CSE 413
Languages & Implementation

Hal Perkins

Autumn 2012

Structs, Implementing Languages
(credits: Dan Grossman, CSE 341)

Outline

- Representing programs
- Racket structs
- Implementing programming languages
  - Interpreters

Data structures in Racket

We've been using functions to abstract from lists

```
(make-expr left op right) =>
               (list left op right)
(operator expr) => (cadr expr)
```

etc.

We could also build weakly "typed" or self-describing data by tagging each list:

```
(define (const i)    (list `const i))
(define (add e1 e2) (list `add e1 e2))
(define (negate e)   (list `negate e))
```

Racket structs

Racket provides structs with fields

Makes a new type (different from pair?, etc.)

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

Provides constructor, predicate, accessors

```
(define exp (add 3 4))
(add? exp) => true
(pair? exp) => false
(add-e1 exp) => 3  (add-e2 exp) => 4
```

Representing program as trees

Can use either lists or structs (we'll use structs) to build trees to represent compound data & programs

```
(add (const 4)
     (negate (add (const 1)
                  (negate (const 7)))))
```

There's nothing that ties **`add`**, **`negate`**, **`const`** together as the "expression type" other than the convention we have in our heads and in program comments

Implementing programming languages

Much of the course has been about fundamental concepts for *using* PLs

- Syntax, semantics, idioms
- Important concepts like closures, delayed evaluation, …

But we also want to learn basics of *implementing* PLs

- Requires fully understanding semantics
- Things like closures and objects are not “magic”
- Many programming techniques are related/similar
  - Ex: rendering a document (“program” is the structured document, “pixels” is the output)

Implementing languages

Two fundamental ways to implement a prog. lang. A

Write an *interpreter* in another language B

- Read program in A as data, carry out its instructions, and produce an answer (in A)
- (Better names: evaluator, executor)

Write a *compiler* in another language B

- Read program in A as data, produce an equivalent program in another language C
- Translation must *preserve meaning*
- (Better name: translator)

It’s really more complicated

Evaluation (interpreter) and translation (compiler) are the options, but many languages are implemented with both and have multiple layers

Example: Java

- Compiler to bytecode intermediate language (.class)
- Can interpret the bytecode directly, but also
- Compile frequently executed code to binary
- The chip is an interpreter for binary
  - Except these days the chip translates x86 binary to a more primitive code that it executes

Racket uses a similar mix

Sermon (er, rant)

Interpreter vs compiler vs combinations is about a language implementation, not language definition

There is no such thing as a “compiled language” or “interpreted language”

- Program cannot see how the implementation works

Unfortunately you hear nonsense like this all the time:

- “C is faster because it’s compiled and LISP is interpreted”
- Nonsense: You can write a C interpreter or a LISP compiler
- Please politely correct your managers, friends, and other professors. ☺

OK, they do have a point

A traditional compiler does not need the language implementation to run the program

- Can "ship the binary" without the compiler

But Racket, Scheme, Javascript, Ruby, … have **`eval`**

- At runtime can create data and treat it as a program
- Then run that program
- So we need an implementation (compiler, interpreter, combination) at runtime

It is also true that some languages are designed with a particular implementation strategy in mind, but it doesn't mean they couldn't be implemented differently.

Embedding one language in another

How is **`(negate (add (const 2) (const 2)))`** a “program” compared to “-(2+2)” ?

A traditional implementation includes a *parser* to read the string “-(2+2)” and turn it into a tree-like data structure called an *abstract syntax tree (AST)*.

- Ideal representation for either interpreting or as an intermediate stage in compiling
- For now we’ll create trees directly and interpret them. Parsing later in the quarter.
- We’ll also assume perfect programmers and not worry about syntax or semantic errors.

The arith-exp example

This embedding approach is exactly what we did to represent the language of arithmetic expressions using Racket structs

```
(struct const (i)    #:transparent)
(struct add (e1 e2) #:transparent)
(struct negate (e)   #:transparent)
(add (const 4)
     (negate (add (const 1)
                  (negate (const 7)))))
```

The missing piece is to define the interpreter

```
(define (eval-exp e) … )
```

The interpreter

An interpreter takes programs in the language and produces values (answers) in the language

- Typically via recursive helper functions with cases
- This example is so simple we don't need helpers and can assume all recursive results are constants

```
(define (eval-exp e)
  (cond
    ((const? e) e)
    ((add? e)
     (const (+ (const-i (eval-exp (add-e1 e)))
               (const-i (eval-exp (add-e2 e))))))
    ((negate? e)
     (const (- (const-i (eval-exp (negate-e e))))))
    (#t (error "eval-exp expected an expression"))))
```

“Macros”

Another advantage of the embedding approach is we can use the metalanguage to define helper functions that create (new) programs in our language

- They generate the (abstract) syntax
- Result can *then* be put in a larger program or evaluated

Example:

```
(define (triple x) (add x (add x x)))
(define p (add (const 1 (triple (const 2)))))
```

(all this does in create a program with 4 constant expressions)

What’s missing

- Two major things missing from this language
  - Local variables
  - Higher-order functions with lexical scope (closures)

- To support local variables:
  - Interpreter helper function(s) need an *environment* as an additional argument
    - Environment maps names to values
    - A Racket association list works fine for us
  - Evaluate a variable expression by looking up the name
  - A let-body is evaluated in an augmented environment with the local bindings

Higher-order functions

The “magic”: How is the “right environment” round 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

To evaluate a function expression:

- A function is not a value, a closure is a value
- Create a closure out of (i) the function and (ii) the current environment

To evaluate a function call…

Function calls

To evaluate `(exp1 exp2)`

- Evaluate `exp1` in the current environment to get a closure
- Evaluate `exp2` in the current environment to get a value
- Evaluate the closure's function's body *in the closure's environment* extended to map the function's argument name to the argument value
  - We only will implement single-argument functions
  - For recursion, a function name will evaluate to its entire closure

This is the same semantics we've been learning

Given a closure, the code part is only ever evaluated using the closure's environment part (extended with the argument binding), *not* the current environment at the call site.

Sounds expensive!

It isn't!!

- *Time* to build a closure is tiny: struct with two fields
- *Space* to store closures *might* be large if the environment is large
  - But environments are immutable, so lots of sharing is natural and correct

Possible HW challenge problem (extra credit): when creating a closure store a possibly smaller environment holding only function *free variables*, i.e., "global" variables used in a function but not bound in it

- Function body would never need anything else from the environment

Coming attractions

- Specific details of MUPL (interpreter assignment)
- Encoding MUPL programs as Racket structs and encoding MUPL environments

- Then mostly done with functional programming…

…but need to take out the garbage

- After that: Ruby and object-oriented programming, grammars, scanners, parsers, more implementation