CSE413: Programming Languages and Implementation

Racket structs
Implementing languages with interpreters
Implementing closures

Dan Grossman
Fall 2014
Hi!

• I’m not Hal 😊

• I love this stuff and have taught this stuff many times
  – But not in CSE413
  – So stop me if I use jargon you don’t know, etc.

• This stuff is crucial for your next homework
  – Without it you will be totally lost
  – With it, it’s challenging but straightforward

• This material hopefully takes just about two class periods
  – See also code
Announcements from Hal

• Reminder: HW4 due Thursday night, 11PM

• HW5 Posted, due a week from Thursday, 11PM

• Midterm the following Monday, November 3, in class
Goals

1. Learn how to write interpreters for implementing programming languages

2. In particular, use closures to implement higher-order functions

3. Represent the code of one language as data in another language

4. Learn Racket’s structs as a better way than lists to represent the code of another language

Goals will overlap, but very roughly will go 3-and-4, then 1, then 2
Dynamic typing + lists = everything

We know:
- Racket has lists
- Racket is dynamically typed

So: (nested) lists can hold any kind of tree-shaped data
  - 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

“This works” – see code for a little language of arithmetic expressions
Comments on what we did

*Using lists where car of list encodes “what kind of expression”*

Key points:

- Define our own constructor, test-variant, extract-data functions
  - Just better style than hard-to-read uses of `car`, `cdr`
- Elegant recursive structure with a “big cond”
- With no type system, no notion of “what is an expression” except in documentation
  - But if we use the helper functions correctly, then okay
  - Could add more explicit error-checking if desired

Note: Use of symbols and `eq?` is idiomatic Racket, but not necessary
New feature

(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 \texttt{bar}, \texttt{baz}, \texttt{quux} fields holding results of evaluating \(e_1\), \(e_2\), and \(e_3\)

• \((\text{foo-}\text{? } e)\) evaluates \(e\) and returns \#t if and only if the result is something that was made with the \texttt{foo} function

• \((\text{foo-}\text{bar } e)\) evaluates \(e\). If result was made with the \texttt{foo} function, return the contents of the \texttt{bar} field, else an error

• \((\text{foo-}\text{baz } e)\) evaluates \(e\). If result was made with the \texttt{foo} function, return the contents of the \texttt{baz} field, else an error

• \((\text{foo-}\text{quux } e)\) evaluates \(e\). If result was made with the \texttt{foo} function, return the contents of the \texttt{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 “types” like expression, create one struct for each “kind of exp”

- Conveniently defines constructor, tester, and extractor functions
  - E.g., const, const?, const-int
- Dynamic typing means “these are the kinds of expression” is “still [just] in comments”
- Dynamic typing means “types” of fields are also “just in comments”
All we need

These structs are all we need to:

- Build trees representing expressions, e.g.,

  \[
  (\text{multiply} \ (\text{n}egate \ (\text{add} \ (\text{const} \ 2) \ (\text{const} \ 2)))) \ \\
  (\text{const} \ 7))
  \]

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

  \[
  (\text{define} \ (\text{eval-exp} \ e) \ \\
  (\text{cond} \ [(\text{const?} \ e) \ e] \ \\
  [(\text{n}egate? \ e) \ \\
  (\text{const} \ (- \ (\text{const-int} \ \\
  (\text{eval-exp} \ (\text{n}egate-e \ e))))))]) \ \\
  [(\text{add?} \ e) \ ...] \ \\
  [(\text{multiply?} \ e) \ ...]...
  \]
Attributes

• #:transparent is an optional attribute on struct definitions
  – For us, prints struct values in the REPL (interactions window) rather than hiding them, which is convenient for debugging homework

• #:mutable is another optional attribute on struct definitions

(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
    • We will avoid this attribute; guarantees no mutation
  – mcons is just a predefined mutable struct
Contrasting Approaches

\[(\text{struct add (e1 e2) #:transparent})\]

