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

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

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?`
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)
```

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`)

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

(\textbf{struct foo (bar baz quux) #:transparent})

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

- \((\text{foo } e_1 \ e_2 \ e_3)\) returns "a foo" with \(\text{bar, baz, quux}\) fields holding results of evaluating \(e_1, e_2,\) and \(e_3\)
- \((\text{foo? } \ e)\) evaluates \(\ e\) and returns \#t only if the result is something that was made with the \(\text{foo}\) function
- \((\text{foo-bar } \ e)\) evaluates \(\ e\). If result was made with the \(\text{foo}\) function, return the contents of the \(\text{bar}\) field, else an error
- \((\text{foo-baz } \ e)\) evaluates \(\ e\). If result was made with the \(\text{foo}\) function, return the contents of the \(\text{baz}\) field, else an error
- \((\text{foo-quux } \ e)\) evaluates \(\ e\). If result was made with the \(\text{foo}\) function, return the contents of the \(\text{quux}\) field, else an error

Winter 2018  
CSE 341: Programming Languages  
9

An idiom

(\textbf{struct const (int) #:transparent})  
(\textbf{struct negate (e) #:transparent})  
(\textbf{struct add (el e2) #:transparent})  
(\textbf{struct multiply (el e2) #:transparent})

For "datatypes" like \textit{exp}, create one \textbf{struct} for each "kind of \textit{exp}"
  
  - structs are like ML constructors!
  
  - But provide constructor, tester, and extractor functions
  
    - Instead of patterns
    
      - E.g., \textbf{const}, \textbf{const?}, \textbf{const-int}
    
    - Dynamic typing means "these are the kinds of \textit{exp}" is "in comments" rather than a \textit{type system}
    
    - Dynamic typing means "types" of fields are also "in comments"

Winter 2018  
CSE 341: Programming Languages  
10

All we need

These structs are all we need to:

- Build trees representing expressions, e.g.,
  
  \((\text{multiply } (\text{negate } (\text{add } (\text{const 2}) (\text{const 2})))) (\text{const 7}))\)

- Build our \textbf{eval-exp} function (see code):

  \begin{verbatim}
  (define (eval-exp e)  
    (cond  
      [(const? e) e]  
      [(negate? e)  
        (const (- (const-int (eval-exp (negate-e e))))))]  
      [(add? e) ...]  
      [(multiply? e) ...]...  
  )
  \end{verbatim}

Winter 2018  
CSE 341: Programming Languages  
11

Attributes

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

- \#:mutable is another optional attribute on \textbf{struct} definitions
  
    - Provides more functions, for example:
    
      \begin{verbatim}
      (struct card (suit rank) #:transparent #:mutable)  
      \end{verbatim}
      
    - Also defines \textbf{set-card-suit!}, \textbf{set-card-rank!}

    - Can decide if each \textbf{struct} supports mutation, with usual advantages and disadvantages
      
      - As expected, we will avoid this attribute

    - \textbf{mcons} is just a predefined mutable \textbf{struct}

Winter 2018  
CSE 341: Programming Languages  
12
Contrasting Approaches

\[(\text{struct add } (e_1 \ e_2) \ #:\text{transparent})\]

Versus

\[(\text{define } (\text{add } e_1 e_2) (\text{list } '\text{add} \ e_1 \ e_2))\]
\[(\text{define } (\text{add? } e) (\text{eq? } (\text{car } e) '\text{add}))\]
\[(\text{define } (\text{add-e1 } e) (\text{car } (\text{cdr } e)))\]
\[(\text{define } (\text{add-e2 } e) (\text{car } (\text{cdr } (\text{cdr } e))))\]

This is not a case of syntactic sugar

List approach is error-prone

\[(\text{define } (\text{add } e_1 e_2) (\text{list } '\text{add} \ e_1 \ e_2))\]
\[(\text{define } (\text{add? } e) (\text{eq? } (\text{car } e) '\text{add}))\]
\[(\text{define } (\text{add-e1 } e) (\text{car } (\text{cdr } e)))\]
\[(\text{define } (\text{add-e2 } e) (\text{car } (\text{cdr } (\text{cdr } e))))\]

- Can break abstraction by using \text{car}, \text{cdr}, and list-library functions directly on “add expressions”
  - Silent likely error:
    \[(\text{define } xs (\text{list } (\text{add } (\text{const } 1) (\text{const } 4)) ...))\]
    \[(\text{car } (\text{car } xs))\]

- Can make data that \text{add?} wrongly answers \#t to
  \[(\text{cons } '\text{add } "I \text{ am not an add")\}

The key difference

\[(\text{struct add } (e_1 \ e_2) \ #:\text{transparent})\]

- The result of calling \text{(add x y)} is not a list
  - And there is no list for which \text{add?} returns \#t

- \text{struct} makes a new kind of thing: extending Racket with a new kind of data

- So calling \text{car}, \text{cdr}, or mult-e1 on “an add” is a run-time error

Summary of advantages

Struct approach:

- Is better style and more concise for \textit{defining} data types

- Is about equally convenient for \textit{using} data types

- But much better at timely errors when \textit{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.