Programming Language
Part B

Coursera - University of Washington

September 6, 2016

Programming language on Coursera, Part B Summary.

Coursera Programming Languages Course
Section 5 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

Switching from ML to Racket

For Part B of Programming Languages, we will use the Racket programming language (instead of ML) and the DrRacket programming environment (instead of SML/NJ and Emacs). Notes on installation and basic usage instructions are on the course website in a different document than this one.

Our focus will remain largely on key programming language constructs. We will "switch" to Racket because some of these concepts shine better in Racket. That said, Racket and ML share many similarities: They are both mostly functional languages (i.e., mutation exists but is discouraged) with closures, anonymous functions, convenient support for lists, no return statements, etc. Seeing these features in a second language should help re-enforce the underlying ideas. One moderate difference is that we will not use pattern matching in Racket.

For us, the most important differences between Racket and ML are:

- Racket does not use a static type system. So it accepts more programs and programmers do not need

to define new types all the time, but most errors do not occur until run time.

- Racket has a very minimalist and uniform syntax.

Racket has many advanced language features, including macros, a module system quite different from ML, quoting/eval, first-class continuations, contracts, and much more. We will have time to cover only a couple of these topics.

The first few topics cover basic Racket programming since we need to introduce Racket before we start using it to study more advanced concepts. We will do this quickly because (a) we have already seen a similar language and (b) The Racket Guide, `http://docs.racket-lang.org/guide/index.html`, and other documentation at `http://racket-lang.org/` are excellent and free.

Racket vs. Scheme

Racket is derived from Scheme, a well-known programming language that has evolved since 1975. (Scheme in turn is derived from LISP, which has evolved since 1958 or so.) The designers of Racket decided in 2010 that they wanted to make enough changes and additions to Scheme that it made more sense to give the result a new name than just consider it a dialect of Scheme. The two languages remain *very* similar with a short list of key differences (how the empty list is written, whether pairs built by cons are mutable, how modules work), a longer list of minor differences, and a longer list of additions that Racket provides.

Overall, Racket is a modern language under active development that has been used to build several "real" (whatever that means) systems. The improvements over Scheme make it a good choice for this course and for real-world programming. However, it is more of a "moving target" — the designers do not feel as bound to historical precedent as they try to make the language and the accompanying DrRacket system better. So details in the course materials are more likely to become outdated.

Getting Started: Definitions, Functions, Lists (and `if`)

The first line of a Racket file (which is also a Racket module) should be

```
#lang racket
```

This is discussed in the installation/usage instructions for the course. These lecture notes will focus instead on the content of the file after this line. A Racket file contains a collection of definitions.

A definition like

```
(define a 3)
```

extends the top-level environment so that `a` is bound to 3. Racket has very lenient rules on what characters can appear in a variable name, and a common convention is hyphens to separate words like `my-favorite-identifier`.

A subsequent definition like

```
(define b (+ a 2))
```

would bind `b` to 5. In general, if we have `(define x e)` where `x` is a variable and `e` is an expression, we evaluate `e` to a value and change the environment so that `x` is bound to that value. Other than the syntax,

this should seem very familiar, although at the end of the lecture we will discuss that, unlike ML, bindings can refer to later bindings in the file. In Racket, *everything* is prefix, such as the addition function used above.

An anonymous function that takes one argument is written `(lambda (x) e)` where the argument is the variable `x` and the body is the expression `e`. So this definition binds a cubing function to `cube1`:

```
(define cube1
  (lambda (x)
    (* x (* x x))))
```

In Racket, different functions really take different numbers of arguments and it is a run-time error to call a function with the wrong number. A three argument function would look like `(lambda (x y z) e)`. However, many functions can take any number of arguments. The multiplication function, `*`, is one of them, so we could have written

```
(define cube2
  (lambda (x)
    (* x x x)))
```

You can consult the Racket documentation to learn how to define your own variable-number-of-arguments functions.

Unlike ML, you can use recursion with anonymous functions because the definition itself is in scope in the function body:

```
(define pow
  (lambda (x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1))))))
```

The above example also used an if-expression, which has the general syntax `(if e1 e2 e3)`. It evaluates `e1`. If the result is `#f` (Racket's constant for false), it evaluates `e3` for the result. If the result is *anything else*, including `#t` (Racket's constant for true), it evaluates `e2` for the result. Notice how this is much more flexible type-wise than anything in ML.

There is a very common form of syntactic sugar you should use for defining functions. It does not use the word `lambda` explicitly:

```
(define (cube3 x)
  (* x x x))
(define (pow x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1)))))
```

This is more like ML's `fun` binding, but in ML `fun` is not just syntactic sugar since it is necessary for recursion.

We can use currying in Racket. After all, Racket's first-class functions are closures like in ML and currying is just a programming idiom.

```
(define pow
  (lambda (x)
    (lambda (y)
      (if (= y 0)
          1
          (* x ((pow x) (- y 1)))))))

(define three-to-the (pow 3))
(define eightyone (three-to-the 4))
(define sixteen ((pow 2) 4))
```

Because Racket's multi-argument functions really are multi-argument functions (not sugar for something else), currying is not as common. There is no syntactic sugar for calling a curried function: we have to write `((pow 2) 4)` because `(pow 2 4)` calls the one-argument function bound to `pow` with two arguments, which is a run-time error. Racket has added sugar for *defining* a curried function. We could have written:

```
(define ((pow x) y)
   (if (= y 0)
       1
       (* x ((pow x) (- y 1)))))
```

This is a fairly new feature and may not be widely known.

Racket has built-in lists, much like ML, and Racket programs probably use lists even more often in practice than ML programs. We will use built-in functions for building lists, extracting parts, and seeing if lists are empty. The function names `car` and `cdr` are a historical accident.

Primitive	Description	Example
`null`	The empty list	`null`
`cons`	Construct a list	`(cons 2 (cons 3 null))`
`car`	Get first element of a list	`(car some-list)`
`cdr`	Get tail of a list	`(cdr some-list)`
`null?`	Return `#t` for the empty-list and `#f` otherwise	`(null? some-value)`

Unlike Scheme, you cannot write `()` for the empty list. You can write `'()`, but we will prefer `null`.

There is also a built-in function `list` for building a list from any number of elements, so you can write `(list 2 3 4)` instead of `(cons 2 (cons 3 (cons 4 null)))`. Lists need not hold elements of the same type, so you can create `(list #t "hi" 14)` without error.

Here are three examples of list-processing functions. `map` and `append` are actually provided by default, so we would not write our own.

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

```
(define (map f xs)
  (if (null? xs)
      null
      (cons (f (car xs)) (map f (cdr xs)))))
```

Syntax and Parentheses

Ignoring a few bells and whistles, Racket has an amazingly simple syntax. Everything in the language is either:

- Some form of *atom*, such as `#t`, `#f`, `34`, `"hi"`, `null`, etc. A particularly important form of atom is an identifier, which can either be a variable (e.g., `x` or `something-like-this!`) or a *special form* such as `define`, `lambda`, `if`, and many more.
- A sequence of things in parentheses `(t1 t2 ... tn)`.

The first thing in a sequence affects what the rest of the sequence means. For example, `(define ...)` means we have a definition and the next thing can be a variable to be defined or a sequence for the sugared version of function definitions.

If the first thing in a sequence is not a special form and the sequence is part of an expression, then we have a function call. Many things in Racket are just functions, such as `+` and `>`.

As a minor note, Racket also allows `[` and `]` in place of `(` and `)` anywhere. As a matter of style, there are a few places we will show where `[...]` is the common preferred option. Racket does *not* allow mismatched parenthesis forms: `(` must be matched by `)` and `[` by `]`. DrRacket makes this easy because if you type `)` to match `[`, it will enter `]` instead.

By "parenthesizing everything" Racket has a syntax that is *unambiguous*. There are never any rules to learn about whether `1+2*3` is `1+(2*3)` or `(1+2)*3` and whether `f x y` is `(f x) y` or `f (x y)`. It makes *parsing*, converting the program text into a tree representing the program structure, trivial. Notice that XML-based languages like HTML take the same approach. In HTML, an "open parenthesis" looks like `<foo>` and the matching close-parenthesis looks like `</foo>`.

For some reason, HTML is only rarely criticized for being littered with parentheses but it is a common complaint leveled against LISP, Scheme, and Racket. If you stop a programmer on the street and ask him or her about these languages, they may well say something about "all those parentheses." This is a bizarre obsession: people who use these languages quickly get used to it and find the uniform syntax pleasant. For example, it makes it very easy for the editor to indent your code properly.

From the standpoint of learning about programming languages and fundamental programming constructs, you should recognize a strong opinion about parentheses (either for or against) as a syntactic prejudice. While everyone is entitled to a personal opinion on syntax, one should not allow it to keep you from learning advanced ideas that Racket does well, like hygienic macros or abstract datatypes in a dynamically typed language or first-class continuations. An analogy would be if a student of European history did not want to learn about the French Revolution because he or she was not attracted to people with french accents.

All that said, practical programming in Racket does require you to get your parentheses correct and Racket differs from ML, Java, C, etc. in an important regard: *Parentheses change the meaning of your program. You cannot add or remove them because you feel like it. They are never optional or meaningless.*

In expressions, `(e)` means evaluate `e` and then call the resulting function with 0 arguments. So `(42)` will be a run-time error: you are treating the number 42 as a function. Similarly, `((+ 20 22))` is an error for the

same reason.

Programmers new to Racket sometimes struggle with remembering that parentheses matter and determining why programs fail, often at run-time, when they are misparenthesized. As an example consider these seven definitions. The first is a correct implementation of factorial and the others are wrong:

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))   ; 1
(define (fact n) (if (= n 0) (1) (* n (fact (- n 1))))) ; 2
(define (fact n) (if = n 0 1 (* n (fact (- n 1)))))     ; 3
(define fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))   ; 4
(define (fact n) (if (= n 0) 1 (* n fact (- n 1))))     ; 5
(define (fact n) (if (= n 0) 1 (* n ((fact) (- n 1))))) ; 6
(define (fact n) (if (= n 0) 1 (n * (fact (- n 1)))))   ; 7
```

Line	Error
2	calls `1` as a function taking no arguments
3	uses `if` with 5 subexpressions instead of 3
4	bad definition syntax: `(n)` looks like an expression followed by more stuff
5	calls `*` with a function as one of the arguments
6	calls `fact` with 0 arguments
7	treats `n` as a function and calls it with `*`

Dynamic Typing (and `cond`)

Racket does not use a static type system to reject programs before they are run. As an extreme example, the function `(lambda () (1 2))` is a perfectly fine zero-argument function that will cause an error if you ever call it. We will spend significant time in a later lecture comparing dynamic and static typing and their relative benefits, but for now we want to get used to dynamic typing.

As an example, suppose we want to have lists of numbers but where some of the elements can actually be other lists that themselves contain numbers or other lists and so on, any number of levels deep. Racket allows this directly, e.g., `(list 2 (list 4 5) (list (list 1 2) (list 6)) 19 (list 14 0))`. In ML, such an expression would not type-check; we would need to create our own datatype binding and use the correct constructors in the correct places.

Now in Racket suppose we wanted to compute something over such lists. Again this is no problem. For example, here we define a function to sum all the numbers anywhere in such a data structure:

```
(define (sum xs)
  (if (null? xs)
      0
      (if (number? (car xs))
          (+ (car xs) (sum (cdr xs)))
          (+ (sum (car xs)) (sum (cdr xs))))))
```

This code simply uses the built-in *predicates* for empty-lists (`null?`) and numbers (`number?`). The last line assumes `(car xs)` is a list; if it is not, then the function is being misused and we will get a run-time error.

We now digress to introduce the `cond` special form, which is better style for nested conditionals than actually using multiple if-expressions. We can rewrite the previous function as:

```
(define (sum xs)
```

```
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (sum (cdr xs)))]
        [#t (+ (sum (car xs)) (sum (cdr xs)))]))
```

A `cond` just has any number of parenthesized pairs of expressions, `[e1 e2]`. The first is a test; if it evaluates to `#f` we skip to the next branch. Otherwise we evaluate `e2` and that is the answer. As a matter of style, your last branch should have the test `#t`, so you do not "fall off the bottom" in which case the result is some sort of "void object" that you do not want to deal with.

As with `if`, the result of a test does not have to be `#t` or `#f`. Anything other than `#f` is interpreted as true for the purpose of the test. It is sometimes bad style to exploit this feature, but it can be useful.

Now let us take dynamic typing one step further and change the specification for our `sum` function. Suppose we even want to allow non-numbers and non-lists in our lists in which case we just want to "ignore" such elements by adding 0 to the sum. If this is what you want (and it may not be — it could silently hide mistakes in your program), then we can do that in Racket. This code will never raise an error unless the initial argument was neither a number nor a list:

```
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? xs) xs]
        [(list? (car xs)) (+ (sum (car xs)) (sum (cdr xs)))]
        [#t (sum (cdr xs))]))
```

Local bindings: `let`, `let*`, `letrec`, local `define`

For all the usual reasons, we need to be able to define local variables inside functions. Like ML, there are expression forms that we can use anywhere to do this. Unlike ML, instead of one construct for local bindings, there are four. This variety is good: Different ones are convenient in different situations and using the most natural one communicates to anyone reading your code something useful about how the local bindings are related to each other. This variety will also help us learn about scope and environments rather than just accepting that there can only be one kind of let-expression with one semantics. How variables are looked up in an environment is a fundamental feature of a programming language.

First, there is the expression of the form

```
(let ([x1 e1]
      [x2 e2]
      ...
      [xn en])
  e)
```

As you might expect, this creates local variables `x1`, `x2`, ... `xn`, bound to the results of evaluating `e1`, `e2`, ..., `en`. and then the body `e` can use these variables (i.e., they are in the environment) and the result of `e` is the overall result. Syntactically, notice the "extra" parentheses around the collection of bindings and the common style of where we use square parentheses.

But the description above left one thing out: What environment do we use to evaluate `e1`, `e2`, ..., `en`? It turns out we use the environment from "*before*" the let-expression. That is, later variables do *not* have earlier ones in their environment. If `e3` uses `x1` or `x2`, that would either be an error or would mean some *outer* variable of the same name. This is *not* how ML let-expressions work. As a silly example, this function doubles its argument:

```
(define (silly-double x)
  (let ([x (+ x 3)]
        [y (+ x 2)])
    (+ x y -5)))
```

This behavior is sometimes useful. For example, to swap the meaning of `x` and `y` in some local scope you can write `(let ([x y][y x]) ...)`. More often, one uses `let` where this semantics versus "each binding has the previous ones in its environment" does not matter: it communicates that the expressions are independent of each other.

If we write `let*` in place of let, then the semantics *does* evaluate each binding's expression in the environment produced from the previous ones. This *is* how ML let-expressions work. It is often convenient: If we only had "regular" `let`, we would have to nest let-expressions inside each other so that each later binding was in the body of the outer let-expressions. (We would have use n nested `let` expressions each with 1 binding instead of 1 `let*` with n bindings.) Here is an example using `let*`:

```
(define (silly-double x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```

As indicated above, it is common style to use `let` instead of `let*` when this difference in semantics is irrelevant.

Neither `let` nor `let*` allows recursion since the `e1`, `e2`, ..., `en` cannot refer to the binding being defined or any later ones. To do so, we have a third variant `letrec`, which lets us write:

```
(define (triple x)
  (letrec ([y (+ x 2)]
           [f (lambda (z) (+ z y w x))]
           [w (+ x 7)])
    (f -9)))
```

One typically uses `letrec` to define one or more (mutually) recursive functions, such as this very slow method for taking a non-negative number mod 2:

```
(define (mod2 x)
  (letrec
      ([even? (lambda (x) (if (zero? x) #t (odd? (- x 1))))]
       [odd?  (lambda (x) (if (zero? x) #f (even? (- x 1))))])
    (if (even? x) 0 1)))
```

Alternately, you can get the same behavior as `letrec` by using local defines, which is very common in real Racket code and is in fact the preferred style over let-expressions. In this course, you can use it if you like but do not have to. There are restrictions on where local defines can appear; at the beginning of a function body is one common place where they are allowed.

```
(define (mod2_b x)
  (define even? (lambda(x) (if (zero? x) #t (odd? (- x 1)))))
  (define odd?  (lambda(x) (if (zero? x) #f (even? (- x 1)))))
  (if (even? x) 0 1))
```

We need to be careful with `letrec` and local definitions: They allow code to refer to variables that are initialized *later*, but the expressions for each binding are still evaluated in order.

For mutually recursive functions, this is never a problem: In the examples above, the definition of `even?` refers to the definition of `odd?` even though the expression bound to `odd?` has not yet been evaluated. This is okay because the use in `even?` is in a function body, so it will not be *used* until after `odd?` has been initialized. In contrast, this use of `letrec` is bad:

```
(define (bad-letrec x)
  (letrec ([y z]
           [z 13])
    (if x y z)))
```

The semantics for `letrec` requires that the use of `z` for initializing `y` refers to the `z` in the `letrec`, but the expression for `z` (the `13`) has not been evaluated yet. In this situation, Racket will raise an error when `bad-letrec` is called. (Prior to Racket Version 6.1, it would instead bind `y` to a special "undefined" object, which almost always just had the effect of hiding a bug.)

For this class, you can decide whether to use local defines or not. The lecture materials generally will not, choosing instead whichever of `let`, `let*`, or `letrec` is most convenient and communicates best. But you are welcome to use local defines, with those "next to each other" behaving like `letrec` bindings.

Top-Level Definitions

A Racket file is a module with a sequence of definitions. Just as with let-expressions, it matters greatly to the semantics what environment is used for what definitions. In ML, a file was like an implicit `let*`. In Racket, it is basically like an implicit `letrec`. This is convenient because it lets you order your functions however you like in a module. For example, you do not need to place mutually recursive functions next to each other or use special syntax. On the other hand, there are some new "gotchas" to be aware of:

- You cannot have two bindings use the same variable. This makes no sense: which one would a use of the variable use? With `letrec`-like semantics, we do *not* have one variable shadow another one if they are defined in the same collection of mutually-recursive bindings.
- If an earlier binding uses a later one, it needs to do so in a function body so that the later binding is initialized by the time of the use. In Racket, the "bad" situation of using an uninitialized value causes an error when you use the module (e.g., when you click "Run" for the file in DrRacket).
- So *within* a module/file, there is no top-level shadowing (you can still shadow within a definition or let-expressions), but one module can shadow a binding in another file, such as the files implicitly included from Racket's standard library. For example, although it would be bad style, we could shadow the built-in `list` function with our own. Our own function could even be recursive and call itself like any other recursive function. *However*, the behavior in the REPL is different, so do not shadow a function with your own recursive function definition in the REPL. Defining the recursive function in the Definitions Window and using it in the REPL still works as expected.

