exp, not a number
        [(Negate? e) (Const (- (Const-int (eval-exp (Negate-e e)))))]
        [(Add? e) (let ([v1 (Const-int (eval-exp (Add-e1 e)))]
                        [v2 (Const-int (eval-exp (Add-e2 e)))])
                    (Const (+ v1 v2)))]
        [(Multiply? e) (let ([v1 (Const-int (eval-exp (Multiply-e1 e)))]
                             [v2 (Const-int (eval-exp (Multiply-e2 e)))])
                         (Const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

Similarly, we can use our helper functions to define arithmetic expressions:

```
(define test-exp (Multiply (Negate (Add (Const 2) (Const 2))) (Const 7)))
(define test-ans (eval-exp test-exp))
```

Notice that `test-ans` is `'(Const -28)`, not `-28`.

Also notice that with dynamic typing there is nothing in the program that defines "what is an arithmetic expression." Only our documentation and comments would indicate how arithmetic expressions are built in terms of constants, negations, additions, and multiplications.

Recursive Datatypes Via Racket's `struct`

The approach above for defining arithmetic expressions is inferior to a second approach we now introduce using the special `struct` construct in Racket. A `struct` definition looks like:

```
(struct foo (bar baz quux) #:transparent)
```

This defines a new "struct" called `foo` that is like an ML constructor. It adds to the environment functions for constructing a foo, testing if something is a foo, and extracting the fields bar, baz, and quux from a foo. The names of these bindings are formed systematically from the constructor name `foo` as follows:

- `foo` is a function that takes three arguments and returns a value that is a foo with a bar field holding the first argument, a baz field holding the second argument, and a quux field holding the third argument.
- `foo?` is a function that takes one argument and returns `#t` for values created by calling `foo` and `#f` for everything else.
- `foo-bar` is a function that takes a foo and returns the contents of the bar field, raising an error if passed anything other than a foo.
- `foo-baz` is a function that takes a foo and returns the contents of the baz field, raising an error if passed anything other than a foo.

- `foo-quux` is a function that takes a foo and returns the contents of the quux field, raising an error if passed anything other than a foo.

There are some useful *attributes* we can include in `struct` definitions to modify their behavior, two of which we discuss here.

First, the `#:transparent` attribute makes the fields and accessor functions visible even outside the module that defines the struct. From a modularity perspective this is questionable style, but it has one big advantage when using DrRacket: It allows the REPL to print struct values with their contents rather than just as an abstract value. For example, with our definition of struct `foo`, the result of `(foo "hi" (+ 3 7) #f)` prints as `(foo "hi" 10 #f)`. Without the `#:transparent` attribute, it would print as `#<foo>`, and every value produced from a call to the `foo` function would print this same way. This feature becomes even more useful for examining values built from recursive uses of structs.

Second, the `#:mutable` attribute makes all fields mutable by also providing mutator functions like `set-foo-bar!`, `set-foo-baz!`, and `set-foo-quux!`. In short, the programmer decides when defining a struct whether the advantages of having mutable fields outweigh the disadvantages. It is also possible to make some fields mutable and some fields immutable.

We can use structs to define a new way to represent arithmetic expressions and a function that evaluates such arithmetic expressions:

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (e1 e2) #:transparent)
(struct multiply (e1 e2) #:transparent)

(define (eval-exp e)
  (cond [(const? e) e] ; note returning an exp, not a number
        [(negate? e) (const (- (const-int (eval-exp (negate-e e)))))]
        [(add? e) (let ([v1 (const-int (eval-exp (add-e1 e)))]
                        [v2 (const-int (eval-exp (add-e2 e)))])
                    (const (+ v1 v2)))]
        [(multiply? e) (let ([v1 (const-int (eval-exp (multiply-e1 e)))]
                             [v2 (const-int (eval-exp (multiply-e2 e)))])
                         (const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

Like with our previous approach, nothing in the language indicates how arithmetic expressions are defined in terms of constants, negations, additions, and multiplications. The structure of this version of `eval-exp` is almost identical to the previous version, just using the functions provided by the struct definitions instead of our own list-processing functions. Defining expressions using the constructor functions is also similar:

```
(define test-exp (multiply (negate (add (const 2) (const 2))) (const 7)))
(define test-ans (eval-exp test-exp))
```

