PL: Lecture #2

Tuesday, September 10th

Intro to Racket

- General layout of the parts of Racket:
  - The Racket language is (mostly) in the Scheme family, or more generally in the Lisp family;
  - Racket: the core language implementation (language and runtime), written in C;
  - The actual language(s) that are available in Racket have lots of additional parts that are implemented in Racket itself;
  - GRacket: a portable Racket GUI extension, written in Racket too;
  - DrRacket: a GRacket application (also written in Racket);
  - Our language(s)…
- Documentation: the Racket documentation is your friend (But beware that some things are provided in different forms from different places).

Side-note: “Goto Statement Considered Harmful”

> *A review of “Goto Statement Considered Harmful”, by E.W. DIJKSTRA*
>
> *This paper tries to convince us that the well-known goto statement should be eliminated from our programming languages or, at least (since I don’t think that it will ever be eliminated), that programmers should not use it. It is not clear what should replace it. The paper doesn’t explain to us what would be the use of the `if` statement without a `goto` to redirect the flow of execution: Should all our postconditions consist of a single statement, or should we only use the arithmetic `if`, which doesn’t contain the offensive `goto`?*
>
> *And how will one deal with the case in which, having reached the end of an alternative, the program needs to continue the execution somewhere else?*
>
> *The author is a proponent of the so-called “structured programming” style, in which, if I get it right, gotos are replaced by indentation. Structured programming is a nice academic exercise, which works well for small examples, but I doubt that any real-world program will ever be written in such a style. More than 10 years of industrial experience with Fortran have proved conclusively to everybody concerned that, in the real world, the goto is useful and necessary: its presence might cause some inconveniences in debugging, but it is a de facto standard and we must live with it. It will take more than the academic elucubrations of a purist to remove it from our languages.*
>
> *Publishing this would waste valuable paper: Should it be published, I am as sure it will go uncited and unnoticed as I am confident that, 30 years from now, the goto will still be alive and well and used as widely as it is today.*
>
> *Confidential comments to the editor: The author should withdraw the paper and submit it someplace where it will not be peer reviewed. A letter to the editor would be a perfect choice: Nobody will notice it there!*

Quick Intro to Racket

Racket syntax: similar to other Sexpr-based languages.

Reminder: the parens can be compared to C/etc function call parens — they always mean that some function is applied. This is the reason why `(+ (1) (2))` won’t work: if you use C syntax that is `+(1(), 2())` but `1` isn’t a function so `1()` is an error.

> *An important difference between syntax and semantics: A good way to think about this is the difference between the string `42` stored in a file somewhere (two ASCII values), and the number `42` stored in memory (in some*

> *representation). You could also continue with the above example: there is nothing wrong with “murder” — it’s just a word, but murder is something you’ll go to jail for.*

The evaluation function that Racket uses is actually a function that takes a piece of syntax and returns (or executes) its semantics.

`define` expressions are used for creating new bindings, do not try to use them to change values. For example, you should not try to write something like `(define x (+ x 1))` in an attempt to mimic `x = x+1`. It will not work.

There are two boolean values built in to Racket: `#t` (true) and `#f` (false). They can be used in `if` statements, for example:

```
(if (< 2 3) 10 20)  -->  10
```

because `(< 2 3)` evaluates to `#t`. As a matter of fact, *any* value except for `#f` is considered to be true, so:

```
(if 0 1 2)         -->  1  ; all of these are "truthy"
(if "false" 1 2)  -->  1
(if "" 1 2)        -->  1
(if null 1 2)      -->  1
(if #t 1 2)        -->  1  ; including the true value
(if #f 1 2)        -->  2  ; the only false value
(if #false 1 2)    -->  2  ; another way to write it
(if false 1 2)     -->  2  ; also false since it's bound to #f
```

Note: Racket is a *functional* language — so *everything* has a value.

This means that the expression

```
(if test consequent)
```

has no meaning when `test` evaluates to `#f`. This is unlike Pascal/C where statements *do* something (side effect) like printing or an assignment — here an `if` statement with no alternate part will just *do nothing* if the test is false… Racket, however, must return some value — it could decide on simply returning `#f` (or some unspecified value) as the value of

```
(if #f something)
```

as some implementations do, but Racket just declares it a syntax error. (As we will see in the future, Racket has a more convenient `when` with a clearer intention.)

