CSE341, Fall 2011, Lecture 17 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture covers three topics that are all directly relevant to the upcoming homework assignment in which you will use Racket to write an interpreter for a small programming language:

- Racket's *structs* for defining new (dynamic) types in Racket and contrasting them with manually tagged lists *[This material was actually covered in the "Monday section" and only briefly reviewed in the other materials for this lecture.]*
- How a programming language can be implemented in another programming language, including key stages/concepts like parsing, compilation, interpretation, and variants thereof
- How to implement higher-order functions and closures by explicitly managing and storing environments at run-time

One-of types with lists and dynamic typing

In ML, we studied one-of types (in the form of `datatype` bindings) almost from the beginning, since they were essential for defining many kinds of data. Racket's dynamic typing makes the issue less central since we can simply use primitives like booleans and cons cells to build *anything* we want. Building lists and trees out of dynamically typed pairs is particularly straightforward.

However, the *concept* of one-of types is still prevalent. For example, a list is `null` *or* a cons cell where there cdr holds a list. Moreover, Racket extends its Scheme predecessor with something very similar to ML's constructors — a special form called `struct` — and it is instructive to contrast the two language's approaches.

Before seeing `struct`, let's consider Racket without it. There is no type system to restrict what we pass to a function, put in a pair, etc. However, there are different kinds of values — various kinds of numbers, procedures, pairs, strings, symbols, booleans, the empty-list — and built-in *predicates* like `number?` and `null?` to determine what kind of value some particular value is. For values that have "parts" (cons cells), there are built-in functions for getting the parts (`car` and `cdr`). So in a real sense, *all* Racket values are in "one big datatype" and primitives like `+` check the "tags" of their arguments (e.g., is it a number using `number?`) and raise an exception for a "run-time type error" before performing some computation (e.g., addition) and making a new "tagged" value (e.g., a new number). These "tags" are like ML constructors, but it is like our whole language has *exactly one datatype, everything is in it, all tags are implicit and automatically inserted, and the primitives implicitly include case expressions that raise exceptions for the wrong tags.*

Given struct-less Racket, we can still "code up" our own datatypes as an *idiom*. One reasonable approach is to use lists where the first list element is a symbol[1] to indicate which variant of a one-of type you have. For example, consider this ML datatype:

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

One way to represent similar values in Racket is to use lists where the first element is `'Const`, `'Add`, or `'Negate`. If it is `'Const`, we will have one more list element which is a number. If it is `'Add`, we will

[1]See The Racket Guide for a discussion of what symbols are. In short, they are constants that you can compare using `eq?`, which is fast. They are not strings despite the obvious similarity. A symbol is an atom, meaning it has no subparts: you cannot extract a character of a symbol. There are primitives to convert between symbols and strings, but there are also primitives to convert between numbers and strings, which are clearly not the same thing.

have two more list elements holding subexpressions. Notice Racket's dynamic typing—specifically lists with elements of different types—is essential. So we could build an expression like this:

```
(list 'Negate (list 'Add (list 'Const 2) (list 'Const 2)))
```

Then functions processing expressions could check the `car` of the list to see what sort of expression they have and we could use cond-expressions to "code up" ML-style case expressions without the pattern-matching.

However, it is bad style to assume this sort of data representation all over a large program. Instead, we should *abstract* our decisions into helper functions and then use this *interface* throughout our program. While the entire notion of an exp datatype is just "in our head" (nowhere does our Racket code define what an exp is), we still know what we need, namely:

- A way to build each kind of expression (constructors)
- A way to test for the different possibilities (predicates)
- A way to extract the different pieces of each kind of expression

To keep things simple, we won't have the extractors raise errors for the wrong kind of thing, though we probably should. Defining all the helper functions is easy:[2]

```
(define (Const i)   (list 'Const i))
(define (Add e1 e2) (list 'Add e1 e2))
(define (Negate e)  (list 'Negate e))

(define (Const? x)  (eq? (car x) 'Const))
(define (Add? x)    (eq? (car x) 'Add))
(define (Negate? x) (eq? (car x) 'Negate))

(define Const-int cadr)
(define Add-e1 cadr)
(define Add-e2 caddr)
(define Negate-e cadr)
```

Now we can write expressions in a much better, more abstract style:

```
(Negate (Add (Const 2) (Const 2)))
```