Why the `struct` Approach is Better

Defining structs is *not* syntactic sugar for the list approach we took first. The key distinction is that a struct definition creates a *new type of value*. Given

```
(struct add (e1 e2) #:transparent)
```

the function `add` returns things that cause `add?` to return `#t` and *every other* type-testing function like `number?`, `pair?`, `null?`, `negate?`, and `multiply?` to return `#f`. Similarly, the *only* way to access the `e1` and `e2` fields of an add value is with `add-e1` and `add-e2` — trying to use `car`, `cdr`, `multiply-e1`, etc. is a run-time error. (Conversely, `add-e1` and `add-e2` raise errors for anything that is not an add.)

Notice that our first approach with lists does not have these properties. Something built from the `Add` function we defined *is* a list, so `pair?` returns `#t` for it and we can, despite it being poor style, access the pieces directly with `car` and `cdr`.

So in addition to being more concise, our struct-based approach is superior because it *catches errors sooner*. Using `cdr` or `Multiply-e2` on an addition expression in our arithmetic language is almost surely an error, but our list-based approach sees it as nothing more or less than accessing a list using the Racket primitives for doing so. Similarly, nothing prevents an ill-advised client of our code from writing `(list 'Add "hello")` and yet our list-based `Add?` function would return `#t` given the result list `'(Add "hello")`.

That said, nothing about the struct definitions *as we are using them here* truly enforces invariants. In particular, we would like to ensure the `e1` and `e2` fields of any add expression hold only other arithmetic expressions. Racket has good ways to do that, but we are not studying them here. First, Racket has a *module system* that we can use to expose to clients only parts of a struct definition, so we could hide the constructor function and expose a different function that enforces invariants (much like we did with ML's module system).[1] Second, Racket has a *contract system* that lets programmers define arbitrary functions to use to check properties of struct fields, such as allowing only certain kinds of values to be in the fields.

Finally, we remark that Racket's `struct` is a powerful primitive that *cannot* be described or defined in terms of other things like function definitions or macro definitions. It really creates a new type of data. The feature that the result from `add` causes `add?` to return `#t` but every other type-test to return `#f` is something that no approach in terms of lists, functions, macros, etc. can do. Unless the language gives you a primitive for making new types like this, any other encoding of arithmetic expressions would have to make values that cause *some* other type test such as `pair?` or `procedure?` to return `#t`.

Implementing a Programming Language in General

While this course is mostly about what programming-language features *mean* and not how they are *implemented*, implementing a small programming language is still an invaluable experience. First, one great way to understand the semantics of some features is to have to implement those features, which forces you to think through all possible cases. Second, it dispels the idea that things like higher-order functions or objects are "magic" since we can implement them in terms of simpler features. Third, many programming tasks are analogous to implementing an interpreter for a programming language. For example, processing a structured document like a pdf file and turning it into a rectangle of pixels for displaying is similar to taking an input program and turning it into an answer.

We can describe a typical workflow for a language implementation as follows. First, we take a *string* holding the *concrete syntax* of a program in the language. Typically this string would be the contents of one or more files. The *parser* gives errors if this string is not syntactically well-formed, meaning the string cannot possibly contain a program in the language due to things like misused keywords, misplaced parentheses, etc. If there are no such errors, the parser produces a *tree* that represents the program. This is called the *abstract-syntax tree*, or AST for short. It is a much more convenient representation for the next steps of the

[1] Many people erroneously believe dynamically typed languages cannot enforce modularity like this. Racket's structs, and similar features in other languages, put the lie to this. You do not need abstract types and static typing to enforce ADTs. It suffices to have a way to make new types and then not directly expose the constructors for these types.

language implementation. If our language includes type-checking rules or other reasons that an AST may still not be a legal program, the *type-checker* will use this AST to either produce error messages or not. The AST is then passed to the rest of the implementation.