Well, *almost* everything is a value…

There are certain things that are part of Racket’s syntax — for example `if` and `define` are special forms, they do not have a value! More about this shortly.

(Bottom line: much more things do have a value, compared with other languages.)

`cond` is used for a `if` … `else if` … `else if` … `else` … sequence. The problem is that nested `if`s are inconvenient. For example,

```
(define (digit-num n)
  (if (<= n 9)
    1
    (if (<= n 99)
      2
      (if (<= n 999)
        3
        (if (<= n 9999)
          4
          "a lot")))))
```

In C/Java/Whatever, you'd write:

```
function digit_num(n) {
  if (n <= 9)         return 1;
  else if (n <= 99)   return 2;
  else if (n <= 999)  return 3;
  else if (n <= 9999) return 4;
  else return "a lot";
}
```

(Side question: why isn't there a `return` statement in Racket?)

But trying to force Racket code to look similar:

```
(define (digit-num n)
  (if (<= n 9)
    1
  (if (<= n 99)
    2
  (if (<= n 999)
    3
  (if (<= n 9999)
    4
    "a lot")))))
```

is more than just bad taste — the indentation rules are there for a reason, the main one is that you can see the structure of your program at a quick glance, and this is no longer true in the above code. (Such code will be penalized!)

So, instead of this, we can use Racket's `cond` statement, like this:

```
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [else        "a lot"]))
```

Note that `else` is a keyword that is used by the `cond` form — you should always use an `else` clause (for similar reasons as an `if`, to avoid an extra expression evaluation there, and we will need it when we use a typed language). Also note that square brackets are read by DrRacket like round parens, it will only make sure that the paren pairs match. We use this to make code more readable — specifically, there is a major difference between the above use of `[]` from the conventional use of `()`. Can you see what it is?

The general structure of a `cond`:

```
(cond [test-1 expr-1]
      [test-2 expr-2]
      ...
      [test-n expr-n]
      [else   else-expr])
```

Example for using an `if` expression, and a recursive function:

```
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

Use this to show the different tools, especially:

- special objects that *cannot* be used
- syntax-checker
- stepper
- submission tool (installing, registering and submitting)

An example of converting it to tail recursive form:

```
(define (helper n acc)
  (if (zero? n)
    acc
    (helper (- n 1) (* acc n))))

(define (fact n)
  (helper n 1))
```

Additional notes about homework submissions:

- Begin every function with clear documentation: a type followed by a purpose statement.
- Document the function when needed, and according to the guidelines above and in the style guide.
- After the function, always have a few test cases — they should cover your complete code (make sure to include possible corner cases). Later on, we will switch to testing the whole file through it's "public interface", instead of testing each function.

Lists & Recursion

Lists are a fundamental Racket data type.

A list is defined as either:

1. the empty list (`null`, `empty`, or `'()`),
2. a pair (`cons` cell) of anything and a list.

As simple as this may seem, it gives us precise *formal* rules to prove that something is a list.

- Why is there a "the" in the first rule?

Examples:

```
null
(cons 1 null)
(cons 1 (cons 2 (cons 3 null)))
(list 1 2 3) ; a more convenient function to get the above
```

List operations — predicates:

```
null? ; true only for the empty list
pair? ; true for any cons cell
list? ; this can be defined using the above
```

We can derive `list?` from the above rules:

```
(define (list? x)
  (if (null? x)
    #t
    (and (pair? x) (list? (rest x)))))
```

or better:

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (rest x)))))
```

But why can't we define `list?` more simply as

```
(define (list? x)
  (or (null? x) (pair? x)))
```

The difference between the above definition and the proper one can be observed in the full Racket language, not in the student languages (where there are no pairs with non-list values in their tails).

List operations — destructors for pairs (`cons` cells):

```
first
rest
```

Traditionally called `car`, `cdr`.

Also, any `c<x>r` combination for `<x>` that is made of up to four `a`s and/or `d`s — we will probably not use much more than `cadr`, `caddr` etc.