Here is an *interpreter* for our little expression language, using the interface we defined:

```
(define (eval-exp e)
  (cond [(Const? e) e]
        [(Add? e) (let ([v1 (Const-int (eval-exp (Add-e1 e)))]
                        [v2 (Const-int (eval-exp (Add-e2 e)))])
                    (Const (+ v1 v2)))]
        [(Negate? e) (Const (- 0 (Const-int (eval-exp (Negate-e e)))))]
        [#t (error "eval-exp expected an exp")]))
```

[2] `cadr` and `caddr` are built-in functions that are shorter versions of `(lambda(x) (car (cdr x)))` and `(lambda(x) (car (cdr (cdr x))))` respectively. Similar functions are predefined for any combination of `car` and `cdr` up to four uses deep. This support reflects the historical idiom of using cons cells to code up various datatypes.

As discussed more below, our interpreter takes an expression (in this case for arithmetic) and produces a value (an expression that cannot be evaluated more; in this case a constant).

While this sort of interface is good style in struct-less Racket, it still requires us to remember to use it and "not cheat." Nothing prevents a bad programmer from writing `(list 'Add #f)`, which will lead to a strange error if passed to `eval-exp`. Conversely, while we think of `(Add (Const 2) (Const 2))` as an expression in our little language, it actually gets evaluated to a list and there is nothing to keep some other part of our program from taking the result of evaluating `(Add (Const 2) (Const 2))` and misusing it.

Better: Racket structs

The `struct` special form, along with Racket's module system (see later lecture), fixes the abstraction problems described above. Using `struct` is also much more concise than all the helper functions we wrote above. To use it, just write something like:

```
(struct card (suit value))
```

This defines a new "struct" called `card` that is like an ML constructor. It adds to the environment a constructor, a predicate, and accessors for the fields. The names of these bindings are formed systematically from the constructor name `card` as followed:

- `card` is a function that takes two arguments and returns a value that is a card with a suit field holding the first argument and a value field holding the second argument
- `card?` is a function that takes one argument and returns `#t` for values created by calling `card` and `#f` for everything else
- `card-suit` is a function that takes a card and returns the contents of the suit field, raising an error if passed a non-card
- `card-value` is similar to `card-suit` for the value field

There are some useful *attributes* we can pass to struct definitions to modify their behavior, two of which we discuss here and will find useful.

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 earlier definition, the result of `(card 'Hearts (+ 3 7))` prints as `#<card>` as will any value for which `card?` returns `#t`. But with

```
(struct card (suit value) #:transparent)
```

the result of `(card 'Hearts (+ 3 7))` prints as `(card 'Hearts 10)`. 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-card-suit!` and `set-card-value!`. In short, it is up to you when creating a struct to decide whether the advantages of having mutable fields outweigh the disadvantages. It is also possible to make some fields mutable and some fields immutable.

Here is a different definition of an expression language and an interpreter for it that uses `struct`:

```
(struct const (int) #:transparent)
```