There are basically two approaches to this rest-of-the-implementation for implementing some programming language B. First, we could write an *interpreter* in another language A that takes programs in B and produces answers. Calling such a program in A an "evaluator for B" or an "executor for B" probably makes more sense, but "interpreter for B" has been standard terminology for decades. Second, we could write a *compiler* in another language A that takes programs in B and produces equivalent programs in some other language C (not *the* language C necessarily) and then uses some pre-existing implementation for C. For compilation, we call B the source language and C the target language. A better term than "compiler" would be "translator" but again the term compiler is ubiquitous. For either the interpreter approach or the compiler approach, we call A, the language in which we are writing the implementation of B, the *metalanguage*.

While there are many "pure" interpreters and compilers, modern systems often combine aspects of each and use multiple levels of interpretation and translation. For example, a typical Java system compiles Java source code into a portable intermediate format. The Java "virtual machine" can then start interpreting code in this format but get better performance by compiling the code further to code that can execute directly on hardware. We can think of the hardware itself as an interpreter written in transistors, yet many modern processors actually have translators in the hardware that convert the binary instructions into smaller simpler instructions right before they are executed. There are many variations and enhancements to even this multi-layered story of running programs, but fundamentally each step is some combination of interpretation or translation.

A one-sentence sermon: *Interpreter versus compiler is a feature of a particular programming-language implementation, not a feature of the programming language.* One of the more annoying and widespread misconceptions in computer science is that there are "compiled languages" such as C and "interpreted languages" such as Racket. This is nonsense: I can write an interpreter for C or a compiler for Racket. (In fact, DrRacket takes a hybrid approach not unlike Java.) There is a long history of C being implemented with compilers and functional languages being implemented with interpreters, but compilers for functional languages have been around for decades. SML/NJ, for example, compiles each module/binding to binary code.

Implementing a Programming Language Inside Another Language

Our `eval-exp` function above for arithmetic expressions is a perfect example of an interpreter for a small programming language. The language here is exactly expressions properly built from the constructors for constant, negation, addition, and multiplication expressions. The definition of "properly" depends on the language; here we mean constants hold numbers and negations/additions/multiplications hold other proper subexpressions. We also need a definition of *values* (i.e., results) for our little language, which again is part of the language definition. Here we mean constants, i.e., the subset of expressions built from the `const` constructor. Then `eval-exp` is an interpreter because it is a function that takes expressions in our language and produces values in our language according to the rules for the semantics to our language. Racket is just the *metalanguage*, the "other" language in which we write our interpreter.

What happened to parsing and type-checking? In short, we skipped them. By using Racket's constructors, we basically wrote our programs directly in terms of abstract-syntax trees, relying on having convenient syntax for writing trees rather than having to make up a syntax and writing a parser. That is, we wrote programs with expressions like:

```
(negate (add (const 2) (const 2)))
```

rather than some sort of string like `"- (2 + 2)"`.

While *embedding* a language like arithmetic-expressions inside another language like Racket might seem inconvenient compared to having special syntax, it has advantages even beyond not needing to write a parser. For example, below we will see how we can use the metalanguage (Racket in this case) to write things that act like macros for our language.

Assumptions and Non-Assumptions About Legal ASTs

There is a subtle distinction between two kinds of "wrong" ASTs in a language like our arithmetic expression language. To make this distinction clearer, let's extend our language with three more kinds of expressions:

```
(struct const (int) #:transparent) ; int should hold a number
(struct negate (e1) #:transparent)  ; e1 should hold an expression
(struct add (e1 e2) #:transparent) ; e1, e2 should hold expressions
(struct multiply (e1 e2) #:transparent) ; e1, e2 should hold expressions
(struct bool (b) #:transparent) ; b should hold #t or #f
(struct if-then-else (e1 e2 e3) #:transparent) ; e1, e2, e3 should hold expressions
(struct eq-num (e1 e2) #:transparent) ; e1, e2 should hold expressions
```

The new features include booleans (either true or false), conditionals, and a construct for comparing two numbers and returning a boolean (true if and only if the numbers are the same). Crucially, the result of evaluating an expression in this language could now be:

- an integer, such as `(const 17)`
- a boolean, such as `(bool true)`
- non-existent because when we try to evaluate the program, we get a "run-time type error" – trying to treat a boolean as a number or vice-versa

In other words, there are now two types of *values* in our language – numbers and booleans – and there are operations that should fail if a subexpression evaluates to the wrong kind of value.