Example for recursive function involving lists:

```
(define (list-length list)
  (if (null? list)
    0
    (+ 1 (list-length (rest list)))))
```

Use different tools, esp:

- syntax-checker
- stepper

How come we could use `list` as an argument — use the syntax checker

```
(define (list-length-helper list len)
  (if (null? list)
    len
    (list-length-helper (rest list) (+ len 1))))

(define (list-length list)
  (list-length-helper list 0))
```

Main idea: lists are a recursive structure, so functions that operate on lists should be recursive functions that follow the recursive definition of lists.

Another example for list function — summing a list of numbers

```
(define (sum-list l)
  (if (null? l)
    0
    (+ (first l) (sum-list (rest l)))))
```

Also show how to implement `rcons`, using this guideline.

More examples:

Define `reverse` — solve the problem using `rcons`.

`rcons` can be generalized into something very useful: `append`.

- How would we use `append` instead of `rcons`?
- How much time will this take? Does it matter if we use `append` or `rcons`?

Redefine `reverse` using tail recursion.

- Is the result more complex? (Yes, but not too bad because it collects the elements in reverse.)

Some Style

When you have some common value that you need to use in several places, it is bad to duplicate it. For example:

```
(define (how-many a b c)
  (cond [(> (* b b) (* 4 a c)) 2]
        [(= (* b b) (* 4 a c)) 1]
        [(< (* b b) (* 4 a c)) 0]))
```

What’s bad about it?

- It’s longer than necessary, which will eventually make your code less readable.
- It’s slower — by the time you reach the last case, you have evaluated the two sequences three times.
- It’s more prone to bugs — the above code is short enough, but what if it was longer so you don’t see the three occurrences on the same page? Will you remember to fix all places when you debug the code months after it was written?

In general, the ability to use names is probably the most fundamental concept in computer science — the fact that makes computer programs what they are.

We already have a facility to name values: function arguments. We could split the above function into two like this:

```
(define (how-many-helper b^2 4ac) ; note: valid names!
  (cond [(> b^2 4ac) 2]
        [(= b^2 4ac) 1]
        [else        0]))

(define (how-many a b c)
  (how-many-helper (* b b) (* 4 a c)))
```

But instead of the awkward solution of coming up with a new function just for its names, we have a facility to bind local names — `let`. In general, the syntax for a `let` special form is

```
(let ([id expr] ...) expr)
```

For example,

```
(let ([x 1] [y 2]) (+ x y))
```

But note that the bindings are done “in parallel”, for example, try this:

```
(let ([x 1] [y 2])
  (let ([x y] [y x])
    (list x y)))
```

(Note that “in parallel” is quoted here because it’s not really parallelism, but just a matter of scopes: the RHSs are all evaluated in the surrounding scope!)

Using this for the above problem:

```
(define (how-many a b c)
  (let ([b^2 (* b b)]
        [4ac (* 4 a c)])
    (cond [(> b^2 4ac) 2]
          [(= b^2 4ac) 1]
          [else        0])))
```

- Some notes on writing code (also see the style-guide in the handouts section)
- ***Code quality will be graded to in this course!***
- Use abstractions whenever possible, as said above. This is bad:

  ```
  (define (how-many a b c)
    (cond
      [(> (* b b) (* 4 a c)) 2]
      [(= (* b b) (* 4 a c)) 1]
      [(< (* b b) (* 4 a c)) 0]))

  (define (what-kind a b c)
    (cond
      [(= a 0) 'degenerate]
      [(> (* b b) (* 4 a c)) 'two]
      [(= (* b b) (* 4 a c)) 'one]
      [(< (* b b) (* 4 a c)) 'none]))
  ```

- But don't over abstract: `(define one 1)` or `(define two "two")`
- Always do test cases, you might want to comment them, but you should always make sure your code works. Use DrRacket's covergae features to ensure complete coverage.
- Do not under-document, but also don't over-document.
- ***INDENTATION!*** (Let DrRacket decide; get used to its rules) –> This is part of the culture that was mentioned last time, but it's done this way for good reason: decades of programming experience have shown this to be the most readable format. It's also extremely important to keep good indentation since programmers in all Lisps don't count parens — they look at the structure.
- As a general rule, `if` should be either all on one line, or the condition on the first and each consequent on a separate line. Similarly for `define` — either all on one line or a newline after the object that is being define (either an identifier or a an identifier with arguments).
- Another general rule: you should never have white space after an open-paren, or before a close paren (white space includes newlines). Also, before an open paren there should be either another open paren or white space, and the same goes for after a closing paren.
- Use the tools that are available to you: for example, use `cond` instead of nested `if`s (definitely do not force the indentation to make a nested `if` look like its C counterpart — remember to let DrRacket indent for you).