Versus

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

This is \textit{not} a case of “syntactic sugar”
- Syntactic sugar: More convenient syntax for writing something already in the language
The key difference

\[(\text{struct add (e1 e2) #:transparent})\]

- 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 \text{mult-e1} on “an add” is a run-time error
  - Not true for the version with lists (!)
List approach is error-prone

```scheme
(define (add e1 e2) (list 'add e1 e2))
(define (add? e) (eq? (car e) 'add))
(define (add-e1 e) (car (cdr e)))
 DEFINE (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:
    ```scheme
    (define xs (list (add (const 1)(const 4)) ...))
    (car (car xs))
    ```

- Can make data that `add?` wrongly answers `#t` to
  ```scheme
  (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

But: Still doesn’t enforce that fields “have the right type”
  – Not covering: Could use Racket’s modules or contracts to do that
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

- Creating a new kind of data has to be a “built-in primitive”
  - Result of constructor function returns \#f for every other tester function: `number?`, `pair?`, other structs’ tester functions, etc.
Now…

A step back to talk about general approaches to implementing programming languages…
Implementing languages

Much of course has been fundamental concepts for using PLs
  – Syntax, semantics, idioms
  – Important concepts like closures, delayed evaluation, …

But also valuable (and fun!) to learn basics of implementing PLs
  – Requires fully understanding semantics
  – Things like closures and objects are not “magic”
  – Many programming techniques are related/similar
    • Example: rendering a document (“program” is the structured document, “pixels” is the output)
  – Substantial part of CSE413 course description
Typical workflow

concrete syntax (string)
"(fn x => x + x) 4"

abstract syntax (tree)

Function
  \( \text{Var} \ x \)
  +
  \( \text{Var} \ x \)

Constant 4

Call

Possible errors / warnings

Type checking?

Possible errors / warnings

Rest of implementation
**Interpreter or compiler**

So “rest of implementation” takes the abstract syntax tree (AST) and “runs the program” to produce a result.

Fundamentally, two approaches to implement a PL \( B \):

- Write an **interpreter** in another language \( A \)
  - Better names: evaluator, executor
  - Take a program in \( B \) and produce an answer (in \( B \))

- Write a **compiler** in another language \( A \) to a third language \( C \)
  - Better name: translator
  - Translation must *preserve meaning* (equivalence)

We call \( A \) the **metalanguage**
  - Crucial to keep \( A \) and \( B \) straight
Reality more complicated

Evaluation (interpreter) and translation (compiler) are your options
- But in modern practice have both and multiple layers

A plausible example:
- Java compiler to bytecode intermediate language
- Have an interpreter for bytecode (itself in binary), but compile frequent functions to binary at run-time
- The chip is itself an interpreter for binary
  - Well, except these days the x86 has a translator in hardware to more primitive micro-operations it then executes

Racket implementation itself uses a similar mix
Interpreter versus compiler versus combinations is about a particular language implementation, not the language definition.

So there is no such thing as a “compiled language” or an “interpreted language”

- Programs cannot “see” how the implementation works

Unfortunately, you often hear such phrases

- “C is faster because it’s compiled and LISP is interpreted”
- This is nonsense; politely correct people

- (Admittedly, languages with “eval” must “ship with some implementation of the language” in each program)
Typical workflow

Concrete syntax (string)
"(fn x => x + x) 7"

Abstract syntax (tree)

Parsering

Possible errors / warnings

Possible errors / warnings

Function

Constant

Interpreter or translator

Type checking?
Skipping parsing [until later in quarter]

- If implementing PL $B$ in PL $A$, we can skip parsing
  - Have $B$ programmers write ASTs directly in PL $A$
  - Not so bad with Racket structs
  - Embeds $B$ programs as trees in $A$

```
; define B’s abstract syntax
(struct call ...)
(struct function ...)
(struct var ...)
...

; example B program
(call (function (list "x")
  (add (var "x")
    (var "x")
  )
  (const 4))
```

Call

Function

Constant

+ 4

x

x

x

x

Var

Var

Var

Var
Already did an example!