Bindings are Generally Mutable: `set!` Exists

While Racket encourages a functional-programming style with liberal use of closures and avoiding side effects, the truth is it has assignment statements. If `x` is in your environment, then `(set! x 13)` will *mutate* the

binding so that `x` now maps to the value 13. Doing so affects all code that has this `x` in its environment. Pronounced "set-bang," the exclamation point is a convention to alert readers of your code that side effects are occurring that may affect other code. Here is an example:

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define z (f 4))
(define w c)
```

After evaluating this program, `z` is bound to 9 because the body of the function bound to `f` will, when evaluated, look up `b` and find `5`. However, `w` is bound to `7` because when we evaluated `(define c (+ b 4))`, we found `b` was `3` and, as usual, the result is to bind `c` to `7` regardless of how we got the `7`. So when we evaluate `(define w c)`, we get `7`; it is irrelevant that `b` has changed.

You can also use `set!` for local variables and the same sort of reasoning applies: you have to think about *when* you look up a variable to determine what value you get. But programmers used to languages with assignment statements are all too used to that.

Mutating top-level bindings is particularly worrisome because we may not know all the code that is using the definition. For example, our function `f` above uses `b` and could act strangely, even fail mysteriously, if `b` is mutated to hold an unexpected value. If `f` needed to defend against this possibility it would need to avoid using `b` after `b` might change. There is a general technique in software development you should know: *If something might get mutated and you need the old value, make a copy before the mutation can occur.* In Racket, we could code this up easily enough:

```
(define f
   (let ([b b])
      (lambda (x) (* 1 (+ x b)))))
```

This code makes the `b` in the function body refer to a local `b` that is initialized to the global `b`.

But is this as defensive as we need to be? Since `*` and `+` are just variables bound to functions, we might want to defend against them being mutated later as well:

```
(define f
   (let ([b b]
         [+ +]
         [* *])
      (lambda (x) (* 1 (+ x b)))))
```

Matters would get worse if `f` used other helper functions: Making local copies of variables bound to the functions would not be enough unless those functions made copies of all their helper functions as well.

Fortunately, none of this is necessary in Racket due to a reasonable compromise: A top-level binding is not mutable unless the module that defined it contains a `set!` for it. So if the file containing `(define b 4)` did not have a `set!` that changed it, then we can rest assured that no other file will be allowed to use `set!` on that binding (it will cause an error). And all the predefined functions like `+` and `*` are in a module that does not use `set!` for them, so they also cannot be mutated. (In Scheme, all top-level bindings really are mutable, but programmers typically just assume they won't be mutated since it is too painful to assume otherwise.)

So the previous discussion is *not* something that will affect most of your Racket programming, but it is useful to understand what `set!` means and how to defend against mutation by making a copy. The point is that the possibility of mutation, which Racket often avoids, makes it very difficult to write correct code.

The Truth about `cons`

So far, we have used `cons`, `null`, `car`, `cdr`, and `null?` to create and access lists. For example, `(cons 14 (cons #t null))` makes the list `'(14 #t)` where the quote-character shows this is printing a list value, not indicating an (erroneous) function call to 14.

But the truth is *cons just makes a pair* where you get the first part with `car` and the second part with `cdr`. Such pairs are often called *cons cells* in languages like Racket. So we can write `(cons (+ 7 7) #t)` to produce the pair `'(14 . #t)` where the period shows that this is *not* a list. A list is, by convention and according to the `list?` predefined function, either `null` or a pair where the cdr (i.e., second component) is a list. A cons cell that is not a list is often called an *improper list*, especially if it has nested cons cells in the second position, e.g., `(cons 1 (cons 2 (cons 3 4)))` where the result prints as `'(1 2 3 . 4)`.

Most list functions like `length` will give a run-time error if passed an improper list. On the other hand, the built-in `pair?` primitive returns true for anything built with `cons`, i.e., any improper or proper list *except* the empty list.

What are improper lists good for? The real point is that pairs are a generally useful way to build an each-of type, i.e., something with multiple pieces. And in a dynamically typed language, all you need for lists are pairs and some way to recognize the end of the list, which by convention Racket uses the `null` constant (which prints as `'()`) for. As a matter of style, you should use proper lists and not improper lists for collections that could have any number of elements.

Cons cells are immutable, but there is `mcons`

Cons cells are immutable: When you create a cons cell, its two fields are initialized and will never change. (This is a major difference between Racket and Scheme.) Hence we can continue to enjoy the benefits of knowing that cons cells cannot be mutated by other code in our program. It has another somewhat subtle advantage: The Racket implementation can be clever enough to make `list?` a constant-time operation since it can store with every cons cell whether or not it is a proper list when the cons cell is created. This cannot work if cons cells are mutable because a mutation far down the list could turn it into an improper list.

It is a bit subtle to realize that cons cells really are immutable even though we have `set!`. Consider this code:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

The `set!` of `x` changes the contents of the binding of `x` to be a different pair; it does not alter the contents of the old pair that `x` referred to. You might try to do something like `(set! (car x) 27)`, but this is a syntax error: `set!` requires a variable to assign to, not some other kind of location.

If we want mutable pairs, though, Racket is happy to oblige with a different set of primitives:

- `mcons` makes a mutable pair

- `mcar` returns the first component of a mutable pair
- `mcdr` returns the second component of a mutable pair
- `mpair?` returns `#t` if given a mutable pair
- `set-mcar!` takes a mutable pair and an expression and changes the first component to be the result of the expression
- `set-mcdr!` takes a mutable pair and an expression and changes the second component to be the result of the expression

Since some of the powerful idioms we will study next use mutation to store previously computed results, we will find mutable pairs useful.

Introduction to Delayed Evaluation and Thunks

A key semantic issue for a language construct is *when are its subexpressions evaluated*. For example, in Racket (and similarly in ML and most but not all programming languages), given `(e1 e2 ... en)` we evaluate the function arguments `e2`, ..., `en` once before we execute the function body and given a function `(lambda (...) ...)` we do not evaluate the body until the function is called. We can contrast this rule ("evaluate arguments in advance") with how `(if e1 e2 e3)` works: we do *not* evaluate both `e2` and `e3`. This is why:

```
(define (my-if-bad x y z) (if x y z))
```

is a function that *cannot* be used wherever you use an if-expression; the rules for evaluating subexpressions are fundamentally different. For example, this function would never terminate since every call makes a recursive call:

```
(define (factorial-wrong x)
  (my-if-bad (= x 0)
             1
             (* x (factorial-wrong (- x 1)))))
```

However, we can use the fact that function bodies are not evaluated until the function gets called to make a more useful version of an "if function":

```
(define (my-if x y z) (if x (y) (z)))
```

Now wherever we would write `(if e1 e2 e3)` we could instead write `(my-if e1 (lambda () e2) (lambda () e3))`. The body of `my-if` either calls the zero-argument function bound to `y` or the zero-argument function bound to `z`. So this function is correct (for non-negative arguments):

```
(define (factorial x)
  (my-if (= x 0)
         (lambda () 1)
         (lambda () (* x (factorial (- x 1))))))
```

Though there is certainly no reason to wrap Racket's "if" in this way, the general idiom of using a zero-argument function to *delay evaluation* (do not evaluate the expression now, do it later when/if the zero-argument function is called) is very powerful. As convenient terminology/jargon, when we use a zero-argument function to delay evaluation we call the function a *thunk*. You can even say, "thunk the argument" to mean "use `(lambda () e)` instead of `e`".

Using thunks is a powerful programming idiom. It is not specific to Racket — we could have studied such programming just as well in ML.

Lazy Evaluation with Delay and Force

Suppose we have a large computation that we know how to perform but we do not know if we need to perform it. Other parts of the program know where the result of the computation is needed and there may be 0, 1, or more different places. If we thunk, then we may repeat the large computation many times. But if we do not thunk, then we will perform the large computation even if we do not need to. To get the "best of both worlds," we can use a programming idiom known by a few different (and perhaps technically slightly different) names: lazy-evaluation, call-by-need, promises. The idea is to use mutation to remember the result from the first time we use the thunk so that we do not need to use the thunk again.

One simple implementation in Racket would be:

```
(define (my-delay f)
  (mcons #f f))

(define (my-force th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

We can create a thunk `f` and pass it to `my-delay`. This returns a pair where the first field indicates we have not used the thunk yet. Then `my-force`, if it sees the thunk has not been used yet, uses it and then uses mutation to change the pair to hold the result of using the thunk. That way, any future calls to `my-force` with the same pair will not repeat the computation. Ironically, while we are using mutation in our *implementation*, this idiom is quite error-prone if the thunk passed to `my-delay` has side effects or relies on mutable data, since those effects will occur at most once and it may be difficult to determine when the first call to `my-force` will occur.

Consider this silly example where we want to multiply the result of two expressions `e1` and `e2` using a recursive algorithm (of course you would really just use `*` and this algorithm does not work if `e1` produces a negative number):

```
(define (my-mult x y)
 (cond [(= x 0) 0]
       [(= x 1) y]
       [#t (+ y (my-mult (- x 1) y))]))
```

Now calling `(my-mult e1 e2)` evaluates `e1` and `e2` once each and then does 0 or more additions. But what if `e1` evaluates to 0 and `e2` takes a long time to compute? Then evaluating `e2` was wasteful. So we could thunk it:

```
(define (my-mult x y-thunk)
 (cond [(= x 0) 0]
       [(= x 1) (y-thunk)]
       [#t (+ (y-thunk) (my-mult (- x 1) y-thunk))]))
```

Now we would call `(my-mult e1 (lambda () e2))`. This works great if `e1` evaluates to 0, fine if `e1` evaluates to 1, and terribly if `e1` evaluates to a large number. After all, now we evaluate `e2` on every recursive call. So let's use `my-delay` and `my-force` to get the best of both worlds:

```
(my-mult e1 (let ([x (my-delay (lambda () e2))]) (lambda () (my-force x))))
```

Notice we create the delayed computation once before calling `my-mult`, then the first time the thunk passed to `my-mult` is called, `my-force` will evaluate `e2` and remember the result for future calls to `my-force x`. An alternate approach that might look simpler is to rewrite `my-mult` to expect a result from `my-delay` rather than an arbitrary thunk:

```
(define (my-mult x y-promise)
 (cond [(= x 0) 0]
       [(= x 1) (my-force y-promise)]
       [#t (+ (my-force y-promise) (my-mult (- x 1) y-promise))]))

(my-mult e1 (my-delay (lambda () e2)))
```

Some languages, most notably Haskell, use this approach for all function calls, i.e., the semantics for function calls is different in these languages: If an argument is never used it is never evaluated, else it is evaluated only once. This is called *call-by-need* whereas all the languages we will use are *call-by-value* (arguments are fully evaluated before the call is made).

Streams

A stream is an infinite sequence of values. We obviously cannot create such a sequence explicitly (it would literally take forever), but we can create code that knows how to produce the infinite sequence and other code that knows how to ask for however much of the sequence it needs.

Streams are very common in computer science. You can view the sequence of bits produced by a synchronous circuit as a stream, one value for each clock cycle. The circuit does not know how long it should run, but it can produce new values forever. The UNIX pipe (`cmd1 | cmd2`) is a stream; it causes `cmd1` to produce only as much output as `cmd2` needs for input. Web programs that react to things users click on web pages can treat the user's activities as a stream — not knowing when the next will arrive or how many there are, but ready to respond appropriately. More generally, streams can be a convenient division of labor: one part of the software knows how to produce successive values in the infinite sequence but does not know how many will be needed and/or what to do with them. Another part can determine how many are needed but does not know how to generate them.

There are many ways to code up streams; we will take the simple approach of representing a stream as a thunk that when called produces a pair of (1) the first element in the sequence and (2) a thunk that represents the stream for the second-through-infinity elements. Defining such thunks typically uses recursion. Here are three examples:

```
(define ones (lambda () (cons 1 ones)))
```

```
(define nats
  (letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
(define powers-of-two
  (letrec ([f (lambda (x) (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Given this encoding of streams and a stream `s`, we would get the first element via `(car (s))`, the second element via `(car ((cdr (s))))`, the third element via `(car ((cdr ((cdr (s))))))`, etc. Remember parentheses matter: `(e)` calls the thunk `e`.

We could write a higher-order function that takes a stream and a predicate-function and returns how many stream elements are produced before the predicate-function returns true:

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                  (if (tester (car pr))
                      ans
                      (f (cdr pr) (+ ans 1)))))])
    (f stream 1)))
```

As an example, `(number-until powers-of-two (lambda (x) (= x 16)))` evaluates to 4.

As a side-note, all the streams above can produce their next element given at most their previous element. So we could use a higher-order function to abstract out the common aspects of these functions, which lets us put the stream-creation logic in one place and the details for the particular streams in another. This is just another example of using higher-order functions to reuse common functionality:

```
(define (stream-maker fn arg)
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (fn x arg)))))])
    (lambda () (f arg))))
(define ones (stream-maker (lambda (x y) 1) 1))
(define nats  (stream-maker + 1))
(define powers-of-two (stream-maker * 2))
```

Memoization

An idiom related to lazy evaluation that does not actually use thunks is *memoization*. If a function does not have side-effects, then if we call it multiple times with the same argument(s), we do not actually have to do the call more than once. Instead, we can look up what the answer was the first time we called the function with the argument(s).

Whether this is a good idea or not depends on trade-offs. Keeping old answers in a table takes space and table lookups do take some time, but compared to reperforming expensive computations, it can be a big win. Again, for this technique to even be *correct* requires that given the same arguments a function will always return the same result and have no side-effects. So being able to use this *memo table* (i.e., do memoization) is yet another advantage of avoiding mutation.

To implement memoization we do use mutation: Whenever the function is called with an argument we have not seen before, we compute the answer and then add the result to the table (via mutation).

As an example, let's consider 3 versions of a function that takes an `x` and returns fibonacci(x). (A Fibonacci number is a well-known definition that is useful in modeling populations and such.) A simple recursive definition is:

```
(define (fibonacci x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci (- x 1))
         (fibonacci (- x 2)))))
```

Unfortunately, this function takes exponential time to run. We might start noticing a pause for `(fibonaccci 30)`, and `(fibonacci 40)` takes a thousand times longer than that, and `(fibonacci 10000)` would take more seconds than there are particles in the universe. Now, we could fix this by taking a "count up" approach that remembers previous answers:

```
(define (fibonacci x)
  (letrec ([f (lambda (acc1 acc2 y)
                (if (= y x)
                    (+ acc1 acc2)
                    (f (+ acc1 acc2) acc1 (+ y 1))))])
    (if (or (= x 1) (= x 2))
        1
        (f 1 1 3))))
```

This takes linear time, so `(fibonacci 10000)` returns almost immediately (and with a very large number), but it required a quite different approach to the problem. With memoization we can turn `fibonacci` into an efficient algorithm with a technique that works for lots of algorithms. It is closely related to "dynamic programming," which you often learn about in advanced algorithms courses. Here is the version that does this memoization (the `assoc` library function is described below):

```
(define fibonacci
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans)
                     (let ([new-ans (if (or (= x 1) (= x 2))
                                        1
                                        (+ (f (- x 1))
                                           (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

It is essential that different calls to `f` use the *same* mutable memo-table: if we create the table inside the call to `f`, then each call will use a new empty table, which is pointless. But we do not put the table at top-level just because that would be bad style since its existence should be known only to the implementation of `fibonacci`.

Why does this technique work to make `(fibonacci 10000)` complete quickly? Because when we evaluate `(f (- x 2))` on any recursive calls, the result is already in the table, so there is no longer an exponential

number of recursive calls. This is much more important than the fact that calling `(fibonacci 10000)` a second time will complete even more quickly since the answer will be in the memo-table.

For a large table, using a list and Racket's `assoc` function may be a poor choice, but it is fine for demonstrating the concept of memoization. `assoc` is just a library function in Racket that takes a value and a list of pairs and returns the first pair in the list where the car of the pair equal to the value. It returns `#f` if no pair has such a car. (The reason `assoc` returns the pair rather than the cdr of the pair is so you can distinguish the case where no pair matches from the case where the pair that matches has `#f` in its cdr. This is the sort of thing we would use an option for in ML.)

Macros: The Key Points

The last topic in this module is *macros*, which add to the *syntax* of a language by letting programmers define their own syntactic sugar. To preserve time for other topics, most of the macro material will be optional, but you are encouraged to work through it nonetheless. This section contains the key ideas that are *not* optional: Though you do not need macros for the homework assignment associated with this module, we need the *idea* for part of our study in the next module.

A *macro definition* introduces some new syntax into the language. It describes how to transform the new syntax into different syntax in the language itself. A *macro system* is a language (or part of a larger languages) for defining macros. A *macro use* is just using one of the macros previously defined. The semantics of a macro use is to replace the macro use with the appropriate syntax as defined by the macro definition. This process is often called *macro expansion* because it is common but not required that the syntactic transformation produces a larger amount of code.

The key point is that macro expansion happens *before* anything else we have learned about: before type-checking, before compiling, before evaluation. Think of "expanding all the macros" as a pre-pass over your entire program before anything else occurs. So macros get expanded everywhere, such as in function bodies, both branches of conditionals, etc.

Here are 3 examples of macros one might define in Racket:

- A macro so that programmers can write `(my-if e1 then e2 else e3)` where `my-if`, `then`, and `else` are keywords and this macro-expands to `(if e1 e2 e3)`.
- A macro so that programmers can write `(comment-out e1 e2)` and have it transform to `e2`, i.e., it is a convenient way to take an expression `e1` out of the program (replacing it with `e2`) without actually deleting anything.
- A macro so that programmers can write `(my-delay e)` and have it transform to `(mcons #f (lambda () e))`. This is different from the `my-delay` *function* we defined earlier because the function required the caller to pass in a thunk. Here the macro expansion does the thunk creation and the macro user should *not* include an explicit thunk.

Racket has an excellent and sophisticated macro system. For precise, technical reasons, its macro system is superior to many of the better known macro systems, notably the preprocessor in C or C++. So we can use Racket to learn some of the pitfalls of macros in general. The rest of this module (optional) will discuss:

- How macro systems must handle issues of tokenization, parenthesization, and scope — and how Racket handles parenthesization and scope better than C/C++
- How to define macros in Racket, such as the ones described above

- How macro definitions should be careful about the order expressions are evaluated and how many times they are evaluated
- The key issue of variable bindings in macros and the notion of *hygiene*

Optional: Tokenization, Parenthesization, and Scope

The definition of macros and macro expansion is more structured and subtle than "find-and-replace" like one might do in a text editor or with a script you write manually to perform some string substitution in your program. Macro expansion is different in roughly three ways.

First, consider a macro that, "replaces every use of `head` with `car`." In macro systems, that does *not* mean some variable `headt` would be rewritten as `cart`. So the implementation of macros has to at least understand how a programming language's text is broken into *tokens* (i.e., words). This notion of tokens is different in different languages. For example, `a-b` would be three tokens in most languages (a variable, a subtraction, and another variable), but is one token in Racket.

Second, we can ask if macros do or do not understand parenthesization. For example, in C/C++, if you have a macro