  Another example — do not use `(+ 1 (+ 2 3))` instead of `(+ 1 2 3)` (this might be needed in *extremely* rare situations, only when you know your calculus and have extensive knowledge about round-off errors).

  Another example — do not use `(cons 1 (cons 2 (cons 3 null)))` instead of `(list 1 2 3)`.

  Also — don't write things like:

  ```
  (if (< x 100) #t #f)
  ```

  since it's the same as just

  ```
  (< x 100)
  ```

  A few more of these:

  ```
  (if x #t y)   --same-as-->  (or x y)             ; (almost)
  (if x y #f)   --same-as-->  (and x y)            ; (exacly same)
  (if x #f #t)  --same-as-->  (not x)              ; (almost)
  ```

(Actually the first two are almost the same, for example, `(and 1 2)` will return `2`, not `#t`.)

- Use these as examples for many of these issues:

```
(define (interest x)
  (* x (cond
    [(and (> x 0) (<= x 1000)) 0.04]
    [(and (> x 1000) (<= x 5000)) 0.045]
    [else 0.05])))

(define (how-many a b c)
  (cond ((> (* b b) (* (* 4 a) c))
         2)
        ((< (* b b) (* (* 4 a) c))
         0)
        (else
         1)))

(define (what-kind a b c)
  (if (equal? a 0) 'degenerate
      (if (equal? (how-many a b c) 0) 'zero
          (if (equal? (how-many a b c) 1) 'one
              'two)
          )
      )
  )

(define (interest deposit)
  (cond
  [(< deposit 0) "invalid deposit"]
  [(and (>= deposit 0) (<= deposit 1000)) (* deposit 1.04) ]
  [(and (> deposit 1000) (<= deposit 5000)) (* deposit 1.045)]
  [(> deposit 5000) (* deposit 1.05)]))

(define (interest deposit)
  (if (< deposit 1001) (* 0.04 deposit)
    (if (< deposit 5001) (* 0.045 deposit)
    (* 0.05 deposit))))

(define (what-kind a b c) (cond ((= 0 a) 'degenerate)
                          (else (cond ((> (* b b)(*(* 4 a) c)) 'two)
                               (else (cond ((= (* b b)(*(* 4 a) c)) 'one)
                               (else 'none)))))));
```

Tail calls

You should generally know what tail calls are, but here's a quick review of the subject. A function call is said to be in tail position if there is no context to "remember" when you're calling it. Very roughly, this means that function calls that are not nested as argument expressions of another *call* are tail calls. Pay attention that we're talking about *function calls*, not, for example, being nested in an `if` expression since that's not a function. (The same holds for `cond`, `and`, `or`.)

This definition is something that depends on the context, for example, in an expression like

```
(if (huh?)
  (foo (add1 (* x 3)))
  (foo (/ x 2)))
```

both calls to `foo` are tail calls, but they're tail calls of *this* expression and therefore apply to *this* context. It might be that this code is inside another call, as in

```
(blah (if (huh?)
        (foo (add1 (* x 3)))
        (foo (/ x 2)))
      something-else)
```

and the `foo` calls are now *not* in tail position. The main feature of all Scheme implementations including Racket (and including Javascript) WRT tail calls is that calls that are in tail position of a function are said to be "eliminated". That means that if we're in an `f` function, and we're about to call `g` in tail position and therefore whatever `g` returns would be the result of `f` too, then when Racket does the call to `g` it doesn't bother keeping the `f` context — it won't remember that it needs to "return" to `f` and will instead return straight to its caller. In other words, when you think about a conventional implementation of function calls as frames on a stack, Racket will get rid of a stack frame when it can.

You can also try this with any code in DrRacket: hovering over the paren that starts a function call will show a faint pinkish arrow showing the tail-call chain from there for call that are actually tail calls. This is a simple feature since tail calls are easily identifiable by just looking at the syntax of a function.

