

## CSE341: Programming Languages

### Lecture 16 Datatype-Style Programming With Lists or Structs

Dan Grossman  
Autumn 2017

## The Goal

In ML, we often define datatypes and write recursive functions over them – how do we do analogous things in Racket?

- First way: With lists
- Second way: With structs [a new construct]
  - Contrast helps explain advantages of structs

Autumn 2017

CSE341: Programming Languages

2

## Life without datatypes

Racket has nothing like a datatype binding for one-of types

No need in a dynamically typed language:

- Can just mix values of different types and use primitives like `number?`, `string?`, `pair?`, etc. to “see what you have”
- Can use cons cells to build up any kind of data

Autumn 2017

CSE341: Programming Languages

3

## Mixed collections

In ML, cannot have a list of “ints or strings,” so use a datatype:

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

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

In Racket, dynamic typing makes this natural without explicit tags

- Instead, every value has a tag with primitives to check it
- So just check car of list with `number?` or `string?`

Autumn 2017

CSE341: Programming Languages

4

## Recursive structures

More interesting datatype-programming we know:

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

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

Autumn 2017

CSE341: Programming Languages

5

## Change how we do this

- Previous version of `eval_exp` has type `exp -> int`
- From now on will write such functions with type `exp -> exp`
- Why? Because will be interpreting languages with multiple kinds of results (ints, pairs, functions, ...)
  - Even though much more complicated for example so far
- How? See the ML code file:
  - Base case returns entire expression, e.g., `(Const 17)`
  - Recursive cases:
    - Check variant (e.g., make sure a `Const`)
    - Extract data (e.g., the number under the `Const`)
    - Also return an `exp` (e.g., create a new `Const`)

Autumn 2017

CSE341: Programming Languages

6

## New way in Racket

See the Racket code file for coding up the same new kind of "exp -> exp" interpreter

- Using lists where car of list encodes "what kind of exp"

Key points:

- Define our own constructor, test-variant, extract-data functions
  - Just better style than hard-to-read uses of `car`, `cdr`
- Same recursive structure without pattern-matching
- With no type system, no notion of "what is an exp" except in documentation
  - But if we use the helper functions correctly, then okay
  - Could add more explicit error-checking if desired

## Symbols

Will not focus on Racket *symbols* like `'foo`, but in brief:

- Syntactically start with quote character
- Like strings, can be almost any character sequence
- Unlike strings, compare two symbols with `eq?` which is fast

## New feature

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

Defines a new kind of thing and introduces several new functions:

- `(foo e1 e2 e3)` returns "a foo" with `bar`, `baz`, `quux` fields holding results of evaluating `e1`, `e2`, and `e3`
- `(foo? e)` evaluates `e` and returns `#t` if and only if the result is something that was made with the `foo` function
- `(foo-bar e)` evaluates `e`. If result was made with the `foo` function, return the contents of the `bar` field, else an error
- `(foo-baz e)` evaluates `e`. If result was made with the `foo` function, return the contents of the `baz` field, else an error
- `(foo-quux e)` evaluates `e`. If result was made with the `foo` function, return the contents of the `quux` field, else an error

## An idiom

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

For "datatypes" like exp, create one struct for each "kind of exp"

- structs are like ML constructors!
- But provide constructor, tester, and extractor functions
  - Instead of patterns
  - E.g., `const`, `const?`, `const-int`
- Dynamic typing means "these are the kinds of exp" is "in comments" rather than a *type system*
- Dynamic typing means "types" of fields are also "in comments"

## All we need

These structs are all we need to:

- Build trees representing expressions, e.g.,

```
(multiply (negate (add (const 2) (const 2)))
          (const 7))
```

- Build our `eval-exp` function (see code):

```
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e)
         (const (- (const-int
                    (eval-exp (negate-e e)))))]
        [(add? e) ...]
        [(multiply? e) ...]))
```

## Attributes

- `#:transparent` is an optional attribute on struct definitions
  - For us, prints struct values in the REPL rather than hiding them, which is convenient for debugging homework

- `#:mutable` is another optional attribute on struct definitions
  - Provides more functions, for example:

```
(struct card (suit rank) #:transparent #:mutable)
; also defines set-card-suit!, set-card-rank!
```

- Can decide if each struct supports mutation, with usual advantages and disadvantages

- As expected, we will avoid this attribute
- `mcons` is just a predefined mutable struct

## Contrasting Approaches

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

Versus

```
(define (add e1 e2) (list 'add e1 e2))
(define (add? e) (eq? (car e) 'add))
(define (add-e1 e) (car (cdr e)))
(define (add-e2 e) (car (cdr (cdr e))))
```

This is *not* a case of syntactic sugar

## The key difference

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

- The result of calling `(add x y)` is *not* a list
  - And there is no list for which `add?` returns `#t`
- `struct` makes a new kind of thing: extending Racket with a new kind of data
- So calling `car`, `cdr`, or `mult-e1` on “an add” is a run-time error

## List approach is error-prone

```
(define (add e1 e2) (list 'add e1 e2))
(define (add? e) (eq? (car e) 'add))
(define (add-e1 e) (car (cdr e)))
(define (add-e2 e) (car (cdr (cdr e))))
```

- Can break abstraction by using `car`, `cdr`, and list-library functions directly on “add expressions”
  - Silent likely error:  

```
(define xs (list (add (const 1)(const 4)) ...))
(car (car xs))
```
- Can make data that `add?` wrongly answers `#t` to  

```
(cons 'add "I am not an add")
```

## Summary of advantages

Struct approach:

- Is better style and more concise for *defining* data types
- Is about equally convenient for *using* data types
- But much better at timely errors when *misusing* data types
  - Cannot use accessor functions on wrong kind of data
  - Cannot confuse tester functions

## More with abstraction

Struct approach is even better combined with other Racket features not discussed here:

- The *module system* lets us hide the constructor function to enforce invariants
  - List-approach cannot hide cons from clients
  - Dynamically-typed languages can have abstract types by letting modules define new types!
- The *contract system* lets us check invariants even if constructor is exposed
  - For example, fields of “an add” must also be “expressions”

## Struct is special

Often we end up learning that some convenient feature could be coded up with other features

Not so with struct definitions:

- A function cannot introduce multiple bindings
- Neither functions nor macros can create a new kind of data
  - Result of constructor function returns `#f` for every other tester function: `number?`, `pair?`, other structs' tester functions, etc.