```
#define ADD(x,y) x+y
```

then `ADD(1,2/3)*4` gets rewritten as `1 + 2 / 3 * 4`, which is *not* the same thing as `(1 + 2/3)*4`. So in such languages, macro writers generally include lots of explicit parentheses in their macro definitions, e.g.,

```
#define ADD(x,y) ((x)+(y))
```

In Racket, macro expansion preserves the code structure so this issue is not a problem. A Racket macro use always looks like `(x ...)` where `x` is the name of a macro and the result of the expansion "stays in the same parentheses" (e.g., `(my-if x then y else z)` might expand to `(if x y z)`). This is an advantage of Racket's minimal and consistent syntax.

Third, we can ask if macro expansion happens even when creating variable bindings. If not, then local variables can shadow macros, which is probably what you want. For example, suppose we have:

```
(let  ([hd 0] [car 1]) hd) ; evaluates to 0
(let* ([hd 0] [car 1]) hd) ; evaluates to 0
```

If we replace `car` with `hd`, then the first expression is an error (trying to bind `hd` twice) and the second expression now evaluates to 1. In Racket, macro expansion does not apply to variable definitions, i.e., the `car` above is different and shadows any macro for `car` that happens to be in scope.

Optional: Defining Macros with `define-syntax`

Let's now walk through the syntax we will use to define macros in Racket. (There have been many variations in Racket's predecessor Scheme over the years; this is one modern approach we will use.) Here is a macro that lets users write `(my-if e1 then e2 else e3)` for any expressions `e1`, `e2`, and `e3` and have it mean exactly `(if e1 e2 e3)`:

```
(define-syntax my-if
  (syntax-rules (then else)
    [(my-if e1 then e2 else e3)
     (if e1 e2 e3)]))
```

- `define-syntax` is the special form for defining a macro.
- `my-if` is the name of our macro. It adds `my-if` to the environment so that expressions of the form `(my-if ...)` will be macro-expanded according to the syntax rules in the rest of the macro definition.
- `syntax-rules` is a keyword.
- The next parenthesized list (in this case `(then else)`) is a list of "keywords" for this macro, i.e., any use of `then` or `else` in the body of `my-if` is just syntax whereas anything not in this list (and not `my-if` itself) represents an arbitrary expression.
- The rest is a list of pairs: how `my-if` might be used and how it should be rewritten if it is used that way.
- In this example, our list has only one option: `my-if` must be used in an expression of the form `(my-if e1 then e2 else e3)` and that becomes `(if e1 e2 e3)`. Otherwise an error results. Note the rewriting occurs *before* any evaluation of the expressions `e1`, `e2`, or `e3`, unlike with functions. This is what we want for a conditional expression like `my-if`.

Here is a second simple example where we use a macro to "comment out" an expression. We use `(comment-out e1 e2)` to be rewritten as `e2`, meaning `e1` will never be evaluated. This might be more convenient when debugging code than actually using comments.

```
(define-syntax comment-out
  (syntax-rules ()
    [(comment-out e1 e2) e2]))
```

Our third example is a macro `my-delay` so that, unlike the `my-delay` function defined earlier, users would write `(my-delay e)` to create a promise such that `my-force` would evaluate `e` and remember the result, rather than users writing `(my-delay (lambda () e))`. Only a macro, not a function, can "delay evaluation by adding a thunk" like this because function calls always evaluate their arguments.

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda () e))]))
```

We should *not* create a macro version of `my-force` because our function version from earlier is just what we want. Give `(my-force e)` we *do* want to evaluate `e` to a value, which should be an mcons-cell created by `my-delay` and then perform the computation in the `my-force` function. Defining a macro provides no benefit and can be error prone. Consider this awful attempt:

```
(define-syntax my-force
  (syntax-rules ()
     [(my-force e)
      (if (mcar e)
```

```
          (mcdr e)
          (begin (set-mcar! e #t)
                 (set-mcdr! e ((mcdr e)))
                 (mcdr e)))]))
```

Due to macro expansion, uses of this macro will end up evaluating their argument *multiple times*, which can have strange behavior if `e` has side effects. Macro users will not expect this. In code like:

```
(let ([t (my-delay some-complicated-expression)])
  (my-force t))
```

this does not matter since `t` is already bound to a value, but in code like:

```
(my-force (begin (print "hi") (my-delay some-complicated-expression)))
```

we end up printing multiple times. Remember that macro expansion copies the entire argument `e` everywhere it appears in the macro definition, but we often want it to be evaluated only once. This version of the macro does the right thing in this regard:

```
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let ([x e])
       (if (mcar x)
           (mcdr x)
           (begin (set-mcar! x #t)
                  (set-mcdr! x ((mcdr x)))
                  (mcdr x))))]))
```

But, again, there is *no reason* to define a macro like this since a function does exactly what we need. Just stick with:

```
(define (my-force th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

Optional: Variables, Macros, and Hygiene

Let's consider a macro that doubles its argument. Note this is poor style because if you want to double an argument you should just write a function: `(define (double x) (* 2 x))` or `(define (double x) (+ x x))` which are equivalent to each other. But this short example will let us investigate when macro arguments are evaluated and in what environment, so we will use it just as a poor-style example.

These two macros are *not* equivalent:

```
(define-syntax double1
```

```
  (syntax-rules ()
         [(double1 e)
          (* 2 e)]))
(define-syntax double2
  (syntax-rules ()
    [(double2 e)
     (+ e e)]))
```

The reason is `double2` will evaluate its argument twice. So `(double1 (begin (print "hi") 17))` prints `"hi"` once but `(double2 (begin (print "hi") 17))` prints `"hi"` twice. The function versions print `"hi"` once, simply because, as always, function arguments are evaluated to values before the function is called.

To fix `double2` without "changing the algorithm" to multiplication instead of addition, we should use a local variable:

```
(define-syntax double3
  (syntax-rules ()
    [(double3 e)
     (let ([x e])
       (+ x x))]))
```

Using local variables in macro definitions to control if/when expressions get evaluated is exactly what you should do, but in less powerful macro languages (again, C/C++ is an easy target for derision here), local variables in macros are typically avoided. The reason has to do with scope and something that is called ***hygiene***. For sake of example, consider this silly variant of `double3`:

```
(define-syntax double4
  (syntax-rules ()
    [(double4 e)
     (let* ([zero 0]
            [x e])
       (+ x x zero))]))
```

In Racket, this macro always works as expected, but that may/should surprise you. After all, suppose I have this use of it:

```
(let ([zero 17])
   (double4 zero))
```

If you do the syntactic rewriting as expected, you will end up with

```
(let ([zero 17])
  (let* ([zero 0]
         [x zero])
       (+ x x zero)))
```

But this expression evaluates to 0, not to 34. The problem is a *free variable* at the macro-use (the `zero` in `(double4 zero)`) ended up in the scope of a local variable in the macro definition. That is why in C/C++, local variables in macro definitions tend to have funny names like `__x_hopefully_no_conflict` in the hope that this sort of thing will not occur. In Racket, the rule for macro expansion is more sophisticated to avoid this problem. Basically, every time a macro is used, all of its local variables are *rewritten* to be fresh

new variable names that do not conflict with anything else in the program. This is "one half" of what by definition make Racket macros hygienic.

The other half has to do with free variables in the *macro definition* and making sure they do not wrongly end up in the scope of some local variable where the macro is used. For example, consider this strange code that uses `double3`:

```
(let ([+ *])
   (double3 17))
```

The naive rewriting would produce:

```
(let ([+ *])
  (let ([x 17])
     (+ 17 17)))
```

Yet this produces 17^2, not 34. Again, the naive rewriting is *not* what Racket does. Free variables in a macro definition always refer to what was in the environment where the macro was defined, not where the macro was used. This makes it much easier to write macros that always work as expected. Again macros in C/C++ work like the naive rewriting.

There are situations where you do not want hygiene. For example, suppose you wanted a macro for for-loops where the macro user specified a variable that would hold the loop-index and the macro definer made sure that variable held the correct value on each loop iteration. Racket's macro system has a way to do this, which involves explicitly violating hygiene, but we will not demonstrate it here.

Optional: More macro examples

Finally, let's consider a few more useful macro definitions, including ones that use multiple cases for how to do the rewriting. First, here is a macro that lets you write up to two let-bindings using `let*` semantics but with fewer parentheses:

```
(define-syntax let2
  (syntax-rules ()
    [(let2 () body)
     body]
    [(let2 (var val) body)
     (let ([var val]) body)]
    [(let2 (var1 val1 var2 val2) body)
     (let ([var1 val1])
       (let ([var2 val2])
         body))]))
```

As examples, `(let2 () 4)` evaluates to 4, `(let2 (x 5) (+ x 4)` evaluates to 9, and `(let2 (x 5 y 6) (+ x y))` evaluates to 11.

In fact, given support for recursive macros, we could redefine Racket's `let*` entirely in terms of `let`. We need some way to talk about "the rest of a list of syntax" and Racket's `...` gives us this:

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body)
     body]
    [(my-let* ([var0 val0]
               [var-rest val-rest] ...)
              body)
     (let ([var0 val0])
       (my-let* ([var-rest val-rest] ...)
                body))]))
```

Since macros are recursive, there is nothing to prevent you from generating an infinite loop or an infinite amount of syntax during macro expansion, i.e., before the code runs. The example above does not do this because it recurs on a shorter list of bindings.

Finally, here is a macro for a limited form of for-loop that executes its body $hi - lo$ times. (It is limited because the body is not given the current iteration number.) Notice the use of a let expression to ensure we evaluate `lo` and `hi` exactly once but we evaluate `body` the correct number of times.

```
(define-syntax for
  (syntax-rules (to do)
    [(for lo to hi do body)
     (let ([l lo]
           [h hi])
       (letrec ([loop (lambda (it)
                        (if (> it h)
                            #t
                            (begin body (loop (+ it 1)))))])
         (loop l)))]))
```

Programming Languages (Coursera / University of Washington) Assignment 4

You will write 10 Racket functions (not counting helper functions). Writing a Racket macro is a challenge problem.

Download several files from the course website and put them all in the same directory. Add to `hw4.rkt` to complete your homework.

Provided Code for Graphical Output:

The code at the top of `hw4combined_tests_with_graphics.rkt` uses a graphics library to provide a simple, entertaining (?) outlet for your streams. You need not understand this code (though it is not complicated) or even use it, but it may make the homework more fun. This is how you use it:

- `(open-window)` returns a graphics window you can pass as the first argument to `place-repeatedly`.
- `(place-repeatedly window pause stream n)` uses the first `n` values produced by `stream`. Each stream element must be a pair where the first value is an integer between 0 and 5 inclusive and the second value is a string that is the name of an image file (e.g., `.jpg`). (Sample image files that will work well are available on the course website. Put them in the same directory as your code.) Every `pause` seconds (where `pause` is a decimal, i.e., floating-point, number), the next stream value is retrieved, the corresponding image file is opened, and it is placed in the window using the number in the pair to choose its position in a 2x3 grid as follows:

0	1	2
3	4	5

Two of the provided tests demonstrate how to use `place-repeatedly`. The provided tests require you to complete several of the problems, of course. We hope these tests' expected (visual) behavior is not difficult for you to figure out.

Small Example Tests:

The example tests in `hw4test.rkt` are grouped into a small test suite using Racket's unit-testing framework. You do not need to understand the details, but it is worthwhile to do so by reading `http://docs.racket-lang.org/rackunit/quick-start.html`.

Helpful Guide / Warning:

The first three problems are "warm-up" exercises for Racket. Subsequent problems dive into streams (4–8) and memoization (10). Some short problems may be difficult. Go slowly and focus on using what you learned about thunks, streams, etc.

Some problems require that you use a few standard-library functions that were not used in lecture. See the Racket documentation at `http://docs.racket-lang.org/`, particularly The Racket Guide, as necessary — looking up library functions even in languages new to you is an important skill. It is fine to discuss with others in the class what library functions are useful and how they work.

Turn-in Instructions (same as in Part A of the course):

First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

Problems:

1. Write a function `sequence` that takes 3 arguments `low`, `high`, and `stride`, all assumed to be numbers. Further assume `stride` is positive. `sequence` produces a list of numbers from `low` to `high` (including `low` and possibly `high`) separated by `stride` and in sorted order. Sample solution: 4 lines. Examples:

Call	Result
`(sequence 3 11 2)`	`'(3 5 7 9 11)`
`(sequence 3 8 3)`	`'(3 6)`
`(sequence 3 2 1)`	`'()`

2. Write a function `string-append-map` that takes a list of strings `xs` and a string `suffix` and returns a list of strings. Each element of the output should be the corresponding element of the input appended with `suffix` (with no extra space between the element and `suffix`). You must use Racket-library functions `map` and `string-append`. Sample solution: 2 lines.

3. Write a function `list-nth-mod` that takes a list `xs` and a number `n`. If the number is negative, terminate the computation with `(error "list-nth-mod: negative number")`. Else if the list is empty, terminate the computation with `(error "list-nth-mod: empty list")`. Else return the i^{th} element of the list where we *count from zero* and i is the remainder produced when dividing `n` by the list's length. Library functions `length`, `remainder`, `car`, and `list-tail` are all useful – see the Racket documentation. Sample solution is 6 lines.

4. Write a function `stream-for-n-steps` that takes a stream `s` and a number `n`. It returns a list holding the first `n` values produced by `s` in order. Assume `n` is non-negative. Sample solution: 5 lines. Note: You can test your streams with this function instead of the graphics code.

5. Write a stream `funny-number-stream` that is like the stream of natural numbers (i.e., 1, 2, 3, ...) except numbers divisble by 5 are negated (i.e., 1, 2, 3, 4, -5, 6, 7, 8, 9, -10, 11, ...). Remember a stream is a thunk that when called produces a pair. Here the car of the pair will be a number and the cdr will be another stream.

6. Write a stream `dan-then-dog`, where the elements of the stream alternate between the strings `"dan.jpg"` and `"dog.jpg"` (starting with `"dan.jpg"`). More specifically, `dan-then-dog` should be a thunk that when called produces a pair of `"dan.jpg"` and a thunk that when called produces a pair of `"dog.jpg"` and a thunk that when called... etc. Sample solution: 4 lines.

7. Write a function `stream-add-zero` that takes a stream `s` and returns another stream. If `s` would produce v for its i^{th} element, then `(stream-add-zero s)` would produce the pair `(0 .` v`)` for its i^{th} element. Sample solution: 4 lines. Hint: Use a thunk that when called uses `s` and recursion. Note: One of the provided tests in the file using graphics uses `(stream-add-zero dan-then-dog)` with `place-repeatedly`.

8. Write a function `cycle-lists` that takes two lists `xs` and `ys` and returns a stream. The lists may or may not be the same length, but assume they are both non-empty. The elements produced by the stream are pairs where the first part is from `xs` and the second part is from `ys`. The stream cycles forever through the lists. For example, if `xs` is `'(1 2 3)` and `ys` is `'("a" "b")`, then the stream would produce, `(1 . "a")`, `(2 . "b")`, `(3 . "a")`, `(1 . "b")`, `(2 . "a")`, `(3 . "b")`, `(1 . "a")`, `(2 . "b")`, etc.

   Sample solution is 6 lines and is more complicated than the previous stream problems. Hints: Use one of the functions you wrote earlier. Use a recursive helper function that takes a number `n` and calls itself with `(+ n 1)` inside a thunk.

9. Write a function `vector-assoc` that takes a value `v` and a vector `vec`. It should behave like Racket's `assoc` library function except (1) it processes a vector (Racket's name for an array) instead of a list, (2) it allows vector elements not to be pairs in which case it skips them, and (3) it always takes exactly two arguments. Process the vector elements in order starting from 0. You must use library functions `vector-length`, `vector-ref`, and `equal?`. Return `#f` if no vector element is a pair with a `car` field equal to `v`, else return the first pair with an equal `car` field. Sample solution is 9 lines, using one local recursive helper function.

10. Write a function `cached-assoc` that takes a list `xs` and a number `n` and returns a function that takes one argument `v` and returns the same thing that `(assoc v xs)` would return. However, you should use an n-element *cache of recent results* to possibly make this function faster than just calling `assoc` (if `xs` is long and a few elements are returned often). The cache must be a Racket vector of length n that is created by the call to `cached-assoc` (use Racket library function `vector` or `make-vector`) and used-and-possibly-mutated each time the function returned by `cached-assoc` is called. Assume n is positive.

    The cache starts empty (all elements `#f`). When the function returned by `cached-assoc` is called, it first checks the cache for the answer. If it is not there, it uses `assoc` and `xs` to get the answer and if the result is not `#f` (i.e., `xs` has a pair that matches), it adds the pair to the cache before returning (using `vector-set!`). The cache slots are used in a round-robin fashion: the first time a pair is added to the cache it is put in position 0, the next pair is put in position 1, etc. up to position $n-1$ and then back to position 0 (replacing the pair already there), then position 1, etc.

    Hints:

    - In addition to a variable for holding the vector whose contents you mutate with `vector-set!`, use a second variable to keep track of which cache slot will be replaced next. After modifying the cache, increment this variable (with `set!`) or set it back to 0.
    - To test your cache, it can be useful to add print expressions so you know when you are using the cache and when you are not. But remove these print expressions before submitting your code.
    - Sample solution is 15 lines.

11. **(Challenge Problem:)** Define a macro that is used like `(while-less e1 do e2)` where `e1` and `e2` are expressions and `while-less` and `do` are syntax (keywords). The macro should do the following:

    - It evaluates `e1` exactly once.
    - It evaluates `e2` at least once.
    - It keeps evaluating `e2` until and only until the result is not a number less than the result of the evaluation of `e1`.
    - Assuming evaluation terminates, the result is `#t`.
    - Assume `e1` and `e2` produce numbers; your macro can do anything or fail mysteriously otherwise.