Another way to see this is to use DrRacket's stepper to step through a function call. The stepper is generally an alternative debugger, where instead of visualizing stack frames it assembles an expression that represents these frames. Now, in the case of tail calls, there is no room in such a representation to keep the call — and the thing is that in Racket that's perfectly fine since these calls are not kept on the call stack.

Note that there are several names for this feature:

- "Tail recursion". This is a common way to refer to the more limited optimization of *only* tail-recursive functions into loops. In languages that have tail calls as a feature, this is too limited, since they also optimize cases of mutual recursion, or any case of a tail call.
- "Tail call optimization". In some languages, or more specifically in some compilers, you'll hear this term. This is fine when tail calls are considered only an "optimization" — but in Racket's case (as well as Scheme), it's more than just an optimization: it's a *language feature* that you can rely on. For example, a tail-recursive function like `(define (loop) (loop))` *must* run as an infinite loop, not just optimized to one when the compiler feels like it.
- "Tail call elimination". This is the so far the most common proper name for the feature: it's not just recursion, and it's not an optimization.

When should you use tail calls?

Often, people who are aware of tail calls will try to use them *always*. That's not always a good idea. You should generally be aware of the tradeoffs when you consider what style to use. The main thing to remember is that tail-call elimination is a property that helps reducing *space* use (stack space) — often reducing it from linear space to constant space. This can obviously make things faster, but usually the speedup is just a constant factor since you need to do the same number of iterations anyway, so you just reduce the time spent on space allocation.

Here is one such example that we've seen:

```
(define (list-length-1 list)
  (if (null? list)
    0
    (+ 1 (list-length-1 (rest list)))))

;; versus

(define (list-length-helper list len)
  (if (null? list)
    len
    (list-length-helper (rest list) (+ len 1))))
```

```
(define (list-length-2 list)
  (list-length-helper list 0))
```

In this case the first (recursive) version version consumes space linear to the length of the list, whereas the second version needs only constant space. But if you consider only the asymptotic runtime, they are both *O(length(* `l` *))*.

A second example is a simple implementation of `map`:

```
(define (map-1 f l)
  (if (null? l) l (cons (f (first l)) (map-1 f (rest l)))))

;; versus

(define (map-helper f l acc)
  (if (null? l)
    (reverse acc)
    (map-helper f (rest l) (cons (f (first l)) acc))))
(define (map-2 f l)
  (map-helper f l '()))
```

In this case, both the asymptotic space and the runtime consumption are the same. In the recursive case we have a constant factor for the stack space, and in the iterative one (the tail-call version) we also have a similar factor for accumulating the reversed list. In this case, it is probably better to keep the first version since the code is simpler. In fact, Racket's stack space management can make the first version run faster than the second — so optimizing it into the second version is useless.

Sidenote on Types

> *Note: this is all just a side note for a particularly hairy example. You don't need to follow all of this to write code in this class! Consider this section a kind of an extra type-related puzzle to read trough, and maybe get back to it much later, after we cover typechecking.*

Types can become interestingly complicated when dealing with higher-order functions. Specifically, the nature of the type system used by Typed Racket makes it have one important weakness: it often fails to infer types when there are higher-order functions that operate on polymorphic functions.

For example, consider how `map` receives a function and a list of some type, and applies the function over this list to accumulate its output, so it's a polymorphic function with the following type:

```
map : (A -> B) (Listof A) -> (Listof B)
```

But Racket's `map` is actually more flexible that that: it can take more than a single list input, in which case it will apply the function on the first element in all lists, then the second and so on. Narrowing our vision to the two-input-lists case, the type of `map` then becomes:

```
map : (A B -> C) (Listof A) (Listof B) -> (Listof C)
```

Now, here's a hairy example — what is the type of this function:

```
(define (foo x y)
  (map map x y))
```

Begin by what we know — both `map`s, call them `map1` and `map2`, have the double- and single-list types of `map` respectively, here they are, with different names for types:

```
;; the first `map', consumes a function and two lists
map1 : (A B -> C) (Listof A) (Listof B) -> (Listof C)
```

```
;; the second `map', consumes a function and one list
map2 : (X -> Y) (Listof X) -> (Listof Y)
```

