# CSE 341, Spring 2008, Lecture 17 Summary

*Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.*

This lecture covers three topics that are all directly relevant to homework 5:

- DrScheme's `define-struct` for defining new (dynamic) types in Scheme — and contrasting that with manually tagged lists
- Writing interpreters to implement programming languages — and its relation to "steps" we are skipping (like parsing)
- Implementing higher-order functions — the role of environments and closures

**define-struct and related idioms:**

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. Scheme's dynamic typing makes the issue less central since we can simply use pairs (or more generally lists) to build anything we want. However, the *concept* of one-of types is still prevalent. Moreover, DrScheme extends the standard Scheme language with something very similar to ML's constructors — a special form called `define-struct` — and it is instructive to contrast the two language's approaches.

Before seeing `define-struct`, let's consider "Pure Scheme." 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 — 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* Scheme 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 computing (e.g., addition) and making a new "tagged" value (e.g., a new number). These "tags" are like ML constructors, but it's like our whole language has just one datatype and everything is in it.

Given Pure Scheme, we can still "code up" our own datatypes as an *idiom*. One common approach is to use lists where the first list element is a symbol 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
```

We could decide to represent similar values in Scheme by using lists where the first element is `'Const`, `'Add`, or `'Negate`. If it is `'Const` we would have one more list element which is a number. If it is `'Add` we would have two more list elements which are expressions. Notice Scheme'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)))
```

And then functions processing expressions could check the `car` of the list to see what sort of expression they have.

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" (Scheme has no way for us to specify it except in comments), 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 arguably we should. Defining all the helper functions is easy:

```
(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")]))
```

As we'll discuss 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 Pure Scheme, it still requires us to remember to use it and "not cheat". Nothing is to prevent 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 using `set-car!` to mess it up.

`define-struct`, along with DrScheme's module system (see lecture 19), fixes these abstraction problems. `define-struct` is also much more concise than all the helper functions we wrote above. To use it, just write something like:

```
(define-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, selectors for the fields, and mutators for the fields. The names of all these bindings are formed systematically from the constructor name `card` as followed:

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

- `set-card-suit!` and `set-card-value!` take a card and mutate the appropriate field.

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

```
(define-struct const (int))
(define-struct add (e1 e2))
(define-struct negate (e))

(define (eval-exp2 e)
  (cond [(const? e) e]
        [(add? e) (let ([v1 (const-int (eval-exp2 (add-e1 e)))]
                        [v2 (const-int (eval-exp2 (add-e2 e)))])
                    (make-const (+ v1 v2)))]
        [(negate? e) (make-const (- 0 (const-int (eval-exp2 (negate-e e)))))]
        [#t (error "eval-exp2 expected an exp")]))
```

An example expression for this language is:

```
(make-negate (make-add (make-const 2) (make-const 2)))
```

The key semantic difference between `define-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 `(make-const 4)`, none of Pure Scheme's predicates return `#t`; only `const?` does. This makes `define-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 can't write "generic" code that crawls all over any kind of value. In Pure Scheme, `eval` (from the previous lecture) can actually be written *as a Scheme function*, which is pretty mind-bending. But that will not work with `define-struct` involved because `eval` could not be written in a way that knows about every `define-struct` that is in a program.

**Interpreters, environments, and implementing languages:**

While this course is mostly about what programming-language features *mean* and not how they are *implemented*, it is important to dispel any notion that things like higher-order functions or objects are "magic". So we will consider at a very high level how programming languages are implemented in general and features like higher-order functions are implemented in particular.

There are basically two ways to implement a programming language $A$. We could write an *interpreter* in another language $B$ that takes programs in $A$ and produces answers. Or we could write a *compiler* in another language $B$ that takes programs in $A$ and translates them into equivalent programs in another language $C$ (not *the* language C necessarily) that is already implemented. Conventionally when we think of compilers the *target language* for the translation is assembly, like you see in CSE378. In CSE401 you learn about compilers; on your next homework you will write an interpreter in Scheme for a small programming language.

Earlier in lecture we did exactly this where our language was expressions built from `make-add`, `make-const`, and `make-negate` (with appropriate arguments) and our interpreter was `eval-exp2`. It may seem strange that the programs for the arithmetic-expression language look like

```
(make-negate (make-add (make-const 2) (make-const 2)))
```

rather than some string representation like `"- (2 + 2)"`, which is admittedly more convenient. However, the version made from constructors is a nice tree data-structure that is very convenient for interpreters (and compilers). It is called an *abstract syntax tree*, abbreviated *AST*. The first thing "real" interpreters/compilers do is convert strings or files into ASTs. This is called *parsing* and it is covered in CSE401.

Rather than bother with parsing, we can just write ASTs in Scheme "manually". (As a side-note, Scheme's quoting from last lecture is also a great way to write programs that is more structured than strings but less verbose than calling constructors.) We can also easily write Scheme functions that return "programs" in the language we are defining. Returning an AST is easier than returning a string representation.