    Hint: Define and use a recursive thunk. Sample solution is 9 lines. Example:

    ```
    (define a 2)
    (while-less 7 do (begin (set! a (+ a 1)) (print "x") a))
    (while-less 7 do (begin (set! a (+ a 1)) (print "x") a))
    ```

    Evaluating the second line will print `"x"` 5 times and change `a` to be 7. So evaluating the third line will print `"x"` 1 time and change `a` to be 8.

Assessment: We will automatically test your functions on a variety of inputs, including edge cases. We may automatically check that you used required library functions. We will also ask peers to evaluate your code for simplicity, conciseness, elegance, and good formatting including indentation and line breaks. Do not use mutation except in problem 10. (If doing the challenge problem, you will also use mutation to *test* problem 11.)

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)))))
```

Programming Languages (Coursera / University of Washington) Assignment 5

Set-up: For this assignment, edit a copy of `hw5.rkt`, which is on the course website. In particular, replace occurrences of `"CHANGE"` to complete the problems. Do not use any mutation (`set!`, `set-mcar!`, etc.) anywhere in the assignment.

Overview: This homework has to do with MUPL (a Made Up Programming Language). MUPL programs are written directly in Racket by using the constructors defined by the structs defined at the beginning of `hw5.rkt`. This is the definition of MUPL's syntax:

- If s is a Racket string, then `(var` s`)` is a MUPL expression (a variable use).
- If n is a Racket integer, then `(int` n`)` is a MUPL expression (a constant).
- If e_1 and e_2 are MUPL expressions, then `(add` e_1 e_2`)` is a MUPL expression (an addition).
- If s_1 and s_2 are Racket strings and e is a MUPL expression, then `(fun` s_1 s_2 e`)` is a MUPL expression (a function). In e, s_1 is bound to the function itself (for recursion) and s_2 is bound to the (one) argument. Also, `(fun #f` s_2 e`)` is allowed for anonymous nonrecursive functions.
- If e_1, e_2, and e_3, and e_4 are MUPL expressions, then `(ifgreater` e_1 e_2 e_3 e_4`)` is a MUPL expression. It is a conditional where the result is e_3 if e_1 is strictly greater than e_2 else the result is e_4. Only one of e_3 and e_4 is evaluated.
- If e_1 and e_2 are MUPL expressions, then `(call` e_1 e_2`)` is a MUPL expression (a function call).
- If s is a Racket string and e_1 and e_2 are MUPL expressions, then `(mlet` s e_1 e_2`)` is a MUPL expression (a let expression where the value resulting e_1 is bound to s in the evaluation of e_2).
- If e_1 and e_2 are MUPL expressions, then `(apair` e_1 e_2`)` is a MUPL expression (a pair-creator).
- If e_1 is a MUPL expression, then `(fst` e_1`)` is a MUPL expression (getting the first part of a pair).
- If e_1 is a MUPL expression, then `(snd` e_1`)` is a MUPL expression (getting the second part of a pair).
- `(aunit)` is a MUPL expression (holding no data, much like `()` in ML or `null` in Racket). Notice `(aunit)` is a MUPL expression, but `aunit` is not.
- If e_1 is a MUPL expression, then `(isaunit` e_1`)` is a MUPL expression (testing for `(aunit)`).
- `(closure` *env* f`)` is a MUPL value where f is MUPL function (an expression made from `fun`) and *env* is an environment mapping variables to values. Closures do not appear in source programs; they result from evaluating functions.

A MUPL *value* is a MUPL integer constant, a MUPL closure, a MUPL aunit, or a MUPL pair of MUPL values. Similar to Racket, we can build list values out of nested pair values that end with a MUPL aunit. Such a MUPL value is called a MUPL list.

You should assume MUPL programs are syntactically correct (e.g., do not worry about wrong things like `(int "hi")` or `(int (int 37))`. But do *not* assume MUPL programs are free of type errors like `(add (aunit) (int 7))` or `(fst (int 7))`.

Warning: What makes this assignment challenging is that you have to understand MUPL well and debugging an interpreter is an acquired skill.

Turn-in Instructions (same as for previous assignments): First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment

requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

Problems:

1. **Warm-Up:**
   (a) Write a Racket function `racketlist->mupllist` that takes a Racket list (presumably of MUPL values but that will not affect your solution) and produces an analogous MUPL list with the same elements in the same order.
   (b) Write a Racket function `mupllist->racketlist` that takes a MUPL list (presumably of MUPL values but that will not affect your solution) and produces an analogous Racket list (of MUPL values) with the same elements in the same order.

2. **Implementing the MUPL Language:** Write a MUPL interpreter, i.e., a Racket function `eval-exp` that takes a MUPL expression `e` and either returns the MUPL value that `e` evaluates to under the empty environment or calls Racket's `error` if evaluation encounters a run-time MUPL type error or unbound MUPL variable.

   A MUPL expression is evaluated under an environment (for evaluating variables, as usual). In your interpreter, use a Racket list of Racket pairs to represent this environment (which is initially empty) so that you can use ***without modification*** the provided `envlookup` function. Here is a description of the semantics of MUPL expressions:

   - All values (including closures) evaluate to themselves. For example, `(eval-exp (int 17))` would return `(int 17)`, *not* 17.
   - A variable evaluates to the value associated with it in the environment.
   - An addition evaluates its subexpressions and assuming they both produce integers, produces the integer that is their sum. (Note this case is done for you to get you pointed in the right direction.)
   - Functions are lexically scoped: A function evaluates to a closure holding the function and the current environment.
   - An `ifgreater` evaluates its first two subexpressions to values v_1 and v_2 respectively. If both values are integers, it evaluates its third subexpression if v_1 is a strictly greater integer than v_2 else it evaluates its fourth subexpression.
   - An mlet expression evaluates its first expression to a value v. Then it evaluates the second expression to a value, in an environment extended to map the name in the mlet expression to v.
   - A call evaluates its first and second subexpressions to values. If the first is not a closure, it is an error. Else, it evaluates the closure's function's body in the closure's environment extended to map the function's name to the closure (unless the name field is `#f`) and the function's argument-name (i.e., the parameter name) to the result of the second subexpression.
   - A pair expression evaluates its two subexpressions and produces a (new) pair holding the results.
   - A fst expression evaluates its subexpression. If the result for the subexpression is a pair, then the result for the fst expression is the `e1` field in the pair.
   - A snd expression evaluates its subexpression. If the result for the subexpression is a pair, then the result for the snd expression is the `e2` field in the pair.
   - An `isaunit` expression evaluates its subexpression. If the result is an aunit expression, then the result for the isaunit expression is the MUPL value `(int 1)`, else the result is the MUPL value `(int 0)`.

   Hint: The `call` case is the most complicated. In the sample solution, no case is more than 12 lines and several are 1 line.

3. **Expanding the Language:** MUPL is a small language, but we can write Racket functions that act like MUPL macros so that users of these functions feel like MUPL is larger. The Racket functions produce MUPL expressions that could then be put inside larger MUPL expressions or passed to `eval-exp`. In implementing these Racket functions, do not use `closure` (which is used only internally in `eval-exp`). Also do not use `eval-exp` (we are creating a program, not running it).

   (a) Write a Racket function `ifaunit` that takes three MUPL expressions e_1, e_2, and e_3. It returns a MUPL expression that when run evaluates e_1 and if the result is MUPL's aunit then it evaluates e_2 and that is the overall result, else it evaluates e_3 and that is the overall result. Sample solution: 1 line.

   (b) Write a Racket function `mlet*` that takes a Racket list of Racket pairs '$((s_1 \ . \ e_1) \ldots (s_i \ . \ e_i) \ldots (s_n \ . \ e_n))$ and a final MUPL expression e_{n+1}. In each pair, assume s_i is a Racket string and e_i is a MUPL expression. `mlet*` returns a MUPL expression whose value is e_{n+1} evaluated in an environment where each s_i is a variable bound to the result of evaluating the corresponding e_i for $1 \leq i \leq n$. The bindings are done sequentially, so that each e_i is evaluated in an environment where s_1 through s_{i-1} have been previously bound to the values e_1 through e_{i-1}.

   (c) Write a Racket function `ifeq` that takes four MUPL expressions e_1, e_2, e_3, and e_4 and returns a MUPL expression that acts like `ifgreater` except e_3 is evaluated if and only if e_1 and e_2 are equal integers. Assume none of the arguments to `ifeq` use the MUPL variables `_x` or `_y`. Use this assumption so that when an expression returned from `ifeq` is evaluated, e_1 and e_2 are evaluated exactly once each.

4. **Using the Language:** We can write MUPL expressions directly in Racket using the constructors for the structs and (for convenience) the functions we wrote in the previous problem.

   (a) Bind to the Racket variable `mupl-map` a MUPL function that acts like map (as we used extensively in ML). Your function should be curried: it should take a MUPL function and return a MUPL function that takes a MUPL list and applies the function to every element of the list returning a new MUPL list. Recall a MUPL list is aunit or a pair where the second component is a MUPL list.

   (b) Bind to the Racket variable `mupl-mapAddN` a MUPL function that takes an MUPL integer i and returns a MUPL function that takes a MUPL list of MUPL integers and returns a new MUPL list of MUPL integers that adds i to every element of the list. Use `mupl-map` (a use of `mlet` is given to you to make this easier).

5. **Challenge Problem:** Write a second version of `eval-exp` (bound to `eval-exp-c`) that builds closures with smaller environments: When building a closure, it uses an environment that is like the current environment but holds only variables that are free variables in the function part of the closure. (A free variable is a variable that appears in the function without being under some shadowing binding for the same variable.)

   Avoid computing a function's free variables more than once. Do this by writing a function `compute-free-vars` that takes an expression and returns a different expression that uses `fun-challenge` everywhere in place of `fun`. The new struct `fun-challenge` (provided to you; do not change it) has a field `freevars` to store exactly the set of free variables for the function. Store this set as a Racket set of Racket strings. (Sets are predefined in Racket's standard library; consult the documentation for useful functions such as `set`, `set-add`, `set-member?`, `set-remove`, `set-union`, and any other functions you wish.)

   You must have a top-level function `compute-free-vars` that works as just described — storing the free variables of each function in the `freevars` field — so the grader can test it directly. Then write a new "main part" of the interpreter that expects the sort of MUPL expression that `compute-free-vars` returns. The case for function definitions is the interesting one.

Coursera Programming Languages Course
Section 7 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

ML versus Racket

Before studying the general topic of static typing and the advantages/disadvantages thereof, it is interesting to do a more specific comparison between the two languages we have studied so far, ML and Racket. The languages are similar in many ways, with constructs that encourage a functional style (avoiding mutation, using first-class closures) while allowing mutation where appropriate. There are also many differences, including very different approaches to syntax, ML's support for pattern-matching compared to Racket's accessor functions for structs, Racket's multiple variants of let-expressions, etc.

But the most widespread difference between the two languages is that ML has a static type system that Racket does not.[1]

We study below precisely what a static type system is, what ML's type system guarantees, and what the advantages and disadvantages of static typing are. Anyone who has programmed in ML and Racket probably already has some ideas on these topics, naturally: ML rejects lots of programs before running them by doing type-checking and reporting errors. To do so, ML enforces certain restrictions (e.g., all elements of a list must have the same type). As a result, ML ensures the absence of certain errors (e.g., we will never try to pass a string to the addition operator) "at compile time."

[1]There is a related language Typed Racket also available within the DrRacket system that interacts well with Racket and many other languages — allowing you to mix files written in different languages to build applications. We will not study that in this course, so we refer here only to the language Racket.

More interestingly, could we describe ML and its type system in terms of ideas more Racket-like and, conversely, could we describe Racket-style programming in terms of ML? It turns out we can and that doing so is both mind-expanding and a good precursor to subsequent topics.

First consider how a Racket programmer might view ML. Ignoring syntax differences and other issues, we can describe ML as roughly defining a *subset* of Racket: Programs that run produce similar answers, but ML rejects many more programs as illegal, i.e., not part of the language. What is the advantage of that? ML is designed to reject programs that are likely bugs. Racket allows programs like `(define (f y) (+ y (car y)))`, but any call to `f` would cause an error, so this is hardly a useful program. So it is helpful that ML rejects this program rather than waiting until a programmer tests `f`. Similarly, the type system catches bugs due to inconsistent assumptions by different parts of the program. The functions `(define (g x) (+ x x))` and `(define (h z) (g (cons z 2)))` are both sensible by themselves, but if the `g` in `h` is bound to this definition of `g`, then any call to `h` fails much like any call to `f`. On the other hand, ML rejects Racket-like programs that are not bugs as well. For example, in this code, both the if-expression and the expression bound to `xs` would not type-check but represent reasonable Racket idioms depending on circumstances:

```
(define (f x) (if (> x 0) #t (list 1 2)))
(define xs (list 1 #t "hi"))
(define y (f (car xs)))
```

So now how might an ML programmer view Racket? One view is just the reverse of the discussion above, that Racket accepts a superset of programs, some of which are errors and some of which are not. A more interesting view is that Racket is just ML where *every expression is part of one big datatype*. In this view, the result of every computation is *implicitly* "wrapped" by a constructor into the one big datatype and primitives like `+` have implementations that check the "tags" of their arguments (e.g., to see if they are numbers) and raise errors as appropriate. In more detail, it is like Racket has this one datatype binding:

```
datatype theType = Int of int
                 | String of string
                 | Pair of theType * theType
                 | Fun of theType -> theType
                 | ... (* one constructor per built-in type *)
```

Then it is like when programmers write something like 42, it is *implicitly* really `Int 42` so that the result of every expression has type `theType`. Then functions like `+` raise errors if both arguments do not have the right constructor and their result is also wrapped with the right constructor if necessary. For example, we could think of `car` as being:

```
fun car v = case v of Pair(a,b) => a | _  => raise ... (* give some error *)
```

Since this "secret pattern-matching" is not exposed to programmers, Racket also provides which-constructor functions that programmers can use instead. For example, the primitive `pair?` can be viewed as:

```
fun pair? v = case v of Pair _ => true | _  => false
```

Finally, Racket's struct definitions do one thing you cannot quite do with ML datatype bindings: They dynamically add new constructors to a datatype.[2]

The fact that we can think of Racket in terms of `theType` suggests that anything you can do in Racket can be done, perhaps more awkwardly, in ML: The ML programmer could just program explicitly using something like the `theType` definition above.

[2]You can do this in ML with the `exn` type, but not with datatype bindings. If you could, static checking for missing pattern-matching clauses would not be possible.

What is Static Checking?

What is usually meant by "static checking" is anything done to reject a program *after* it (successfully) parses but *before* it runs. If a program does not parse, we still get an error, but we call such an error a "syntax error" or "parsing error." In contrast, an error from static checking, typically a "type error," would include things like undefined variables or using a number instead of a pair. We do static checking without any input to the program identified — it is "compile-time checking" though it is irrelevant whether the language implementation will use a compiler or an interpreter after static checking succeeds.

What static checking is performed is part of the definition of a programming language. Different languages can do different things; some languages do no static checking at all. Given a language with a particular definition, you could also use other tools that do even more static checking to try to find bugs or ensure their absence even though such tools are not part of the language definition.

The most common way to define a language's static checking is via a *type system*. When we studied ML, we gave typing rules for each language construct: Each variable had a type, the two branches of a conditional must have the same type, etc. ML's static checking is checking that these rules are followed (and in ML's case, inferring types to do so). But this is the language's *approach* to static checking (how it does it), which is different from the *purpose* of static checking (what it accomplishes). The purpose is to reject programs that "make no sense" or "may try to misuse a language feature." There are errors a type system typically does not prevent (such as array-bounds errors) and others that a type system *cannot* prevent unless given more information about what a program is supposed to do. For example, if a program puts the branches of a conditional in the wrong order or calls `+` instead of `*`, this is still a program just not the one intended.

For example, one purpose of ML's type system is to prevent passing strings to arithmetic primitives like the division operator. In contrast, Racket uses "dynamic checking" (i.e., run-time checking) by tagging each value and having the division operator check that its arguments are numbers. The ML implementation does not have to tag values for this purpose because it can rely on static checking. But as we will discuss below, the trade-off is that the static checker has to reject some programs that would not actually do anything wrong.

As ML and Racket demonstrate, the typical points at which to prevent a "bad thing" are "compile-time" and "run-time." However, it is worth realizing that there is really a continuum of eagerness about when we declare something an error. Consider for sake of example something that most type systems do not prevent statically: division-by-zero. If we have some function containing the expression `(/ 3 0)`, when could we cause an error:

- Keystroke-time: Adjust the editor so that one cannot even write down a division with a denominator of 0. This is approximate because maybe we were about to write 0.33, but we were not allowed to write the 0.
- Compile-time: As soon as we see the expression. This is approximate because maybe the context is `(if #f (/ 3 0) 42)`.
- Link-time: Once we see the function containing `(/ 3 0)` might be called from some "main" function. This is less approximate than compile-time since some code might never be used, but we still have to approximate what code may be called.
- Run-time: As soon as we execute the division.
- Even later: Rather than raise an error, we could just return some sort of value indicating division-by-zero and not raise an error until that value was used for something where we needed an actual number, like indexing into an array.

While the "even later" option might seem too permissive at first, it is exactly what floating-point computations do. `(/ 3.0 0.0)` produces `+inf.0`, which can still be computed with but cannot be converted to an exact number. In scientific computing this is very useful to avoid lots of extra cases: maybe we do something like take the tangent of $\pi/2$ but only when this will end up not being used in the final answer.

Correctness: Soundness, Completeness, Undecidability

Intuitively, a static checker is correct if it prevents what it claims to prevent — otherwise, either the language definition or the implementation of static checking needs to be fixed. But we can give a more precise description of correctness by defining the terms *soundness* and *completeness*. For both, the definition is with respect to some thing X we wish to prevent. For example, X could be "a program looks up a variable that is not in the environment."

A type system is *sound* if it never accepts a program that, when run with some input, does X.

A type system is *complete* if it never rejects a program that, no matter what input it is run with, will not do X.

A good way to understand these definitions is that *soundness prevents false negatives* and *completeness prevents false positives*. The terms *false negatives* and *false positives* come from statistics and medicine: Suppose there is a medical test for a disease, but it is not a perfect test. If the test does not detect the disease but the patient actually has the disease, then this is a false negative (the test was negative, but that is false). If the test detects the disease but the patient actually does not have the disease, then this is a false positive (the test was positive, but that's false). With static checking, the disease is "performs X when run with some input" and the test is "does the program type-check?" The terms *soundness* and *completeness* come from logic and are commonly used in the study of programming languages. A sound logic proves only true things. A complete logic proves all true things. Here, our type system is the logic and the thing we are trying to prove is "X cannot occur."

In modern languages, type systems are sound (they prevent what they claim to) but not complete (they reject programs they need not reject). Soundness is important because it lets language users and language implementers rely on X never happening. Completeness would be nice, but hopefully it is rare in practice that a program is rejected unnecessarily and in those cases, hopefully it is easy for the programmer to modify the program such that it type-checks.

Type systems are not complete because for almost anything you might like to check statically, it is *impossible* to implement a static checker that given any program in your language (a) always terminates, (b) is sound, and (c) is complete. Since we have to give up one, (c) seems like the best option (programmers do not like compilers that may not terminate).

The impossibility result is exactly the idea of *undecidability* at the heart of the study of the theory of computation. Knowing what it means that nontrivial properties of programs are undecidable is fundamental to being an educated computer scientist. The fact that undecidability directly implies the inherent approximation (i.e., incompleteness) of static checking is probably the most important ramification of undecidability. We simply cannot write a program that takes as input another program in ML/Racket/Java/etc. that always correctly answers questions such as, "will this program divide-by-zero?" "will this program treat a string as a function?" "will this program terminate?" etc.

Weak Typing

Now suppose a type system is unsound for some property X. Then to be safe the language implementation should still, at least in some cases, perform dynamic checks to prevent X from happening and the language definition should allow that these checks might fail at run-time.

But an alternative is to say it is the programmer's fault if X happens and the language definition does *not* have to check. In fact, if X happens, then the running program can do *anything*: crash, corrupt data, produce the wrong answer, delete files, launch a virus, or set the computer on fire. If a language has programs where a legal implementation is allowed to set the computer on fire (even though it probably would not), we call the language *weakly typed*. Languages where the behavior of buggy programs is more limited are called *strongly typed*. These terms are a bit unfortunate since the correctness of the type system is only part of the issue. After all, Racket is dynamically typed but nonetheless strongly typed. Moreover, a big source of actual undefined and unpredictable behavior in weakly typed languages is array-bounds errors (they need not check the bound — they can just access some other data by mistake), yet few type systems check array bounds.

C and C++ are the well-known weakly typed languages. Why are they defined this way? In short, because the designers do not want the language definition to force implementations to do all the dynamic checks that would be necessary. While there is a time cost to performing checks, the bigger problem is that the implementation has to keep around extra data (like tags on values) to do the checks and C/C++ are designed as lower-level languages where the programmer can expect extra "hidden fields" are not added.

An older now-much-rarer perspective in favor of weak typing is embodied by the saying "strong types for weak minds." The idea is that any strongly typed language is either rejecting programs statically or performing unnecessary tests dynamically (see undecidability above), so a human should be able to "overrule" the checks in places where he/she knows they are unnecessary. In reality, humans are extremely error-prone and we should welcome automatic checking even if it has to err on the side of caution for us. Moreover, type systems have gotten much more expressive over time (e.g., polymorphic) and language implementations have gotten better at optimizing away unnecessary checks (they will just never get all of them). Meanwhile, software has gotten very large, very complex, and relied upon by all of society. It is deeply problematic that 1 bug in a 30-million-line operating system written in C can make the entire computer subject to security exploits. While this is still a real problem and C the language provides little support, it is increasingly common to use other tools to do static and/or dynamic checking with C code to try to prevent such errors.

More Flexible Primitives is a Related but Different Issue

Suppose we changed ML so that the type system accepted any expression `e1 + e2` as long as `e1` and `e2` had *some* type and we changed the evaluation rules of addition to return `0` if one of the arguments did not result in a number. Would this make ML a dynamically typed language? It is "more dynamic" in the sense that the language is more lenient and some "likely" bugs are not detected as eagerly, but there is still a type system rejecting programs — we just changed the definition of what an "illegal" operation is to allow more additions. We could have similarly changed Racket to not give errors if `+` is given bad arguments. The Racket designers choose not to do so because it is likely to mask bugs without being very useful.

Other languages make different choices that report fewer errors by extending the definition of primitive operations to *not* be errors in situations like this. In addition to defining arithmetic over any kind of data, some examples are:

- Allowing out-of-bound array accesses. For example, if `arr` has fewer than 10 elements, we can still allow `arr[10]` by just returning a default value or `arr[10]=e` by making the array bigger.

- Allowing function calls with the wrong number of arguments. Extra arguments can be silently ignored. Too few arguments can be filled in with defaults chosen by the language.

These choices are matters of language design. Giving meaning to what are likely errors is often unwise — it masks errors and makes them more difficult to debug because the program runs long after some nonsense-for-the-application computation occurred. On the other hand, such "more dynamic" features are used by programmers when provided, so clearly someone is finding them useful.

For our purposes here, we just consider this a separate issue from static vs. dynamic typing. Instead of preventing some X (e.g., calling a function with too many arguments) either before the program runs or when it runs, we are changing the language semantics so that we do not prevent X at all – we allow it and extend our evaluation rules to give it a semantics.

Advantages and Disadvantages of Static Checking

Now that we know what static and dynamic typing are, let's wade into the decades-old argument about which is better. We know static typing catches many errors for you early, soundness ensures certain kinds of errors do not remain, and incompleteness means some perfectly fine programs are rejected. We will not answer definitively whether static typing is desirable (if nothing else it depends what you are checking), but we will consider seven specific claims and consider for each valid arguments made both for and against static typing.

1. Is Static or Dynamic Typing More Convenient?

The argument that dynamic typing is more convenient stems from being able to mix-and-match different kinds of data such as numbers, strings, and pairs without having to declare new type definitions or "clutter" code with pattern-matching. For example, if we want a function that returns either a number or string, we can just return a number or a string, and callers can use dynamic type predicates as necessary. In Racket, we can write:

```
(define (f y) (if (> y 0) (+ y y) "hi"))
(let ([ans (f x)]) (if (number? ans) (number->string ans) ans))
```

In contrast, the analogous ML code needs to use a datatype, with constructors in `f` and pattern-matching to use the result:

```
datatype t = Int of int | String of string
fun f y = if y > 0 then Int(y+y) else String "hi"
val _ = case f x of Int i => Int.toString i | String s => s
```

On the other hand, static typing makes it more convenient to assume data has a certain type, knowing that this assumption cannot be violated, which would lead to errors later. For a Racket function to ensure some data is, for example, a number, it has to insert an explicit dynamic check in the code, which is more work and harder to read. The corresponding ML code has no such awkwardness.

```
(define (cube x)
  (if (not (number? x))
      (error "cube expects a number")
```

```
      (* x x x)))
(cube 7)

fun cube x = x * x * x
val _ = cube 7
```

Notice that without the check in the Racket code, the actual error would arise in the body of the multiplication, which could confuse callers that did not know `cube` was implemented using multiplication.

2. Does Static Typing Prevent Useful Programs?

Dynamic typing does not reject programs that make perfect sense. For example, the Racket code below binds `'((7 . 7) . (#t . #t))` to `pair_of_pairs` without problem, but the corresponding ML code does not type-check since there is no type the ML type system can give to `f`.[3]

```
(define (f g) (cons (g 7) (g #t)))
(define pair_of_pairs (f (lambda (x) (cons x x))))

fun f g = (g 7, g true) (* does not type-check *)
val pair_of_pairs = f (fn x => (x,x))
```

Of course we can write an ML program that produces `((7,7),(true,true))`, but we may have to "work around the type-system" rather than do it the way we want.

On the other hand, dynamic typing derives its flexibility from putting a tag on every value. In ML and other statically typed languages, we can do the same thing *when we want to* by using datatypes and explicit tags. In the extreme, if you want to program like Racket in ML, you can use a datatype to represent "The One Racket Type" and insert explicit tags and pattern-matching everywhere. While this programming style would be painful to use everywhere, it proves the point that there is nothing we can do in Racket that we cannot do in ML. (We discussed this briefly already above.)

```
datatype tort = Int of int
              | String of string
              | Pair of tort * tort
              | Fun of tort -> tort
              | Bool of bool
              | ...
fun f g = (case g of Fun g' => Pair(g' (Int 7), g' (Bool true)))
val pair_of_pairs = f (Fun (fn x => Pair(x,x)))
```

Perhaps an even simpler argument in favor of static typing is that modern type systems are expressive enough that they rarely get in your way. How often do you try to write a function like `f` that does not type-check in ML?

[3] This is a limitation of ML. There are languages with more expressive forms of polymorphism that can type-check such code. But due to undecidability, there are always limitations.

3. Is Static Typing's Early Bug-Detection Important?

A clear argument in favor of static typing is that it catches bugs earlier, as soon you statically check (informally, "compile") the code. A well-known truism of software development is that bugs are easier to fix if discovered sooner, while the developer is still thinking about the code. Consider this Racket program:

```
(define (pow x)
   (lambda (y)
      (if (= y 0)
          1
          (* x (pow x (- y 1))))))
```

While the algorithm looks correct, this program has a bug: `pow` expects curried arguments, but the recursive call passes `pow` two arguments, not via currying. This bug is not discovered until testing `pow` with a `y` not equal to `0`. The equivalent ML program simply does not type-check:

```
fun pow x y = (* does not type-check *)
  if y = 0
  then 1
  else x * pow (x,y-1)
```

Because static checkers catch known kinds of errors, expert programmers can use this knowledge to focus attention elsewhere. A programmer might be quite sloppy about tupling versus currying when writing down most code, knowing that the type-checker will later give a list of errors that can be quickly corrected. This could free up mental energy to focus on other tasks, like array-bounds reasoning or higher-level algorithm issues.

A dynamic-typing proponent would argue that static checking usually catches only bugs you would catch with testing anyway. Since you still need to test your program, the additional value of catching some bugs before you run the tests is reduced. After all, the programs below do not work as exponentiation functions (they use the wrong arithmetic), ML's type system will not detect this, and testing catches this bug and would also catch the currying bug above.

```
(define (pow x) ; wrong algorithm
  (lambda (y)
     (if (= y 0)
         1
         (+ x ((pow x) (- y 1))))))

fun pow x y = (* wrong algorithm *)
   if y = 0
   then 1
   else x + pow x (y - 1)
```

4. Does Static or Dynamic Typing Lead to Better Performance?

Static typing can lead to faster code since it does not need to perform type tests at run time. In fact, much of the performance advantage may result from not storing the type tags in the first place, which takes more space and slows down constructors. In ML, there are run-time tags only where the programmer uses datatype constructors rather than everywhere.

Dynamic typing has three reasonable counterarguments. First, this sort of low-level performance does not matter in most software. Second, implementations of dynamically typed languages can and do try to *optimize away* type tests it can tell are unnecessary. For example, in `(let ([x (+ y y)]) (* x 4))`, the multiplication does not need to check that `x` and `4` are numbers and the addition can check `y` only once. While no optimizer can remove all unnecessary tests from every program (undecidability strikes again), it may be easy enough in practice for the parts of programs where performance matters. Third, if programmers in statically typed languages have to work around type-system limitations, then those workarounds can erode the supposed performance advantages. After all, ML programs that use datatypes have tags too.

5. Does Static or Dynamic Typing Make Code Reuse Easier?

Dynamic typing arguably makes it easier to reuse library functions. After all, if you build lots of different kinds of data out of cons cells, you can just keep using `car`, `cdr`, `cadr`, etc. to get the pieces out rather than defining lots of different getter functions for each data structure. On the other hand, this can mask bugs. For example, suppose you accidentally pass a list to a function that expects a tree. If `cdr` works on both of them, you might just get the wrong answer or cause a mysterious error later, whereas using different types for lists and trees could catch the error sooner.

This is really an interesting design issue more general than just static versus dynamic typing. Often it is good to reuse a library or data structure you already have especially since you get to reuse all the functions available for it. Other times it makes it too difficult to separate things that are really different conceptually so it is better to define a new type. That way the static type-checker or a dynamic type-test can catch when you put the wrong thing in the wrong place.

6. Is Static or Dynamic Typing Better for Prototyping?

Early in a software project, you are developing a prototype, often at the same time you are changing your views on what the software will do and how the implementation will approach doing it.

Dynamic typing is often considered better for prototyping since you do not need to expend energy defining the types of variables, functions, and data structures when those decisions are in flux. Moreover, you may know that part of your program does not yet make sense (it would not type-check in a statically typed language), but you want to run the rest of your program anyway (e.g., to test the parts you just wrote).

Static typing proponents may counter that it is never too early to document the types in your software design even if (perhaps especially if) they are unclear and changing. Moreover, commenting out code or adding stubs like pattern-match branches of the form `_ => raise Unimplemented` is often easy and documents what parts of the program are known not to work.

7. Is Static or Dynamic Typing Better for Code Evolution?

A lot of effort in software engineering is spent maintaining working programs, by fixing bugs, adding new features, and in general evolving the code to make some change.

Dynamic typing is sometimes more convenient for code evolution because we can change code to be more permissive (accept arguments of more types) without having to change any of the pre-existing clients of the code. For example, consider changing this simple function:

```
(define (f x) (* 2 x))
```

to this version, which can process numbers or strings:

```
(define (f x)
  (if (number? x)
      (* 2 x)
      (string-append x x)))
```

No existing caller, which presumably uses `f` with numbers, can tell this change was made, but new callers can pass in strings or even values where they do not know if the value is a number or a string. If we make the analogous change in ML, no existing callers will type-check since they all must wrap their arguments in the `Int` constructor and use pattern-matching on the function result:

```
fun f x = 2 * x

datatype t = Int of int | String of string
fun f x =
  case f x of
    Int i    => Int (2 * i)
  | String s => String (s ^ s)
```

On the other hand, static type-checking is very useful when evolving code to catch bugs that the evolution introduces. When we change the type of a function, all callers no longer type-check, which means the type-checker gives us an invaluable "to-do list" of all the call-sites that need to change. By this argument, the safest way to evolve code is to change the types of any functions whose specification is changing, which is an argument for capturing as much of your specification as you can in the types.

A particularly good example in ML is when you need to add a new constructor to a datatype. If you did not use wildcard patterns, then you will get a warning for all the case-expressions that use the datatype.

As valuable as the "to-do list from the type-checker" is, it can be frustrating that the program will not run until all items on the list are addressed or, as discussed under the previous claim, you use comments or stubs to remove the parts not yet evolved.

Optional: `eval` and `quote`

(This short section barely scratches the surface of programming with `eval`. It really just introduces the concept. Interested students are encouraged to learn more on their own.)

There is one sense where it is slightly fair to say Racket is an interpreted language: it has a primitive `eval` that can take a representation of a program at run-time and evaluate it. For example, this program, which is poor style because there are much simpler ways to achieve its purpose, may or may not print something depending on `x`:

```
(define (make-some-code y)
   (if y
       (list 'begin (list 'print "hi") (list '+ 4 2))
       (list '+ 5 3)))
(define (f x)
  (eval (make-some-code x)))
```

The Racket function `make-some-code` is strange: It does *not* ever print or perform an addition. All it does is return some list containing symbols, strings, and numbers. For example, if called with `#t`, it returns

```
'(begin (print "hi") (+ 4 2))
```

This is nothing more and nothing less than a three element list where the first element is the symbol `begin`. It is just Racket data. But if we look at this data, it looks just like a Racket program we could run. The nested lists together are a perfectly good *representation* of a Racket expression that, if evaluated, would print `"hi"` and have a result of 6.

The `eval` primitive takes such a representation and, at run-time, evaluates it. We can perform whatever computation we want to generate the data we pass to `eval`. As a simple example, we could append together two lists, like `(list '+ 2)` and `(list 3 4)`. If we call `eval` with the result `'(+ 2 3 4)`, i.e., a 4-element list, then `eval` returns 9.

Many languages have `eval`, many do not, and what the appropriate idioms for using it are is a subject of significant dispute. Most would agree it tends to get overused but is also a really powerful construct that is sometimes what you want.

Can a compiler-based language implementation (notice we did not say "compiled language") deal with `eval`? Well, it would need to have the compiler or an interpreter around at run-time since it cannot know in advance what might get passed to `eval`. An interpreter-based language implementation would also need an interpreter or compiler around at run-time, but, of course, it *already* needs that to evaluate the "regular program."

In languages like Javascript and Ruby, we do not have the convenience of Racket syntax where programs and lists are so similar-looking that `eval` can take a list-representation that looks exactly like Racket syntax. Instead, in these languages, `eval` takes a string and interprets it as concrete syntax by first parsing it and then running it. Regardless of language, `eval` will raise an error if given an ill-formed program or a program that raises an error.

In Racket, it is painful and unnecessary to write `make-some-code` the way we did. Instead, there is a special form `quote` that treats everything under it as symbols, numbers, lists, etc., *not* as functions to be called. So we could write:

```
(define (make-some-code y)
  (if y
      (quote (begin (print "hi") (+ 4 2)))
      (quote (+ 5 3))))
```

Interestingly, `eval` and `quote` are inverses: For any expression `e`, we should have `(eval (quote e))` as a terrible-style but equivalent way to write `e`.

Often `quote` is "too strong" — we want to quote *most* things, but it is convenient to evaluate some code inside of what is mostly syntax we are building. Racket has quasiquote and unquote for doing this (see the manual if interested) and Racket's linguistic predecessors have had this functionality for decades. In modern scripting languages, one often sees analogous functionality: the ability to embed expression evaluation inside a string (which one might or might not then call `eval` on, just as one might or might not use a Racket quote expression to build something for `eval`). This feature is sometimes called *interpolation* in scripting languages, but it is just *quasiquoting*.

Summary

Why Functional Programming Matters

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN. rjmh@cs.chalmers.se

This paper dates from 1984, and circulated as a Chalmers memo for many years. Slightly revised versions appeared in 1989 and 1990 as [Hug90] and [Hug89]. This version is based on the original Chalmers memo `nroff` *source, lightly edited for LaTeX and to bring it closer to the published versions, and with one or two errors corrected. Please excuse the slightly old-fashioned type-setting, and the fact that the examples are not in Haskell!*

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularised. Functional languages push those limits back. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). Since modularity is the key to successful programming, functional languages are vitally important to the real world.

1 Introduction

This paper is an attempt to demonstrate to the "real world" that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are.

Functional programming is so called because a program consists entirely of functions. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. These functions are much like ordinary mathematical functions, and in this paper will be defined by ordinary equations. Our

notation follows Turner's language Miranda(TM) [Tur85], but should be readable with no prior knowledge of functional languages. (Miranda is a trademark of Research Software Ltd.)

The special characteristics and advantages of functional programming are often summed up more or less as follows. Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side-effect can change the value of an expression, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts.

Such a catalogue of "advantages" is all very well, but one must not be surprised if outsiders don't take it too seriously. It says a lot about what functional programming is *not* (it has no assignment, no side effects, no flow of control) but not much about what it is. The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous. To those more interested in material benefits, these "advantages" are not very convincing.

Functional programmers argue that there *are* great material benefits - that a functional programmer is an order of magnitude more productive than his conventional counterpart, because functional programs are an order of magnitude shorter. Yet why should this be? The only faintly plausible reason one can suggest on the basis of these "advantages" is that conventional programs consist of 90% assignment statements, and in functional programs these can be omitted! This is plainly ridiculous. If omitting assignment statements brought such enormous benefits then FORTRAN programmers would have been doing it for twenty years. It is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be.

Even a functional programmer should be dissatisfied with these so-called advantages, because they give him no help in exploiting the power of functional languages. One cannot write a program which is particularly lacking in assignment statements, or particularly referentially transparent. There is no yardstick of program quality here, and therefore no ideal to aim at.

Clearly this characterisation of functional programming is inadequate. We must find something to put in its place - something which not only explains the power of functional programming, but also gives a clear indication of what the functional programmer should strive towards.

2 An Analogy with Structured Programming

It is helpful to draw an analogy between functional and structured programming. In the past, the characteristics and advantages of structured programming have been summed up more or less as follows. Structured programs contain no **goto** statements. Blocks in a structured program do not have multiple entries or exits. Structured programs are more tractable mathematically than their unstructured counterparts. These "advantages" of structured programming are very similar in spirit to the "advantages" of functional programming we discussed earlier. They are essentially negative statements, and have led to much fruitless argument about "essential **goto**s" and so on.

With the benefit of hindsight, it is clear that these properties of structured programs, although helpful, do not go to the heart of the matter. The most important difference between structured and unstructured programs is that structured programs are designed in a modular way. Modular design brings with it great productivity improvements. First of all, small modules can be coded quickly and easily. Secondly, general purpose modules can be re-used, leading to faster development of subsequent programs. Thirdly, the modules of a program can be tested independently, helping to reduce the time spent debugging.

The absence of **goto**s, and so on, has very little to do with this. It helps with "programming in the small", whereas modular design helps with "programming in the large". Thus one can enjoy the benefits of structured programming in FORTRAN or assembly language, even if it is a little more work.

It is now generally accepted that modular design is the key to successful programming, and languages such as Modula-II [Wir82], Ada [oD80] and Standard ML [MTH90] include features specifically designed to help improve modularity. However, there is a very important point that is often missed. When writing a modular program to solve a problem, one first divides the problem into sub-problems, then solves the sub-problems and combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase ones ability to modularise a problem conceptually, one must provide new kinds of glue in the programming language. Complicated scope rules and provision for separate compilation only help with clerical details; they offer no new conceptual tools for decomposing problems.

One can appreciate the importance of glue by an analogy with carpentry. A chair can be made quite easily by making the parts - seat, legs, back etc. - and sticking them together in the right way. But this depends on an ability to make joints and wood glue. Lacking that ability, the only way to make a chair is to carve it in one piece out of a solid block of wood, a much harder task. This example demonstrates both the enormous power of modularisation and the importance of having the right glue.

Now let us return to functional programming. We shall argue in the remainder of this paper that functional languages provide two new, very important kinds of glue. We shall give many examples of programs that can be modularised in new ways, and thereby greatly simplified. This is the key to functional

programming's power - it allows greatly improved modularisation. It is also the goal for which functional programmers must strive - smaller and simpler and more general modules, glued together with the new glues we shall describe.

3 Glueing Functions Together

The first of the two new kinds of glue enables simple functions to be glued together to make more complex ones. It can be illustrated with a simple list-processing problem - adding up the elements of a list. We define lists by

```
listof X ::= nil | cons X (listof X)
```

which means that a list of Xs (whatever X is) is either nil, representing a list with no elements, or it is a cons of an X and another list of Xs. A cons represents a list whose first element is the X and whose second and subsequent elements are the elements of the other list of Xs. X here may stand for any type - for example, if X is "integer" then the definition says that a list of integers is either empty or a cons of an integer and another list of integers. Following normal practice, we will write down lists simply by enclosing their elements in square brackets, rather than by writing conses and nils explicitly. This is simply a shorthand for notational convenience. For example,

```
[]              means   nil
[1]             means   cons 1 nil
[1,2,3]         means   cons 1 (cons 2 (cons 3 nil))
```

The elements of a list can be added up by a recursive function *sum*. Sum must be defined for two kinds of argument: an empty list (nil), and a cons. Since the sum of no numbers is zero, we define

```
sum nil = 0
```

and since the sum of a cons can be calculated by adding the first element of the list to the sum of the others, we can define

```
sum (cons num list) = num + sum list
```

Examining this definition, we see that only the boxed parts below are specific to computing a sum.

```
          +---+
sum nil = | 0 |
          +---+
                            +---+
sum (cons num list) = num   | + | sum list
                            +---+
```

This means that the computation of a sum can be modularised by glueing together a general recursive pattern and the boxed parts. This recursive pattern is conventionally called *reduce* and so sum can be expressed as

```
sum = reduce add 0
```

where for convenience reduce is passed a two argument function *add* rather than an operator. Add is just defined by

```
add x y = x + y
```

The definition of reduce can be derived just by parameterising the definition of sum, giving

```
(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Here we have written brackets around (reduce f x) to make it clear that it replaces sum. Conventionally the brackets are omitted, and so ((reduce f x) l) is written as (reduce f x l). A function of 3 arguments such as reduce, applied to only 2 is taken to be a function of the one remaining argument, and in general, a function of n arguments applied to only $m(< n)$ is taken to be a function of the $n - m$ remaining ones. We will follow this convention in future.

Having modularised sum in this way, we can reap benefits by re-using the parts. The most interesting part is reduce, which can be used to write down a function for multiplying together the elements of a list with no further programming:

```
product = reduce multiply 1
```

It can also be used to test whether any of a list of booleans is true

```
anytrue = reduce or false
```

or whether they are all true

```
alltrue = reduce and true
```

One way to understand (reduce f a) is as a function that replaces all occurrences of cons in a list by f, and all occurrences of nil by a. Taking the list [1,2,3] as an example, since this means

```
cons 1 (cons 2 (cons 3 nil))
```

then (reduce add 0) converts it into

```
add 1 (add 2 (add 3 0))  =  6
```

and (reduce multiply 1) converts it into

```
multiply 1 (multiply 2 (multiply 3 1))  =  6
```

Now it is obvious that (reduce cons nil) just copies a list. Since one list can be appended to another by consing its elements onto the front, we find

```
append a b = reduce cons b a
```

As an example,

```
append [1,2] [3,4] = reduce cons [3,4] [1,2]
                   = (reduce cons [3,4]) (cons 1 (cons 2 nil))
                   = cons 1 (cons 2 [3,4]))
                        (replacing cons by cons and nil by [3,4])
                   = [1,2,3,4]
```

A function to double all the elements of a list could be written as

```
        doubleall = reduce doubleandcons nil
where   doubleandcons num list = cons (2*num) list
```

Doubleandcons can be modularised even further, first into

```
        doubleandcons = fandcons double
where   double n = 2*n
        fandcons f el list = cons (f el) list
```

and then by

```
        fandcons f = cons . f
```

where "." (function composition, a standard operator) is defined by

```
        (f . g) h = f (g h)
```

We can see that the new definition of fandcons is correct by applying it to some arguments:

```
        fandcons f el = (cons . f) el
                      = cons (f el)
so      fandcons f el list = cons (f el) list
```

The final version is

```
        doubleall = reduce (cons . double) nil
```

With one further modularisation we arrive at

```
        doubleall = map double
        map f = reduce (cons . f) nil
```

where map applies any function f to all the elements of a list. Map is another generally useful function.

We can even write down a function to add up all the elements of a matrix, represented as a list of lists. It is

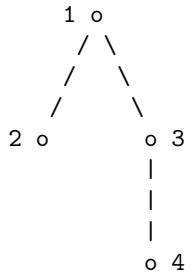
```
        summatrix = sum . map sum
```

The map sum uses sum to add up all the rows, and then the left-most sum adds up the row totals to get the sum of the whole matrix.

These examples should be enough to convince the reader that a little modularisation can go a long way. By modularising a simple function (sum) as a combination of a "higher order function" and some simple arguments, we have arrived at a part (reduce) that can be used to write down many other functions on lists with no more programming effort. We do not need to stop with functions on lists. As another example, consider the datatype of ordered labelled trees, defined by

```
treeof X ::= node X (listof (treeof X))
```

This definition says that a tree of Xs is a node, with a label which is an X, and a list of subtrees which are also trees of Xs. For example, the tree



would be represented by

```
node 1
    (cons (node 2 nil)
          (cons (node 3
                      (cons (node 4 nil) nil))
                nil))
```

Instead of considering an example and abstracting a higher order function from it, we will go straight to a function redtree analogous to reduce. Recall that reduce took two arguments, something to replace cons with, and something to replace nil with. Since trees are built using node, cons and nil, redtree must take three arguments - something to replace each of these with. Since trees and lists are of different types, we will have to define two functions, one operating on each type. Therefore we define

```
redtree f g a (node label subtrees) =
        f label (redtree' f g a subtrees)
redtree' f g a (cons subtree rest) =
        g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a nil = a
```

Many interesting functions can be defined by glueing redtree and other functions together. For example, all the labels in a tree of numbers can be added together using

```
sumtree = redtree add add 0
```

Taking the tree we wrote down earlier as an example, sumtree gives

```
add 1
    (add (add 2 0)
         (add (add 3
                   (add (add 4 0) 0))
              0))
= 10
```

A list of all the labels in a tree can be computed using

```
labels = redtree cons append nil
```

The same example gives

```
cons 1
    (append (cons 2 nil)
            (append (cons 3
                          (append (cons 4 nil) nil))
                    nil))
= [1,2,3,4]
```

Finally, one can define a function analogous to map which applies a function f to all the labels in a tree:

```
maptree f = redtree (node . f) cons nil
```

All this can be achieved because functional languages allow functions which are indivisible in conventional programming languages to be expressed as a combination of parts - a general higher order function and some particular specialising functions. Once defined, such higher order functions allow many operations to be programmed very easily. Whenever a new datatype is defined higher order functions should be written for processing it. This makes manipulating the datatype easy, and also localises knowledge about the details of its representation. The best analogy with conventional programming is with extensible languages - it is as though the programming language can be extended with new control structures whenever desired.

4 Glueing Programs Together

The other new kind of glue that functional languages provide enables whole programs to be glued together. Recall that a complete functional program is just a function from its input to its output. If f and g are such programs, then (g . f) is a program which, when applied to its input, computes

```
g (f input)
```

The program f computes its output which is used as the input to program g. This might be implemented conventionally by storing the output from f in a temporary file. The problem with this is that the temporary file might occupy so much memory that it is impractical to glue the programs together in this way. Functional languages provide a solution to this problem. The two programs f and g are run together in strict synchronisation. F is only started once g tries to read some input, and only runs for long enough to deliver the output g is trying to read. Then f is suspended and g is run until it tries to read another input. As an added bonus, if g terminates without reading all of f's output then f is aborted. F can even be a non-terminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies - a powerful modularisation.

Since this method of evaluation runs f as little as possible, it is called "lazy evaluation". It makes it practical to modularise a program as a generator which constructs a large number of possible answers, and a selector which chooses the appropriate one. While some other systems allow programs to be run together in this manner, only functional languages use lazy evaluation uniformly for every function call, allowing any part of a program to be modularised in this way. Lazy evaluation is perhaps the most powerful tool for modularisation in the functional programmer's repertoire.

4.1 Newton-Raphson Square Roots

We will illustrate the power of lazy evaluation by programming some numerical algorithms. First of all, consider the Newton-Raphson algorithm for finding square roots. This algorithm computes the square root of a number N by starting from an initial approximation a0 and computing better and better ones using the rule

```
a(n+1) = (a(n) + N/a(n)) / 2
```

If the approximations converge to some limit a, then

```
        a = (a + N/a) / 2
so      2a = a + N/a
        a = N/a
        a*a = N
        a = squareroot(N)
```

In fact the approximations converge rapidly to a limit. Square root programs take a tolerance (eps) and stop when two successive approximations differ by less than eps.

The algorithm is usually programmed more or less as follows:

```
C   N IS CALLED ZN HERE SO THAT IT HAS THE RIGHT TYPE
      X = A0
      Y = A0 + 2.*EPS
```

```
C   THE VALUE OF Y DOES NOT MATTER SO LONG AS ABS(X-Y).GT.EPS
100   IF (ABS(X-Y).LE.EPS) GOTO 200
      Y = X
      X = (X + ZN/X) / 2.
      GOTO 100
200   CONTINUE
C   THE SQUARE ROOT OF ZN IS NOW IN X
```

This program is indivisible in conventional languages. We will express it in a more modular form using lazy evaluation, and then show some other uses to which the parts may be put.

Since the Newton-Raphson algorithm computes a sequence of approximations it is natural to represent this explicitly in the program by a list of approximations. Each approximation is derived from the previous one by the function

```
next N x = (x + N/x) / 2
```

so (next N) is the function mapping one approximation onto the next. Calling this function f, the sequence of approximations is

```
[a0, f a0, f(f a0), f(f(f a0)), ..]
```

We can define a function to compute this:

```
repeat f a = cons a (repeat f (f a))
```

so that the list of approximations can be computed by

```
repeat (next N) a0
```

Repeat is an example of a function with an "infinite" output - but it doesn't matter, because no more approximations will actually be computed than the rest of the program requires. The infinity is only potential: all it means is that any number of approximations can be computed if required, repeat itself places no limit.

The remainder of a square root finder is a function *within*, that takes a tolerance and a list of approximations and looks down the list for two successive approximations that differ by no more than the given tolerance. It can be defined by

```
within eps (cons a (cons b rest)) =
        = b,                          if abs(a-b) <= eps
        = within eps (cons b rest), otherwise
```

Putting the parts together,

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

Now that we have the parts of a square root finder, we can try combining them in different ways. One modification we might wish to make is to wait for the ratio between successive approximations to approach one, rather than for the

difference to approach zero. This is more appropriate for very small numbers (when the difference between successive approximations is small to start with) and for very large ones (when rounding error could be much larger than the tolerance). It is only necessary to define a replacement for within:

```
relative eps (cons a (cons b rest)) =
        = b,                          if abs(a-b) <= eps*abs b
        = relative eps (cons b rest), otherwise
```

Now a new version of sqrt can be defined by

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

It is not necessary to rewrite the part that generates approximations.

4.2 Numerical Differentiation

We have re-used the sequence of approximations to a square root. Of course, it is also possible to re-use within and relative with any numerical algorithm that generates a sequence of approximations. We will do so in a numerical differentiation algorithm.

The result of differentiating a function at a point is the slope of the function's graph at that point. It can be estimated quite easily by evaluating the function at the given point and at another point nearby and computing the slope of a straight line between the two points. This assumes that, if the two points are close enough together then the graph of the function will not curve much in between. This gives the definition

```
easydiff f x h = (f(x+h)-f x) / h
```

In order to get a good approximation the value of h should be very small. Unfortunately, if h is too small then the two values f(x+h) and f(x) are very close together, and so the rounding error in the subtraction may swamp the result. How can the right value of h be chosen? One solution to this dilemma is to compute a sequence of approximations with smaller and smaller values of h, starting with a reasonably large one. Such a sequence should converge to the value of the derivative, but will become hopelessly inaccurate eventually due to rounding error. If (within eps) is used to select the first approximation that is accurate enough then the risk of rounding error affecting the result can be much reduced. We need a function to compute the sequence:

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

Here h0 is the initial value of h, and successive values are obtained by repeated halving. Given this function, the derivative at any point can be computed by

```
within eps (differentiate h0 f x)
```

Even this solution is not very satisfactory because the sequence of approximations converges fairly slowly. A little simple mathematics can help here. The elements of the sequence can be expressed as

```
the right answer + an error term involving h
```

and it can be shown theoretically that the error term is roughly proportional to a power of h, so that it gets smaller as h gets smaller. Let the right answer be A, and the error term be B*h**n. Since each approximation is computed using a value of h twice that used for the next one, any two successive approximations can be expressed as

```
        a(i)   = A + B*(2**n)*(h**n)
and     a(i+1) = A + B*(h**n)
```

Now the error term can be eliminated. We conclude

```
            a(i+1)*(2**n) - a(i)
        A = --------------------
                   2**n - 1
```

Of course, since the error term is only roughly a power of h this conclusion is also approximate, but it is a much better approximation. This improvement can be applied to all successive pairs of approximations using the function

```
elimerror n (cons a (cons b rest)) =
        = cons ((b*(2**n)-a)/(2**n-1)) (elimerror n (cons b rest))
```

Eliminating error terms from a sequence of approximations yields another sequence which converges much more rapidly.

One problem remains before we can use elimerror - we have to know the right value of n. This is difficult to predict in general, but is easy to measure. It is not difficult to show that the following function estimates it correctly, but we won't include the proof here.

```
order (cons a (cons b (cons c rest))) =
        = round(log2( (a-c)/(b-c) - 1 ))
round x = x rounded to the nearest integer
log2 x = the logarithm of x to the base 2
```

Now a general function to improve a sequence of approximations can be defined:

```
improve s = elimerror (order s) s
```

The derivative of a function f can be computed more efficiently using improve, as follows

```
within eps (improve (differentiate h0 f x))
```

Improve only works on sequences of approximations which are computed using a parameter h, which is halved between each approximation. However, if it is applied to such a sequence its result is also such a sequence! This means that a sequence of approximations can be improved more than once. A different error term is eliminated each time, and the resulting sequences converge faster and faster. So, one could compute a derivative very efficiently using

```
within eps (improve (improve (improve (differentiate h0 f x))))
```

In numerical analysts terms, this is likely to be a fourth order method, and gives an accurate result very quickly. One could even define

```
super s = map second (repeat improve s)
second (cons a (cons b rest)) = b
```

which uses repeat improve to get a sequence of more and more improved sequences of approximations, and constructs a new sequence of approximations by taking the second approximation from each of the improved sequences (it turns out that the second one is the best one to take - it is more accurate than the first and doesn't require any extra work to compute). This algorithm is really very sophisticated - it uses a better and better numerical method as more and more approximations are computed. One could compute derivatives very efficiently indeed with the program:

```
within eps (super (differentiate h0 f x))
```

This is probably a case of using a sledge-hammer to crack a nut, but the point is that even an algorithm as sophisticated as super is easily expressed when modularised using lazy evaluation.

