CSE 341: Programming Languages

Dan Grossman
Spring 2008
Lecture 17—define-struct; Implementing higher-order functions
Data in Scheme

Recall ML’s approach to each-of, one-of, and self-referential types:

```
datatype t =
    Foo of int | Bar of int * int | Baz of string * t
```

Pure Scheme’s approach:

- There is One Big Datatype holding *every value*.
- Primitives implicitly raise errors for “wrong variant”
- Use pairs (lists) for each-of types
- Can also use for one-of types with explicit “tags”
  - Like our `force/delay` with a boolean field
  - Symbols better style
- Use helper functions like `caddr` (and/or define your own).
Dynamic typing

There is still good reason to have support for constructors:

- Make a foo that has fields x, y, z
- Test to see if you have a foo or not

But with dynamic typing:

- Constructors are not “grouped” into types (just added to the One Big Datatype)
- The fields can hold anything

Orthogonally: We don’t have pattern-matching.
**define-struct**

DrScheme extends Scheme with define-struct, e.g.:

```
(define-struct card (suit value))
```

Semantics: Introduce several new bindings...

- **constructor** `(make-card)` that takes arguments and make values (like `cons`)
- **predicate** `(card?)` that takes 1 argument, return `#t` only for values made from the right constructor (like `cons?`).
- **accessors** `(card-suit, card-value)` that take 1 argument, return a field, or call `error` for values not made from the right constructor (like `car` and `cdr`).
- **mutators** `(set-card-suit!, set-card-value!)` that are like accessors except they mutate field contents (like `set-car!` and `set-cdr!`).
Idiom for ML datatypes

Instead of a datatype with \( n \) constructors, you just use define-struct \( n \) times.

That “these \( n \) go together” is just convention.

Instead of case, you have a cond with \( n \) predicates and one “catch-all” error case.

For homework 5:

```scheme
;; a variable, e.g., (make-var "foo")
(define-struct var (string))

;; a constant number, e.g., (make-int 17)
(define-struct int (num))

(define-struct add (e1 e2)) ;; add two expressions
(define-struct ifgreater (e1 e2 e3 e4)) ;; etc.
...```
define-struct is special

define-struct creates a new variant for The One Big Datatype.

Claim: define-struct is not a function.

Claim: define-struct is not a macro.

It could be a macro except for one key bit of its semantics: Values built from the constructor cause every other predicate (including all built-in ones like pair?) to return #f.

Advantage: abstraction and bug-catchng (clients can’t “abuse” your things as though they were something else)

Disadvantage: Can’t write “generic” code that has a case for every possible variant in every Scheme program (like eval).
Implementing Languages

Mostly 341 is about language meaning, not “how can an implementation do that”, but it’s important to “dispel the magic”.

At super high-level, there are two ways to implement a language $A$:

- Write an interpreter in language $B$ that evaluates a program in $A$
  - Like we just saw for a little expression language
- Write a compiler in language $B$ that translates a program in $A$ to a program in language $C$ (and have an implementation of $C$)

In theory, this is just an implementation decision.

HW5: An interpreter for MUPL in Scheme.

Most interesting thing about MUPL: higher-order functions.
How is one language inside another?

How is:
(make-negate (make-add (make-const 2) (make-const 2)))
a “program” instead of
"- (2 + 2)"

Because parsing — turning a string/file into a tree of datatype-like
things is covered in CSE401.

These trees are called abstract-syntax trees (or ASTs).

They are ideal program representations for passing to an interpreter.

We can write them by hand, or write a parser, or write code that
produces them.
An interpreter

A “direct” language implementation is often just writing our evaluation rules for our language in another language.

- Languages with variables need interpreters with *environments*
- “eval-prog” takes an environment and an expression and returns a value (the subset of expressions that we define to be answers)
- An environment is just a mapping from variables to values (e.g., an association list)
- “eval-prog” uses recursion
  - Example: To evaluate an addition expression, evaluate the two subexpressions under the same environment, then...
- For homework 5, expressions & environments are all we need
  - Exceptions or mutation can require more inputs/outputs to “eval-prog”
Implementing Higher-Order Functions

The magic: How is the “right environment” around for lexical scope (the environment from when the function was defined)?

Lack of magic: Implementation keeps it around!

Interpreter:

• The interpreter has a “current environment”

• To evaluate a function (expression), create a closure (value), a pair of the function and the “current environment”.

• Application will now apply a closure to an argument: Interpret function body, but instead of using “current environment”, use closure’s environment extended with the argument.

Note: This is directly implementing the semantics from week 3.
Is that expensive?

Building a closure is easy; you already have the environment.

Since environments are immutable, it’s easy to share them.

Still, a given closure doesn’t need most of the environment, so for space efficiency it can be worth it to make a new smaller environment holding only the function’s free variables.

- That is, an approximation of the things a call to the function might look up.

- Challenge problem in homework 5
Compiling Higher-Order Functions

The key to the interpreter approach: The interpreter has an explicit environment and can “change” it to implement lexical scope.

We can also compile higher-order functions to a language without higher-order functions:
Instead of an implicit environment, we pass an explicit environment to every function.

• As with interpreter, we build a closure to evaluate functions.
• But all functions now take one extra argument.
• Application passes a closure’s code its own environment for the extra argument.
• Evaluating variables uses this extra argument.
  – Compiler translates them to environment-reads.

Plus: Data-structure optimizations so variable-lookup is $O(1)$