This last possibility is something an interpreter should check for and give an appropriate error message. If evaluating some kind of expression (e.g., addition) requires the result of evaluating subexpressions to have a certain type (e.g., a number like `(const 4)` and not a boolean like `(bool #t)`), then the interpreter should check for this result (e.g., using `const?`) rather than assuming the recursive result has the right type. That way, the error message is appropriate (e.g., "argument to addition is not a number") rather than something in terms of the implementation of the interpreter.

The code posted with the course materials corresponding to these notes has two full interpreters for this language. The first does not include any of this checking while the second, better one does. Calling the first interpreter `eval-exp-wrong` and the second one `eval-exp`, here is just the addition case for both:

```
; eval-exp-wrong
[(add? e)
 (let ([i1 (const-int (eval-exp-wrong (add-e1 e)))]
       [i2 (const-int (eval-exp-wrong (add-e2 e)))])
   (const (+ i1 i2)))]

; eval-exp
[(add? e)
```

```
(let ([v1 (eval-exp (add-e1 e))]
      [v2 (eval-exp (add-e2 e))])
  (if (and (const? v1) (const? v2))
      (const (+ (const-int v1) (const-int v2)))
      (error "add applied to non-number")))]
```

However, `eval-exp` *is* assuming that the expression it is evaluating is a legal AST for the language. It can handle `(add (const 2) (const 2))`, which evaluates to `(const 4)` or `(add (const 2) (bool #f))`, which encounters an error, but it does not gracefully handle `(add #t #f)` or `(add 3 4)`. These are not legal ASTs, according to the rules we have in comments, namely:

- The `int` field of a `const` should hold a Racket number.
- The `b` field of a `bool` should hold a Racket boolean.
- All other fields of expressions should hold other legal ASTs. (Yes, the definition is recursive.)

It is reasonable for an interpreter to *assume* it is given a legal AST, so it is *okay* for it to "just crash" with a strange, implementation-dependent error message if given an illegal AST.

Interpreters for Languages With Variables Need Environments

The biggest thing missing from our arithmetic-expression language is variables. That is why we could just have one recursive function that took an expression and returned a value. As we have known since the very beginning of the course, since expressions can contain variables, evaluating them requires an environment that maps variables to values. So an interpreter for a language with variables needs a recursive helper function that takes an expression and an environment and produces a value.[2]

The representation of the environment is part of the interpreter's implementation in the metalanguage, not part of the abstract syntax of the language. Many representations will suffice and fancy data structures that provide fast access for commonly used variables are appropriate. But for our purposes, ignoring efficiency is okay. Therefore, with Racket as our metalanguage, a simple association list holding pairs of strings (variable names) and values (what the variables are bound to) can suffice.

Given an environment, the interpreter uses it differently in different cases:

- To evaluate a variable expression, it looks up the variable's name (i.e., the string) in the environment.
- To evaluate most subexpressions, such as the subexpressions of an addition operation, the interpreter passes to the recursive calls the same environment that was passed for evaluating the outer expression.
- To evaluate things like the body of a let-expression, the interpreter passes to the recursive call a slightly different environment, such as the environment it was passed with one more binding (i.e., pair of string and value) in it.

To evaluate an entire program, we just call our recursive helper function that takes an environment with the program and a suitable initial environment, such as the empty environment, which has no bindings in it.

[2] In fact, for languages with features like mutation or exceptions, the helper function needs even more parameters.

Implementing Closures

To implement a language with function closures and lexical scope, our interpreter needs to "remember" the environment that "was current" when the function was defined so that it can use this environment *instead of* the caller's environment when the function is called. The "trick" to doing this is rather direct: We can literally create a small data structure called a *closure* that includes the environment along with the function itself. *It is this pair (the closure) that is the result of interpreting a function.* In other words, a function is not a value, a closure is, so the evaluation of a function produces a closure that "remembers" the environment from when we evaluated the function.