Now, we know that `map2` is the first argument to `map1`, so the type of `map1`s first argument should be the type of `map2`:

```
(A B -> C) = (X -> Y) (Listof X) -> (Listof Y)
```

From here we can conclude that

```
A = (X -> Y)

B = (Listof X)

C = (Listof Y)
```

If we use these equations in `map1`’s type, we get:

```
map1 : ((X -> Y) (Listof X) -> (Listof Y))
       (Listof (X -> Y))
       (Listof (Listof X))
       -> (Listof (Listof Y))
```

Now, `foo`’s two arguments are the 2nd and 3rd arguments of `map1`, and its result is `map1`s result, so we can now write our “estimated” type of `foo`:

```
(: foo : (Listof (X -> Y))
         (Listof (Listof X))
         -> (Listof (Listof Y)))
(define (foo x y)
  (map map x y))
```

This should help you understand why, for example, this will cause a type error:

```
(foo (list add1 sub1 add1) (list 1 2 3))
```

and why this is valid:

```
(foo (list add1 sub1 add1) (map list (list 1 2 3)))
```

But…!

There’s a big “but” here which is that weakness of Typed Racket that was mentioned. If you try to actually write such a defninition in `#lang pl` (which is based on Typed Racket), you will first find that you need to explicitly list the type variable that are needed to make it into a generic type. So the above becomes:

```
(: foo : (All (X Y)
           (Listof (X -> Y))
           (Listof (Listof X))
           -> (Listof (Listof Y))))
(define (foo x y)
  (map map x y))
```

But not only does that not work — it throws an obscure type error. That error is actually due to TR’s weakness: it’s a result of not being able to infer the proper types. In such cases, TR has two mechanisms to “guide it” in the right direction. The first one is `inst`, which is used to instantiate a generic (= polymorphic) type some actual type. The problem here is with the second `map` since that’s the polymorphic function that is given to a higher-order function (the first `map`). If we provide the types to instantiate this, it will work fine:

```
(: foo : (All (X Y)
           (Listof (X -> Y))
```

```
            (Listof (Listof X))
            -> (Listof (Listof Y))))
(define (foo x y)
  (map (inst map Y X) x y))
```

Now, you can use this definition to run the above example:

```
(foo (list add1 sub1 add1) (list (list 1) (list 2) (list 3)))
```

This example works fine, but that's because we wrote the list argument explicitly. If you try to use the exact example above,

```
(foo (list add1 sub1 add1) (map list (list 1 2 3)))
```

you'd run into the same problem again, since this also uses a polymorphic function (`list`) with a higher-order one (`map`). Indeed, an `inst` can make this work for this too:

```
(foo (list add1 sub1 add1) (map (inst list Number) (list 1 2 3)))
```

The second facility is `ann`, which can be used to annotate an expression with the type that you expect it to have.

```
(define (foo x y)
  (map (ann map ((X -> Y) (Listof X) -> (Listof Y)))
       x y))
```

(Note: this is not type casting! It's using a different type which is also applicable for the given expression, and having the type checker validate that this is true. TR does have a similar `cast` form, which is used for a related but different cases.)

This tends to be more verbose than `inst`, but is sometimes easier to follow, since the expected type is given explicitly. The thing about `inst` is that it's kind of "applying" a polymorphic `(All (A B) ...)` type, so you need to know the order of the `A B` arguments, which is why in the above we use `(inst map Y X)` rather than `(inst map X Y)`.

> *Again, remember that this is all not something that you need to know. We will have a few (very rare) cases where we'll need to use `inst`, and in each of these, you'll be told where and how to use it.*

Side-note: Names are important

An important "discovery" in computer science is that we *don't* need names for every intermediate sub-expression — for example, in almost any language we can write something like:

```
s = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

instead of

```
x₁ = b * b
y₁ = 4 * a
y₂ = y * c
x₂ = x - y
x₃ = sqrt(x)
y₃ = -b
x₄ = y + x
y₄ = 2 * a
s  = x / y
```

Such languages are put in contrast to assembly languages, and were all put under the generic label of "high level languages".

(Here’s an interesting idea — why not do the same for function values?)