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

(define (eval-exp e)
  (cond [(const? e) e]
        [(add? e) (let ([v1 (const-int (eval-exp (add-e1 e)))]
                        [v2 (const-int (eval-exp (add-e2 e)))])
                    (const (+ v1 v2)))]
        [(negate? e) (const (- 0 (const-int (eval-exp (negate-e e)))))]
        [#t (error "eval-exp expected an exp")]))
```

An example expression for this language is:

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

The key semantic difference between `struct` and the more manual approach we took first has to do with predicates. When we wrote `(list 'Const 4)` we made something that causes `pair?` to return `#t`. With `(const 4)`, every type predicate in Racket returns `#f` except for `const?`. This makes `struct` quite special: you cannot define something like it as a function or a macro. For abstraction, getting a brand new kind of thing is very powerful. However, it means we cannot write "generic" code that crawls all over any kind of value since no code in our program knows all the structs that may be defined.[3]

Implementing Programming Languages: Interpreters and Compilers

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 way 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 implementing them requires only using simpler features we already understand. Third, many programming tasks are analogous to implementing an interpreter for programming language. For example, processing a structured document like a pdf file and turning it into a rectangle of pixels for displaying is aimilar to taking an input program and turning it into an answer.

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

While there are certainly many "pure" interpreters and compilers, many modern systems 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 your modern Intel x86 processor actually has a translator in the hardware that converts the binary instructions into smaller simpler instructions write 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.

[3] Actually, structs declared with `#:transparent` allow clients to query what fields they have, but we will not use this abstraction-breaking feature. We use `#:transparent` for the convenience of having the printer show the contents of values built from structs.

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 N/J for example compiles each module/binding to binary code.

A separate course (CSE401) focuses on techniques for constructing compilers, especially, as one assumes unless otherwise specified with "compiler," when the target language is assembly (as you see in CSE351).

Semi-Digression: eval

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`, which just happens to be a Racket special form. In fact, the nested lists together are a perfectly good representation of a Racket expression. And the `eval` primitive takes such a representation and, at run-time, evaluates it. 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 tool that is sometimes what you want.

Can a compiler-based language implementation (notice I didn't 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."

Parsing and Source-Code Analysis (e.g., Type-Checking)

You normally write your program in a text file, which is basically one long string. Strings are not convenient data structures for the core work of an interpreter or compiler, so the first phase of a language implementation is usally *parsing*: converting the string into an *abstract syntax tree (AST)*, which really is just a tree (as you could build with ML datatypes, Racket structs, or Java classes) describing the program structure. A parse error is when this process fails, for example due to unmatched parentheses. Parsing is a major topic in the compilers course; we will not discuss it.

After parsing, we might still perform additional checks to reject a program that "doesn't make sense." ML performs many such checks in the name of type-checking, such as making sure you do not use undefined

variables. Racket performs many fewer checks. Of course, even programs that pass all the "compile-time checks" (this term is standard even if the implementation uses an interpreter; "static checking" is also common) may still have run-time errors. An entire lecture on whether it is better to have more or less static checking is coming soon.

Implementing an Interpreter (and Skipping Parsing / Static Checking)

Our `eval-exp` functions above are perfect examples of interpreters for a small programming language. The language here is exactly expressions properly built from the constructors for constants, negations, and addition expressions. The definition of "properly" depends on the language; here we mean constants hold numbers and negations/additions hold other proper subexpressions. We also need a definition of *values* 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 happend to parsing and static 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)"`. We are also assuming that we will not encounter nonsensical things like `(add #t #f)`.

For this tiny language, there are no run-time errors. Suppose we also had constructors for string constants and operations like concatenating strings. Then we could view negation applied to a string as a run-time error, something the interpreter should check for (by checking that the recursive result of evaluating negation's subexpression is a constant) and give an appropriate error message.

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 lecture 1, since expressions can have variables, evaluating them requires an environment that maps variables to values. So an interpreter for a language with expressions needs a recursive helper function that takes an expression and an environment and produces a value. (In fact, for languages with features like mutation or exceptions, the helper function needs even more parameters.) To evaluate a subexpression where more variables are in scope requires passing a larger environment to the recursive call.

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 do and some of the fancier data structures for dictionaries you learn in a data structures course (CSE 332) are appropriate. For our purposes using Racket as our metalanguage, a simple association list holding pairs of strings (variable names) and values (what the variables are bound to) can suffice.

Higher-Order Functions and Closures

If our language supports higher-order functions and lexical scope, then 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 rather is direct: It literally creates 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.

At a function call, we evaluate the function part to a value (a closure, else we have an error like trying to treat a number as a function) and the argument to another value. Next 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.*

That really is how interpreters implement higher-order functions.

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 only storing 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 definition are outside the function body) and the environment we store in the closure needs only these variables.

Finally, you might wonder how compilers implement closures when the target language does not have higher-order functions. 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).

Using the meta-language for adding "macros"

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 and produces an arithmetic-expression-language value. So for exmple, 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 (triple x) ; produce an arith-exp that when interpreted triples
   (add x (add x x)))
```

Here `triple` is a Racket function that takes an arithmetic expression and produces an arithmetic expression. Calling `triple` produces abstract syntax in our language, much like macro expansion. for example, `(negate (triple (negate (const 4))))` produces `(negate (add (negate (const 4)) (add (negate (const 4)) (negate const 4))))`. Notice this "macro" `triple` 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 "embeddeing" our little language inside the Racket metalanguage. The same technique works regardless of the choice of metalanguage.