- Let the metalanguage $A = \text{Racket}$
- Let the language-implemented $B = \text{"Arithmetic Language"}$
- Arithmetic programs written with calls to Racket constructors
- The interpreter is `eval-exp`

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

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

Racket data structure is Arithmetic Language program, which eval-exp runs
What we know

- Define (abstract) syntax of language $B$ with Racket structs
  - $B$ called MUPL in homework
- Write $B$ programs directly in Racket via constructors
- Implement interpreter for $B$ as a (recursive) Racket function

Now, a subtle-but-important distinction:
- Interpreter can assume input is a “legal AST for $B$”
  - Okay to give wrong answer or inscrutable error otherwise
- Interpreter must check that recursive results are the right kind of value
  - Give a good error message otherwise
Legal ASTs

- “Trees the interpreter must handle” are a subset of all the trees Racket allows as a dynamically typed language

\[
\begin{align*}
\text{(struct } & \text{ const } (\text{int}) \ #:\text{transparent}) \\
\text{(struct } & \text{ negate } (\text{e}) \ #:\text{transparent}) \\
\text{(struct } & \text{ add } (\text{e1 e2}) \ #:\text{transparent}) \\
\text{(struct } & \text{ multiply } (\text{e1 e2}) \ #:\text{transparent})
\end{align*}
\]

- Can assume “right types” for struct fields
  - \text{const} holds a number
  - \text{negate} holds a legal AST
  - \text{add} and \text{multiply} hold 2 legal ASTs

- Illegal ASTs can “crash the interpreter” – \textit{this is fine}

\[
\begin{align*}
\text{(multiply } & \text{ (add } (\text{const 3}) \ "uh-oh") \ (\text{const 4})\}) \\
\text{(negate } & \text{ -7})
\end{align*}
\]
**Interpreter results**

- Our interpreters return expressions, but not any expressions
  - Result should always be a *value-in-language-being-interpreted*, a kind of expression that evaluates to itself
  - If not, the interpreter has a bug
- So far, only values are from `const`, e.g., `(const 17)`
- But a larger language has more values than just numbers
  - Booleans, strings, etc.
  - Pairs of values (definition of value recursive)
  - Closures
  - ...
Example

See code for language that adds booleans, number-comparison, and conditionals:

```plaintext
(struct bool (b) #:transparent)
(struct eq-num (e1 e2) #:transparent)
(struct if-then-else (e1 e2 e3) #:transparent)
```

What if the program is a legal AST, but evaluation of it tries to use the wrong kind of value?

- For example, “add a boolean”
- You should detect this and give an error message not in terms of the interpreter implementation
- Means checking a recursive result whenever a particular kind of value is needed
  - No need to check if any kind of value is okay
Dealing with variables

• Interpreters so far have been for languages without variables
  – No let-expressions, functions-with-arguments, etc.
  – Language in homework has all these things

• This segment describes in English what to do
  – Up to you to translate this to code

• Fortunately, what you have to implement is what you have been using conceptually throughout course
Dealing with variables

• An environment is a mapping from variables (Racket strings) to values (as defined by the language)
  – Only ever put pairs of strings and values in the environment

• Evaluation takes place in an environment
  – Environment passed as argument to interpreter helper function
  – A variable expression looks up the variable in the environment
  – Most subexpressions use same environment as outer expression
  – A let-expression evaluates its body in a larger environment
The Set-up

So now a recursive helper function has all the interesting stuff:

```
(define (eval-under-env e env)
  (cond ...
    ; case for each kind of 
  )
)
```

– Recursive calls must “pass down” correct environment

Then `eval-exp` just calls `eval-under-env` with same expression and the *empty environment*

On homework, environments themselves are just Racket lists containing Racket pairs of a string (the MUPL variable name, e.g., "x") and a MUPL value (e.g., `(int 17)`)
A grading detail

- Stylistically `eval-under-env` would be a helper function one could define locally inside `eval-exp`.