4.3 Numerical Integration

The last example we will discuss in this section is numerical integration. The problem may be stated very simply: given a real valued function f of one real argument, and two end-points a and b, estimate the area under the curve f describes between the end-points. The easiest way to estimate the area is to assume that f is nearly a straight line, in which case the area would be

```
easyintegrate f a b = (f a + f b)*(b-a)/2
```

Unfortunately this estimate is likely to be very inaccurate unless a and b are close together. A better estimate can be made by dividing the interval from a to b in two, estimating the area on each half, and adding the results together. We can define a sequence of better and better approximations to the value of the integral by using the formula above for the first approximation, and then adding together better and better approximations to the integrals on each half to calculate the others. This sequence is computed by the function

```
integrate f a b = cons (easyintegrate f a b)
                       (map addpair (zip (integrate f a mid)
                                         (integrate f mid b)))
        where mid = (a+b)/2
```

Zip is another standard list-processing function. It takes two lists and returns a list of pairs, each pair consisting of corresponding elements of the two lists. Thus the first pair consists of the first element of the first list and the first element of the second, and so on. Zip can be defined by

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```

In integrate, zip computes a list of pairs of corresponding approximations to the integrals on the two sub-intervals, and map addpair adds the elements of the pairs together to give a list of approximations to the original integral.

Actually, this version of integrate is rather inefficient because it continually recomputes values of f. As written, easyintegrate evaluates f at a and at b, and then the recursive calls of integrate re-evaluate each of these. Also, (f mid) is evaluated in each recursive call. It is therefore preferable to use the following version which never recomputes a value of f.

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb = cons ((fa+fb)*(b-a)/2)
                        (map addpair (zip (integ f a m fa fm)
                                          (integ f m b fm fb)))
        where m = (a+b)/2
              fm = f m
```

Integrate computes an infinite list of better and better approximations to the integral, just as differentiate did in the section above. One can therefore just write down integration routines that integrate to any required accuracy, as in

```
within eps (integrate f a b)
relative eps (integrate f a b)
```

This integration algorithm suffers from the same disadvantage as the first differentiation algorithm in the preceding sub-section - it converges rather slowly. Once again, it can be improved. The first approximation in the sequence is computed (by easyintegrate) using only two points, with a separation of b-a. The second approximation also uses the mid-point, so that the separation between neighbouring points is only (b-a)/2. The third approximation uses this method on each half-interval, so the separation between neighbouring points is only (b-a)/4. Clearly the separation between neighbouring points is halved between each approximation and the next. Taking this separation as "h", the sequence is a candidate for improvement using the "improve" function defined in the preceding section. Therefore we can now write down quickly converging sequences of approximations to integrals, for example

```
super (integrate sin 0 4)
improve (integrate f 0 1)
where f x = 1/(1+x*x)
```

(This latter sequence is an eighth order method for computing pi/4. The second approximation, which requires only five evaluations of f to compute, is correct to five decimal places).

In this section we have taken a number of numerical algorithms and programmed them functionally, using lazy evaluation as glue to stick their parts together. Thanks to this, we were able to modularise them in new ways, into generally useful functions such as within, relative and improve. By combining these parts in various ways we programmed some quite good numerical algorithms very simply and easily.

5 An Example from Artificial Intelligence

We have argued that functional languages are powerful primarily because they provide two new kinds of glue: higher-order functions and lazy evaluation. In this section we take a larger example from Artificial Intelligence and show how it can be programmed quite simply using these two kinds of glue.

The example we choose is the alpha-beta "heuristic", an algorithm for estimating how good a position a game-player is in. The algorithm works by looking ahead to see how the game might develop, but avoids pursuing unprofitable lines.

Let game-positions be represented by objects of the type "position". This type will vary from game to game, and we assume nothing about it. There must be some way of knowing what moves can be made from a position: assume that there is a function

```
moves: position -> listof position
```

that takes a game-position as its argument and returns the list of all positions that can be reached from it in one move. Taking noughts and crosses (tic-tac-toe) as an example,

```
       | |        X| |    |X|    | |
      -+-+-       -+-+-  -+-+-  -+-+-
moves | |    =  [ | | ,  | | ,  |X| ]
      -+-+-       -+-+-  -+-+-  -+-+-
       | |         | |    | |    | |

       | |        O| |    |O|
      -+-+-       -+-+-  -+-+-
moves |X|    =  [ |X| ,  |X| ]
      -+-+-       -+-+-  -+-+-
       | |         | |    | |
```

This assumes that it is always possible to tell which player's turn it is from a position. In noughts and crosses this can be done by counting the noughts and crosses, in a game like chess one would have to include the information explicitly in the type "position".

Given the function moves, the first step is to build a game tree. This is a tree in which the nodes are labelled by positions, such that the children of a node are labelled with the positions that can be reached in one move from that node. That is, if a node is labelled with position p, then its children are labelled with the positions in (moves p). A game tree may very well be infinite, if it is possible for a game to go on for ever with neither side winning. Game trees are exactly like the trees we discussed in section 2 - each node has a label (the position it represents) and a list of subnodes. We can therefore use the same datatype to represent them.

A game tree is built by repeated applications of moves. Starting from the root position, moves is used to generate the labels for the sub-trees of the root. Moves is then used again to generate the sub-trees of the sub-trees and so on. This pattern of recursion can be expressed as a higher-order function,

```
reptree f a = node a (map (reptree f) (f a))
```

Using this function another can be defined which constructs a game tree from a particular position

```
gametree p = reptree moves p
```

For an example, look at figure 1. The higher-order function used here (reptree) is analogous to the function repeat used to construct infinite lists in the preceding section.

The alpha-beta algorithm looks ahead from a given position to see whether the game will develop favourably or unfavourably, but in order to do so it must be able to make a rough estimate of the value of a position without looking ahead. This "static evaluation" must be used at the limit of the look-ahead, and may be used to guide the algorithm earlier. The result of the static evaluation is a measure of the promise of a position from the computer's point of view (assuming that the computer is playing the game against a human opponent). The larger the result, the better the position for the computer. The smaller the result, the worse the position. The simplest such function would return (say) +1 for positions where the computer has already won, -1 for positions where the computer has already lost, and 0 otherwise. In reality, the static evaluation function measures various things that make a position "look good", for example material advantage and control of the centre in chess. Assume that we have such a function,

```
static: position -> number
```

Since a game-tree is a (treeof position), it can be converted into a (treeof number) by the function (maptree static), which statically evaluates all the positions in the tree (which may be infinitely many). This uses the function maptree defined in section 2.

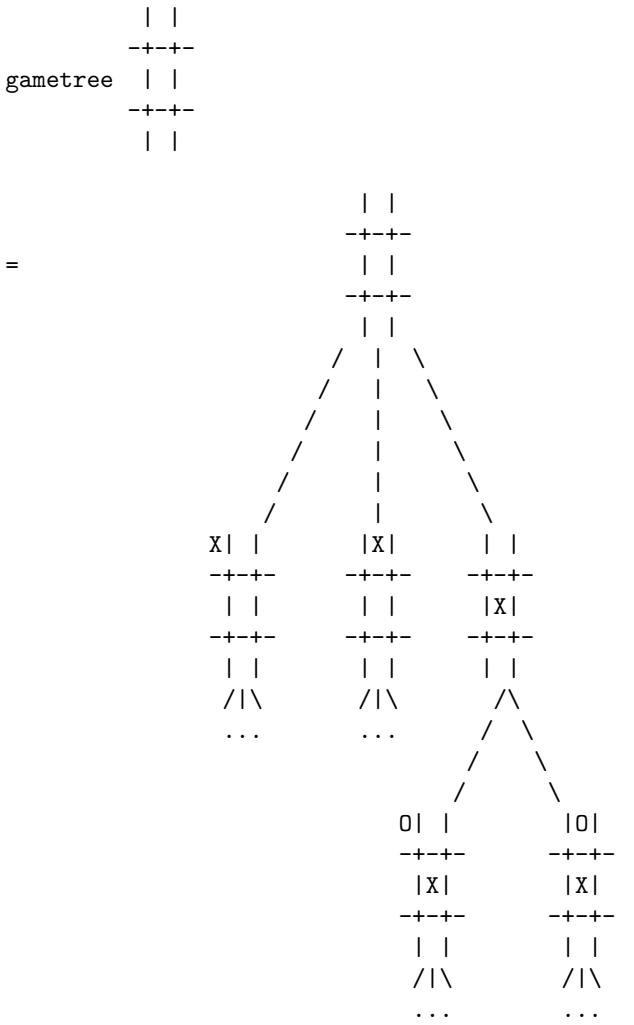


Figure 1: An Example of a Game-Tree.

Given such a tree of static evaluations, what is the true value of the positions in it? In particular, what value should be ascribed to the root position? Not its static value, since this is only a rough guess. The value ascribed to a node must be determined from the true values of its subnodes. This can be done by assuming that each player makes the best moves he can. Remembering that a high value means a good position for the computer, it is clear that when it is the computer's move from any position, it will choose the move leading to the sub-node with the maximum true value. Similarly, the opponent will choose the move leading to the sub-node with the minimum true value. Assuming that the computer and its opponent alternate turns, the true value of a node is computed by the function maximise if it is the computer's turn and minimise if it is not:

```
maximise (node n sub) = max (map minimise sub)
minimise (node n sub) = min (map maximise sub)
```

Here max and min are functions on lists of numbers that return the maximum and minimum of the list respectively. These definitions are not complete because they recurse for ever - there is no base case. We must define the value of a node with no successors, and we take it to be the static evaluation of the node (its label). Therefore the static evaluation is used when either player has already won, or at the limit of look-ahead. The complete definitions of maximise and minimise are

```
maximise (node n nil) = n
maximise (node n sub) = max (map minimise sub)
minimise (node n nil) = n
minimise (node n sub) = min (map maximise sub)
```

One could almost write down a function at this stage that would take a position and return its true value. This would be:

```
evaluate = maximise . maptree static . gametree
```

There are two problems with this definition. First of all, it doesn't work for infinite trees. Maximise keeps on recursing until it finds a node with no subtrees - an end to the tree. If there is no end then maximise will return no result. The second problem is related - even finite game trees (like the one for noughts and crosses) can be very large indeed. It is unrealistic to try to evaluate the whole of the game tree - the search must be limited to the next few moves. This can be done by pruning the tree to a fixed depth,

```
prune 0 (node a x) = node a nil
prune n (node a x) = node a (map (prune (n-1)) x)
```

(prune n) takes a tree and "cuts off" all nodes further than n from the root. If a game tree is pruned it forces maximise to use the static evaluation for nodes at depth n, instead of recursing further. Evaluate can therefore be defined by

```
evaluate = maximise . maptree static . prune 5 . gametree
```

which looks (say) 5 moves ahead.

Already in this development we have used higher-order functions and lazy evaluation. Higher order functions reptree and maptree allow us to construct and manipulate game trees with ease. More importantly, lazy evaluation permits us to modularise evaluate in this way. Since gametree has a potentially infinite result, this program would never terminate without lazy evaluation. Instead of writing

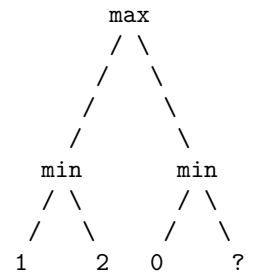
```
prune 5 . gametree
```

we would have to fold these two functions together into one which only constructed the first five levels of the tree. Worse, even the first five levels may be too large to be held in memory at one time. In the program we have written, the function

```
maptree static . prune 5 . gametree
```

only constructs parts of the tree as maximise requires them. Since each part can be thrown away (reclaimed by the garbage collector) as soon as maximise has finished with it, the whole tree is never resident in memory. Only a small part of the tree is stored at a time. The lazy program is therefore efficient. Since this efficiency depends on an interaction between maximise (the last function in the chain of compositions) and gametree (the first), it could only be achieved without lazy evaluation by folding all the functions in the chain together into one big one. This is a drastic reduction in modularity, but it is what is usually done. We can make improvements to this evaluation algorithm by tinkering with each part: this is relatively easy. A conventional programmer must modify the entire program as a unit, which is much harder.

So far we have only described simple minimaxing. The heart of the alpha-beta algorithm is the observation that one can often compute the value of maximise or minimise without looking at the whole tree. Consider the tree:



Strangely enough, it is unnecessary to know the value of the question mark in order to evaluate the tree. The left minimum evaluates to 1, but the right minimum clearly evaluates to something less than or equal to 0. Therefore the maximum of the two minima must be 1. This observation can be generalised and built into maximise and minimise.

The first step is to separate maximise into an application of max to a list of numbers; that is, we decompose maximise as

```
maximise = max . maximise'
```

(Minimise is decomposed in a similar way. Since minimise and maximise are entirely symmetrical we shall discuss maximise and assume that minimise is treated similarly). Once decomposed in this way, maximise can use minimise' rather than minimise itself, to discover which numbers minimise would take the minimum of. It may then be able to discard some of the numbers without looking at them. Thanks to lazy evaluation, if maximise doesn't look at all of the list of numbers, some of them will not be computed, with a potential saving in computer time.

It is easy to "factor out" max from the definition of maximise, giving

```
maximise' (node n nil) = cons n nil
maximise' (node n l) = map minimise l
                     = map (min . minimise') l
                     = map min (map minimise' l)
                     = mapmin (map minimise' l)
where mapmin = map min
```

Since minimise' returns a list of numbers, the minimum of which is the result of minimise, (map minimise' l) returns a list of lists of numbers. Maximise' should return a list of the minima of those lists. However, only the maximum of this list matters. We shall define a new version of mapmin which omits the minima of lists whose minimum doesn't matter.

```
mapmin (cons nums rest) =
        = cons (min nums) (omit (min nums) rest)
```

The function omit is passed a "potential maximum" - the largest minimum seen so far - and omits any minima which are less than this.

```
omit pot nil = nil
omit pot (cons nums rest) =
        = omit pot rest,                          if minleq nums pot
        = cons (min nums) (omit (min nums) rest), otherwise
```

Minleq takes a list of numbers and a potential maximum, and returns true if the minimum of the list of numbers is less than or equal to the potential maximum. To do this, it does not need to look at all the list! If there is any element in the list less than or equal to the potential maximum, then the minimum of the list is sure to be. All elements after this particular one are irrelevant - they are like the question mark in the example above. Therefore minleq can be defined by

```
minleq nil pot = false
minleq (cons num rest) pot = true,           if num<=pot
                           = minleq rest pot, otherwise
```

Having defined maximise' and minimise' in this way it is simple to write a new evaluator:

```
evaluate = max . maximise' . maptree static . prune 8 . gametree
```

Thanks to lazy evaluation, the fact that maximise' looks at less of the tree means that the whole program runs more efficiently, just as the fact that prune looks at only part of an infinite tree enables the program to terminate. The optimisations in maximise', although fairly simple, can have a dramatic effect on the speed of evaluation, and so can allow the evaluator to look further ahead.

Other optimisations can be made to the evaluator. For example, the alpha-beta algorithm just described works best if the best moves are considered first, since if one has found a very good move then there is no need to consider worse moves, other than to demonstrate that the opponent has at least one good reply to them. One might therefore wish to sort the sub-trees at each node, putting those with the highest values first when it is the computer's move, and those with the lowest values first when it is not. This can be done with the function

```
highfirst (node n sub) = node n (sort higher (map lowfirst sub))
lowfirst (node n sub) = node n (sort (not.higher) (map highfirst sub))
higher (node n1 sub1) (node n2 sub2) = n1>n2
```

where sort is a general purpose sorting function. The evaluator would now be defined by

```
evaluate = max . maximise' . highfirst . maptree static .
                prune 8 . gametree
```

One might regard it as sufficient to consider only the three best moves for the computer or the opponent, in order to restrict the search. To program this, it is only necessary to replace highfirst with (taketree 3 . highfirst), where

```
taketree n = redtree (nodett n) cons nil
nodett n label sub = node label (take n sub)
```

Taketree replaces all the nodes in a tree with nodes with at most n subnodes, using the function (take n) which returns the first n elements of a list (or fewer if the list is shorter than n).

Another improvement is to refine the pruning. The program above looks ahead a fixed depth even if the position is very dynamic - it may decide to look no further than a position in which the queen is threated in chess, for example. It is usual to define certain "dynamic" positions and not to allow look-ahead to stop in one of these. Assuming a function "dynamic" that recognises such positions, we need only add one equation to prune to do this:

```
prune 0 (node pos sub) = node pos (map (prune 0) sub),
                                        if dynamic pos
```

Making such changes is easy in a program as modular as this one. As we remarked above, since the program depends crucially for its efficiency on an interaction between maximise, the last function in the chain, and gametree, the first, it can only be written as a monolithic program without lazy evaluation. Such a program is hard to write, hard to modify, and very hard to understand.

6 Conclusion

In this paper, we've argued that modularity is the key to successful programming. Languages which aim to improve productivity must support modular programming well. But new scope rules and mechanisms for separate compilation are not enough - modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue - higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways, and we've shown many examples of this. Smaller and more general modules can be re-used more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones. It also provides a target for functional programmers to aim at. If any part of a program is messy or complicated, the programmer should attempt to modularise it and to generalise the parts. He should expect to use higher-order functions and lazy evaluation as his tools for doing this.

Of course, we are not the first to point out the power and elegance of higher-order functions and lazy evaluation. For example, Turner shows how both can be used to great advantage in a program for generating chemical structures [Tur81]. Abelson and Sussman stress that streams (lazy lists) are a powerful tool for structuring programs [AS86]. Henderson has used streams to structure functional operating systems [P.H82]. The main contribution of this paper is to assert that better modularity alone is the key to the power of functional languages.

It is also relevant to the present controversy over lazy evaluation. Some believe that functional languages should be lazy, others believe they should not. Some compromise and provide only lazy lists, with a special syntax for constructing them (as, for example, in SCHEME [AS86]). This paper provides further evidence that lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess. One should not obstruct access to such a vital tool.

Acknowledgements

This paper owes much to many conversations with Phil Wadler and Richard Bird in the Programming Research Group at Oxford. Magnus Bondesson at Chalmers University, Goteborg pointed out a serious error in an earlier version of one of the numerical algorithms, and thereby prompted development of many of the others. This work was carried out with the support of a Research Fellowship from the UK Science and Engineering Research Council.

References

[AS86] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Boston, 1986.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.

[Hug90] John Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[oD80] United States Department of Defense. *The Programming Language Ada Reference Manual*. Springer-Verlag, 1980.

[P.H82] P.Henderson. Purely Functional Operating Systems. 1982.

[Tur81] D. A. Turner. The Semantic Elegance of Applicative Languages. In *Proceedings 1981 Conference on Functional Languages and Computer Architecture*, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.

[Tur85] D. A. Turner. Miranda: A non-strict language with polymorphic types. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985.

[Wir82] N. Wirth. *Programming in Modula-II*. Springer-Verlag, 1982.

The Roots of Lisp

PAUL GRAHAM

Draft, January 18, 2002.

In 1960, John McCarthy published a remarkable paper in which he did for programming something like what Euclid did for geometry.[1] He showed how, given a handful of simple operators and a notation for functions, you can build a whole programming language. He called this language Lisp, for "List Processing," because one of his key ideas was to use a simple data structure called a *list* for both code and data.

It's worth understanding what McCarthy discovered, not just as a landmark in the history of computers, but as a model for what programming is tending to become in our own time. It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them. As computers have grown more powerful, the new languages being developed have been moving steadily toward the Lisp model. A popular recipe for new programming languages in the past 20 years has been to take the C model of computing and add to it, piecemeal, parts taken from the Lisp model, like runtime typing and garbage collection.

In this article I'm going to try to explain in the simplest possible terms what McCarthy discovered. The point is not just to learn about an interesting theoretical result someone figured out forty years ago, but to show where languages are heading. The unusual thing about Lisp—in fact, the defining quality of Lisp—is that it can be written in itself. To understand what McCarthy meant by this, we're going to retrace his steps, with his mathematical notation translated into running Common Lisp code.

1 Seven Primitive Operators

To start with, we define an *expression*. An expression is either an *atom*, which is a sequence of letters (e.g. `foo`), or a *list* of zero or more expressions, separated by whitespace and enclosed by parentheses. Here are some expressions:

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

The last expression is a list of four elements, the third of which is itself a list of one element.

[1] "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM* 3:4, April 1960, pp. 184–195.

In arithmetic the expression 1 + 1 has the value 2. Valid Lisp expressions also have values. If an expression e yields a value v we say that e *returns* v. Our next step is to define what kinds of expressions there can be, and what value each kind returns.

If an expression is a list, we call the first element the *operator* and the remaining elements the *arguments*. We are going to define seven primitive (in the sense of axioms) operators: `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, and `cond`.

1. `(quote x)` returns x. For readability we will abbreviate `(quote x)` as `'x`.

```
> (quote a)
a
> 'a
a
> (quote (a b c))
(a b c)
```

2. `(atom x)` returns the atom `t` if the value of x is an atom or the empty list. Otherwise it returns `()`. In Lisp we conventionally use the atom `t` to represent truth, and the empty list to represent falsity.

```
> (atom 'a)
t
> (atom '(a b c))
()
> (atom '())
t
```

Now that we have an operator whose argument is evaluated we can show what `quote` is for. By quoting a list we protect it from evaluation. An unquoted list given as an argument to an operator like `atom` is treated as code:

```
> (atom (atom 'a))
t
```

whereas a quoted list is treated as mere list, in this case a list of two elements:

```
> (atom '(atom 'a))
()
```

This corresponds to the way we use quotes in English. Cambridge is a town in Massachusetts that contains about 90,000 people. "Cambridge" is a word that contains nine letters.

Quote may seem a bit of a foreign concept, because few other languages have anything like it. It's closely tied to one of the most distinctive features of Lisp: code and data are made out of the same data structures, and the quote operator is the way we distinguish between them.

3. `(eq x y)` returns `t` if the values of x and y are the same atom or both the empty list, and `()` otherwise.

```
> (eq 'a 'a)
t
> (eq 'a 'b)
()
> (eq '() '())
t
```

4. `(car x)` expects the value of x to be a list, and returns its first element.

```
> (car '(a b c))
a
```

5. `(cdr x)` expects the value of x to be a list, and returns everything after the first element.

```
> (cdr '(a b c))
(b c)
```

6. `(cons x y)` expects the value of y to be a list, and returns a list containing the value of x followed by the elements of the value of y.

```
> (cons 'a '(b c))
(a b c)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (car (cons 'a '(b c)))
a
> (cdr (cons 'a '(b c)))
(b c)
```

7. (`cond` $(p_1\ e_1) \ldots (p_n\ e_n)$) is evaluated as follows. The p expressions are evaluated in order until one returns `t`. When one is found, the value of the corresponding e expression is returned as the value of the whole `cond` expression.

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a)  'second))
second
```

In five of our seven primitive operators, the arguments are always evaluated when an expression beginning with that operator is evaluated.[2] We will call an operator of that type a *function*.

2 Denoting Functions

Next we define a notation for describing functions. A function is expressed as `(lambda (`$p_1 \ldots p_n$`) `e`)`, where $p_1 \ldots p_n$ are atoms (called *parameters*) and e is an expression. An expression whose first element is such an expression

`((lambda (`$p_1 \ldots p_n$`) `e`) `$a_1 \ldots a_n$`)`

is called a *function call* and its value is computed as follows. Each expression a_i is evaluated. Then e is evaluated. During the evaluation of e, the value of any occurrence of one of the p_i is the value of the corresponding a_i in the most recent function call.

```
> ((lambda (x) (cons x '(b))) 'a)
(a b)
> ((lambda (x y) (cons x (cdr y)))
   'z
   '(a b c))
(z b c)
```

If an expression has as its first element an atom f that is not one of the primitive operators

`(`f $a_1 \ldots a_n$`)`

and the value of f is a function `(lambda (`$p_1 \ldots p_n$`) `e`)` then the value of the expression is the value of

`((lambda (`$p_1 \ldots p_n$`) `e`) `$a_1 \ldots a_n$`)`

In other words, parameters can be used as operators in expressions as well as arguments:

```
> ((lambda (f) (f '(b c)))
   '(lambda (x) (cons 'a x)))
(a b c)
```

There is another notation for functions that enables the function to refer to itself, thereby giving us a convenient way to define recursive functions.[3] The

[2] Expressions beginning with the other two operators, `quote` and `cond`, are evaluated differently. When a `quote` expression is evaluated, its argument is not evaluated, but is simply returned as the value of the whole `quote` expression. And in a valid `cond` expression, only an L-shaped path of subexpressions will be evaluated.

[3] Logically we don't need to define a new notation for this. We could define recursive functions in our existing notation using a function on functions called the Y combinator. It may be that McCarthy did not know about the Y combinator when he wrote his paper; in any case, `label` notation is more readable.

notation

`(label` f `(lambda (`$p_1 \ldots p_n$`)` e`))`

denotes a function that behaves like `(lambda (`$p_1 \ldots p_n$`)` e`)`, with the additional property that an occurrence of f within e will evaluate to the `label` expression, as if f were a parameter of the function.

Suppose we want to define a function `(subst` x y z`)`, which takes an expression x, an atom y, and a list z, and returns a list like z but with each instance of y (at any depth of nesting) in z replaced by x.

```
> (subst 'm 'b '(a b (a b c) d))
(a m (a m c) d)
```

We can denote this function as

```
(label subst (lambda (x y z)
               (cond ((atom z)
                      (cond ((eq z y) x)
                            ('t z)))
                     ('t (cons (subst x y (car z))
                               (subst x y (cdr z)))))))
```

We will abbreviate $f =$ `(label` f `(lambda (`$p_1 \ldots p_n$`)` e`))` as

`(defun` f `(`$p_1 \ldots p_n$`)` e`)`

so

```
(defun subst (x y z)
  (cond ((atom z)
         (cond ((eq z y) x)
               ('t z)))
        ('t (cons (subst x y (car z))
                  (subst x y (cdr z)))))))
```

Incidentally, we see here how to get a default clause in a `cond` expression. A clause whose first element is `'t` will always succeed. So

`(cond (`x y`) ('t` z`))`

is equivalent to what we might write in a language with syntax as

`if` x `then` y `else` z

3 Some Functions

Now that we have a way of expressing functions, we define some new ones in terms of our seven primitive operators. First it will be convenient to introduce

some abbreviations for common patterns. We will use cxr, where x is a sequence of as or ds, as an abbreviation for the corresponding composition of car and cdr. So for example (cadr e) is an abbreviation for (car (cdr e)), which returns the second element of e.

```
> (cadr '((a b) (c d) e))
(c d)
> (caddr '((a b) (c d) e))
e
> (cdar '((a b) (c d) e))
(b)
```

Also, we will use (list $e_1 \ldots e_n$) for (cons e_1 ... (cons e_n '()) ...).

```
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (list 'a 'b 'c)
(a b c)
```

Now we define some new functions. I've changed the names of these functions by adding periods at the end. This distinguishes primitive functions from those defined in terms of them, and also avoids clashes with existing Common Lisp functions.

1. (null. x) tests whether its argument is the empty list.

   ```
   (defun null. (x)
     (eq x '()))

   > (null. 'a)
   ()
   > (null. '())
   t
   ```

2. (and. x y) returns t if both its arguments do and () otherwise.

   ```
   (defun and. (x y)
     (cond (x (cond (y 't) ('t '())))
           ('t '())))

   > (and. (atom 'a) (eq 'a 'a))
   t
   > (and. (atom 'a) (eq 'a 'b))
   ()
   ```

3. (not. x) returns t if its argument returns (), and () if its argument returns t.

```
(defun not. (x)
  (cond (x '())
        ('t 't)))

> (not (eq 'a 'a))
()
> (not (eq 'a 'b))
t
```

4. `(append.` x y`)` takes two lists and returns their concatenation.

```
(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

> (append. '(a b) '(c d))
(a b c d)
> (append. '() '(c d))
(c d)
```

5. `(pair.` x y`)` takes two lists of the same length and returns a list of two-element lists containing successive pairs of an element from each.

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list (car x) (car y))
               (pair. (cdr x) (cdr y))))))

> (pair. '(x y z) '(a b c))
((x a) (y b) (z c))
```

6. `(assoc.` x y`)` takes an atom x and a list y of the form created by `pair.`, and returns the second element of the first list in y whose first element is x.

```
(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))

> (assoc. 'x '((x a) (y b)))
a
> (assoc. 'x '((x new) (x a) (y b)))
new
```

4 The Surprise

So we can define functions that concatenate lists, substitute one expression for another, etc. An elegant notation, perhaps, but so what? Now comes the surprise. We can also, it turns out, write a function that acts as an interpreter for our language: a function that takes as an argument any Lisp expression, and returns its value. Here it is:

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```

The definition of `eval.` is longer than any of the others we've seen before. Let's consider how each part works.

The function takes two arguments: `e`, the expression to be evaluated, and `a`, a list representing the values that atoms have been given by appearing as

parameters in function calls. This list is called the *environment*, and it is of the form created by `pair.`. It was in order to build and search these lists that we wrote `pair.` and `assoc.`.

The spine of `eval.` is a `cond` expression with four clauses. How we evaluate an expression depends on what kind it is. The first clause handles atoms. If `e` is an atom, we look up its value in the environment:

```
> (eval. 'x '((x a) (y b)))
a
```

The second clause of `eval.` is another `cond` for handling expressions of the form $(a\ \ldots)$, where a is an atom. These include all the uses of the primitive operators, and there is a clause for each one.

```
> (eval. '(eq 'a 'a) '())
t
> (eval. '(cons x '(b c))
         '((x a) (y b)))
(a b c)
```

All of these (except `quote`) call `eval.` to find the value of the arguments.

The last two clauses are more complicated. To evaluate a `cond` expression we call a subsidiary function called `evcon.`, which works its way through the clauses recursively, looking for one in which the first element returns `t`. When it finds such a clause it returns the value of the second element.

```
> (eval. '(cond ((atom x) 'atom)
                ('t 'list))
         '((x '(a b))))
list
```

The final part of the second clause of `eval.` handles calls to functions that have been passed as parameters. It works by replacing the atom with its value (which ought to be a `lambda` or `label` expression) and evaluating the resulting expression. So

```
(eval. '(f '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

turns into

```
(eval. '((lambda (x) (cons 'a x)) '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

which returns `(a b c)`.

The last two clauses in `eval.` handle function calls in which the first element is an actual `lambda` or `label` expression. A `label` expression is evaluated by pushing a list of the function name and the function itself onto the environment, and then calling `eval.` on an expression with the inner `lambda` expression substituted for the `label` expression. That is,

```
(eval. '((label firstatom (lambda (x)
                            (cond ((atom x) x)
                                  ('t (firstatom (car x))))))
         y)
       '((y ((a b) (c d)))))
```

becomes

```
(eval. '((lambda (x)
           (cond ((atom x) x)
                 ('t (firstatom (car x)))))
         y)
       '((firstatom
          (label firstatom (lambda (x)
                             (cond ((atom x) x)
                                   ('t (firstatom (car x)))))))
         (y ((a b) (c d)))))
```

which eventually returns `a`.

Finally, an expression of the form `((lambda (`$p_1 \ldots p_n$`) `e`)` $a_1 \ldots a_n$`)` is evaluated by first calling `evlis.` to get a list of values `(`v_1 `...` v_n`)` of the arguments $a_1 \ldots a_n$, and then evaluating e with `(`p_1 v_1`)` $\ldots$ `(`p_n v_n`)` appended to the front of the environment. So

```
(eval. '((lambda (x y) (cons x (cdr y)))
         'a
         '(b c d))
       '())
```

becomes

```
(eval. '(cons x (cdr y))
       '((x a) (y (b c d))))
```

which eventually returns `(a c d)`.

5 Aftermath

Now that we understand how `eval` works, let's step back and consider what it means. What we have here is a remarkably elegant model of computation. Using just `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, and `cond`, we can define a function, `eval.`, that actually implements our language, and then using that we can define any additional function we want.

There were already models of computation, of course—most notably the Turing Machine. But Turing Machine programs are not very edifying to read. If you want a language for describing algorithms, you might want something more abstract, and that was one of McCarthy's aims in defining Lisp.

The language he defined in 1960 was missing a lot. It has no side-effects, no sequential execution (which is useful only with side effects anyway), no practical numbers,[4] and dynamic scope. But these limitations can be remedied with surprisingly little additional code. Steele and Sussman show how to do it in a famous paper called "The Art of the Interpreter."[5]

If you understand McCarthy's `eval`, you understand more than just a stage in the history of languages. These ideas are still the semantic core of Lisp today. So studying McCarthy's original paper shows us, in a sense, what Lisp really is. It's not something that McCarthy designed so much as something he discovered. It's not intrinsically a language for AI or for rapid prototyping, or any other task at that level. It's what you get (or one thing you get) when you try to axiomatize computation.

Over time, the median language, meaning the language used by the median programmer, has grown consistently closer to Lisp. So by understanding `eval` you're understanding what will probably be the main model of computation well into the future.

[4] It is possible to do arithmetic in McCarthy's 1960 Lisp by using e.g. a list of n atoms to represent the number n.

[5] Guy Lewis Steele, Jr. and Gerald Jay Sussman, "The Art of the Interpreter, or the Modularity Complex (Parts Zero, One, and Two)," MIT AI Lab Memo 453, May 1978.

Notes

In translating McCarthy's notation into running code I tried to change as little as possible. I was tempted to make the code easier to read, but I wanted to keep the flavor of the original.

In McCarthy's paper, falsity is represented by `f`, not the empty list. I used `()` to represent falsity so that the examples would work in Common Lisp. The code nowhere depends on falsity happening also to be the empty list; nothing is ever consed onto the result returned by a predicate.

I skipped building lists out of dotted pairs, because you don't need them to understand `eval`. I also skipped mentioning `apply`, though it was `apply` (a very early form of it, whose main purpose was to quote arguments) that McCarthy called the universal function in 1960; `eval` was then just a subroutine that `apply` called to do all the work.

I defined `list` and the `c`*x*`r`s as abbreviations because that's how McCarthy did it. In fact the `c`*x*`r`s could all have been defined as ordinary functions. So could `list` if we modified `eval`, as we easily could, to let functions take any number of arguments.

McCarthy's paper only had five primitive operators. He used `cond` and `quote` but may have thought of them as part of his metalanguage. He likewise didn't define the logical operators `and` and `not`, but this is less of a problem because adequate versions can be defined as functions.

In the definition of `eval.` we called other functions like `pair.` and `assoc.`, but any call to one of the functions we defined in terms of the primitive operators could be replaced by a call to `eval.`. That is,

```
(assoc. (car e) a)
```

could have been written as

```
(eval. '((label assoc.
               (lambda (x y)
                 (cond ((eq (caar y) x) (cadar y))
                       ('t (assoc. x (cdr y))))))
         (car e)
         a)
        (cons (list 'e e) (cons (list 'a a) a)))
```

There was a small bug in McCarthy's `eval`. Line 16 was (equivalent to) `(evlis. (cdr e) a)` instead of just `(cdr e)`, which caused the arguments in a call to a named function to be evaluated twice. This suggests that this description of `eval` had not yet been implemented in IBM 704 machine language when the paper was submitted. It also shows how hard it is to be sure of the correctness of any length of program without trying to run it.

I encountered one other problem in McCarthy's code. After giving the definition of `eval` he goes on to give some examples of higher-order functions—functions that take other functions as arguments. He defines `maplist`:

```
(label maplist
       (lambda (x f)
         (cond ((null x) '())
               ('t (cons (f x) (maplist (cdr x) f))))))
```

then uses it to write a simple function `diff` for symbolic differentiation. But `diff` passes `maplist` a function that uses `x` as a parameter, and the reference to it is captured by the parameter `x` within `maplist`.[6]

It's an eloquent testimony to the dangers of dynamic scope that even the very first example of higher-order Lisp functions was broken because of it. It may be that McCarthy was not fully aware of the implications of dynamic scope in 1960. Dynamic scope remained in Lisp implementations for a surprisingly long time—until Sussman and Steele developed Scheme in 1975. Lexical scope does not complicate the definition of `eval` very much, but it may make compilers harder to write.

[6]Present day Lisp programmers would use `mapcar` instead of `maplist` here. This example does clear up one mystery: why `maplist` is in Common Lisp at all. It was the original mapping function, and `mapcar` a later addition.