We also need to implement function calls. A call has two expressions `e1` and `e2` for what would look like `e1 e2` in ML or `(e1 e2)` in Racket. (We consider here one-argument functions, though the implementation will naturally support currying for simulating multiple argument functions.) We evaluate a call as follows:

- We evaluate `e1` using the current environment. The result should be a closure (else it is a run-time error).
- We evaluate `e2` using the current environment. The result will be the argument to the closure.
- We evaluate the body of the code part of the closure **using the environment part of the closure** extended with the argument of the code part mapping to the argument at the call-site.

In the homework assignment connected to these course materials, there is an additional extension to the environment for a variable that allows the closure to call itself recursively. But the key idea is the same: we extend the environment-stored-with-the-closure to evaluate the closure's function body.

This really is how interpreters implement closures. It is the semantics we learned when we first studied closures, just "coded up" in an interpreter.

Optional: Implementing Closures More Efficiently

It may seem expensive that we store the "whole current environment" in every closure. First, it is not that expensive when environments are association lists since different environments are just extensions of each other and we do not copy lists when we make longer lists with `cons`. (Recall this sharing is a big benefit of not mutating lists, and we do not mutate environments.) Second, in practice we can save space by storing only those parts of the environment that the function body might possibly use. We can look at the function body and see what *free variables* it has (variables used in the function body whose definitions are outside the function body) and the environment we store in the closure needs only these variables. After all, no execution of the closure can ever need to look up a variable from the environment if the function body has no use of the variable. Language implementations *precompute* the free variables of each function before beginning evaluation. They can store the result with each function so that this set of variables is quickly available when building a closure.

Finally, you might wonder how compilers implement closures if the target language does not itself have closures. As part of the translation, function definitions still evaluate to closures that have two parts, code and environment. However, we do not have an interpreter with a "current environment" whenever we get to a variable we need to look up. So instead, we change all the functions in the program to take an *extra argument* (the environment) and change all function calls to *explicitly pass in this extra argument.* Now when we have a closure, the code part will have an extra argument and the caller will pass in the environment part for this argument. The compiler then just needs to translate all uses of free variables to code that uses the extra argument to find the right value. In practice, using good data structures for environments (like arrays) can make these variable lookups very fast (as fast as reading a value from an array).

Defining "Macros" Via Functions in the Metalanguage

When implementing an interpreter or compiler, it is essential to keep separate what is in *the language being implemented* and what is in *the language used for doing the implementation (the metalanguage)*. For example, `eval-exp` is a Racket function that takes an arithmetic-expression-language expression (or whatever language we are implementing) and produces an arithmetic-expression-language value. So for example, an arithmetic-expression-language expression would never include a use of `eval-exp` or a Racket addition expression.

But since we are writing our to-be-evaluated programs in Racket, we can use Racket helper functions to help us create these programs. Doing so is basically defining *macros* for our language using Racket functions as the macro language. Here is an example:

```
(define (double e) ; takes language-implemented syntax and produces language-implemented syntax
   (multiply e (const 2)))
```

Here `double` is a Racket function that takes the syntax for an arithmetic expression and produces the syntax for an arithmetic expression. Calling `double` produces abstract syntax in our language, much like macro expansion. For example, `(negate (double (negate (const 4))))` produces `(negate (multiply (negate (const 4)) (const 2)))`. Notice this "macro" `double` does not evaluate the program in any way: we produce abstract syntax that can then be evaluated, put inside a larger program, etc.

Being able to do this is an advantage of "embedding" our little language inside the Racket metalanguage. The same technique works regardless of the choice of metalanguage. However, this approach does not handle issues related to variable shadowing as well as a real macro system that has hygienic macros.

Here is a different "macro" that is interesting in two ways. First the argument is a *Racket* list of *language-being-implemented* expressions (syntax). Second, the "macro" is recursive, calling itself once for each element in the argument list:

```
(define (list-product es)
  (if (null? es)
      (const 1)
      (multiply (car es) (list-product (cdr es)))))
```