- But homework has you not do this
  - Helps with grading tests to call `eval-under-env` directly, so we need it at top-level.
The best part

- The most interesting and mind-bending part of the homework is that the language being implemented has first-class closures
  - With lexical scope of course

- Fortunately, what you have to implement is what you have previously learned about closures…
Higher-order functions

The “magic”: How do we use the “right environment” for lexical scope when functions may return other functions, store them in data structures, etc.?

Lack of magic: The interpreter uses a closure data structure (with two parts) to keep the environment it will need to use later

```scheme
(struct closure (env fun) #:transparent)
```

Evaluate a function expression:
- A function is not a value; a closure is a value
  - Evaluating a function returns a closure
- Create a closure out of (a) the function and (b) the current environment when the function was evaluated

Evaluate a function call:
- ...
Function calls

• Use current environment to evaluate e1 to a closure
  – Error if result is a value that is not a closure
• Use current environment to evaluate e2 to a value
• Evaluate closure’s function’s body in the closure’s environment, extended to:
  – Map the function’s argument-name to the argument-value
  – And for recursion, map the function’s name to the whole closure

This is the same semantics you learned already “coded up”

Given a closure, the code part is only ever evaluated using the environment part (extended), not the environment at the call-site
Optional: Is that expensive?

- **Time** to build a closure is tiny: a struct with two fields

- **Space** to store closures *might* be large if environment is large
  - But environments are immutable, so natural and correct to have lots of sharing, e.g., of list tails
  - Still, end up keeping around bindings that are not needed

- Alternative used in practice: When creating a closure, store a possibly-smaller environment holding only the variables that are *free variables* in the function body
  - Free variables: Variables that occur, not counting shadowed uses of the same variable name
  - A function body would never need anything else from the environment
Optional: Free variables examples

(lambd() (+ x y z)) ; {x, y, z}

(lambd(x) (+ x y z)) ; {y, z}

(lambd(x) (if x y z)) ; {y, z}

(lambd(x) (let ([y 0]) (+ x y z))) ; {z}

(lambd(x y z) (+ x y z)) ; {} 

(lambd(x) (+ y (let ([y z]) (+ y y)))) ; {y, z}
Optional: Computing free variables

• So does the interpreter have to analyze the code body every time it creates a closure?

• No: Before evaluation begins, compute free variables of every function in program and store this information with the function

• Compared to naïve store-entire-environment approach, building a closure now takes more time but less space
  – And time proportional to number of free variables
  – And various optimizations are possible

• [Also use a much better data structure for looking up variables than a list]
Optional: compiling higher-order functions

- If we are compiling to a language without closures (like assembly), cannot rely on there being a “current environment”

- So compile functions by having the translation produce “regular” functions that all take an *extra explicit argument* called “environment”

- And compiler replaces all uses of free variables with code that looks up the variable using the environment argument
  - Can make these fast operations with some tricks

- Running program still creates closures and every function call passes the closure’s environment to the closure’s code
Recall…

Our approach to language implementation:

• Implementing language \( B \) in language \( A \)
• Skipping parsing by writing language \( B \) programs directly in terms of language \( A \) constructors
• An interpreter written in \( A \) recursively evaluates

Now: The idea of using \( A \) functions like \( B \) macros

• “Feels like” extending the syntax of \( B \)
• Use of a “macro” expands into language syntax before the program is run, i.e., before calling the main interpreter function
Put it together

With our set-up, we can use language A (i.e., Racket) functions that produce language B abstract syntax as language B “macros”

- Language B programs can use the “macros” as though they are part of language B
- No change to the interpreter or struct definitions
- Just a programming idiom enabled by our set-up
  - Helps teach what macros are
- See code for example “macro” definitions and “macro” uses
  - “macro expansion” happens before calling eval-exp