

ICS/CSE 141: Programming Languages

Functional vs. Imperative, Macros, Denotational Semantics

School of Information and Computer Science
University of California, Irvine

Chris Stork
cstork@ics.uci.edu

Overview

1. Functional vs. Imperative Programming
2. Macros
3. Denotational Semantics

Functional vs. Imperative Programming

Parts of Speech

Statement

```
load 4094 r1
```

- Imperative command
- Alters the contents of previously-accessible memory

Expression

```
( x+5 ) / 2
```

- Syntactic entity that is evaluated
- Has a value, need not change accessible memory
- If it does, has a side effect

Declaration

```
integer x
```

- Introduces new identifier
- May bind value to identifier, specify type, etc.

Pure vs. impure

A **purely functional** language has no side effects, i.e. **evaluating an expression does not change the observable state of the machine.**

Side-effects in **impure** Scheme:

- I/O: `write`, `display`, `newline`
- Mutators: `set!`, `set-car!`, `set-cdr!` (note exclamation mark)

Mutation and Sharing in Scheme

```
> (define s
  ((lambda (x) (cons x x)) (cons 'a 'b)))
> (set-car! (car s) 5)
> s
((5 . b) 5 . b)
> (define u
  (cons (cons 'a 'b) (cons 'a 'b)))
> (set-car! (car u) 5)
> u
((5 . b) a . b)
```

Note that in purely functional programs the programmer does not need to be aware of sharing issues (except if she's concerned with resource usage).

Scheme's Equivalence Predicates

Equivalence relation: symmetric, reflexive, transitive

In a purely functional world only `equal?` would be needed.

Everything that “**looks equal**” is equal:

```
(equal? (cons 1 2) (cons 1 2)) ; coarse
```

But in reality we have equality by **identity**:

```
(eq? (cons 1 2) (cons 1 2)) ; finest
```

And what's used most of the time is **equivalence**:

```
(equiv? (cons 1 2) (cons 1 2))  
(equiv? 1 1)  
(equiv? 1000000000 1000000000)
```

Functional vs. Imperative Programs

Functional languages try to facilitate the pure, declarative style!

Declarative Language Test:

Within the scope of specific declarations of x_1, \dots, x_n , all occurrences of an expression e containing only variables x_1, \dots, x_n have the same value.

Implications: Readability, Optimizations

Referential Transparency

A name or noun phrase is **referentially transparent** if it may be replaced with another phrase with the same referent **without** changing the meaning of the sentence.

Examples:

- “I saw Walter get into his {car|Maserati}.”
- “He was called William {Rufus|IV} because of his red beard.”

... but what this means for PLs is subject to interpretation

Example: Factorial

C

```
/* Good old C */
int fac(int n)
{
    int tmp;

    for (tmp=1; n>0; n--)
    {
        tmp *= n;
    }
    return tmp;
}
```

Scheme

```
; Scheme
(define fac
  (lambda (n)
    (if (<= n 0)
        1
        (* n (fac (- n 1))))))
```

ML

```
(* ML *)
fun fac(n) =
  if n <= 0 then 1 else n*fac(n-1)
```

Macros

Background

There are many macro processors:

- m4
- T_EX
- CPP
- many languages mostly for web design
- type “macro processor” into Google!
- Scheme

Whole systems are based on macros:

- L^AT_EX
- [Lightning JIT library](#)

C Macros I

Macro processors have a bad reputation due to C's macros:

```
#define swap(x,y) \  
    tmp = x;      \  
    x = y;        \  
    y = tmp;
```

Problems:

C Macros I

Macro processors have a bad reputation due to C's macros:

```
#define swap(x,y) \  
    tmp = x;      \  
    x = y;        \  
    y = tmp;
```

Problems: Type of `tmp`? Scope?

C Macros II

```
#define swap(x,y) \  
do {              \  
    int tmp;      \  
    tmp = x;      \  
    x = y;        \  
    y = tmp;      \  
} while (0);
```

Problems?

C Macros II

```
#define swap(x,y) \  
do {              \  
    int tmp;      \  
    tmp = x;      \  
    x = y;        \  
    y = tmp;      \  
} while (0);
```

Problems?

swap(a, tmp) Polluted scopes/namespaces!

Scheme Macros

```
(define-syntax swap
  (syntax-rules ()
    ((swap x y) (let ((tmp x))
                  (set! x y)
                  (set! y tmp))))))
```

Scheme macros are **hygienic**:

```
(swap a tmp)
```

expands to

```
(let ((<unique-sym> x))
  (set! x y)
  (set! y <unique-sym>))
```

Hygienic Macros

Hygiene:

If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers.

Referential Transparency:

If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

What's this?

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
      tag)
      val ...))))
```

Quintessence

Macros done right are a great tool!

Can be used to extend the syntax of a block-structured language in a consistent and reliable manner.

Lisp/Scheme provide the perfect infrastructure for web page generation, but few people realize that.

- Read Paul Graham's [Beating the Average!](#)
- Look at DrScheme's web server framework!

Denotational Semantics

Historical Background

Developed in the 1960s/1970s, mainly by Strachey and Scott

- Find a **denotation** for each (part of) a program
 - expression, instruction, declaration
- Programs **denote** functions that explain their **meaning**
 - Functions from states to states (partial!)
 - States are functions from variables to values (total!)
- **Compositionality**: Meaning of whole in terms of meaning of parts
 - $\llbracket \text{if } X \text{ then } Y \text{ else } Z \rrbracket = f(\llbracket X \rrbracket, \llbracket Y \rrbracket, \llbracket Z \rrbracket) =$
 $\llbracket \text{if } A \text{ then } B \text{ else } C \rrbracket, \text{ iff } \llbracket X \rrbracket = \llbracket A \rrbracket, \llbracket Y \rrbracket = \llbracket B \rrbracket, \llbracket Z \rrbracket = \llbracket C \rrbracket$

Binary Numbers

Syntax: $n ::= nb|b$ $b ::= 0|1$

- so 0, 1, 101, and 101001110001101011011110, are binary numbers

What is the **meaning** of a single **digit**?

- The digit 0 denotes the value 0, digit 1 denotes 1 (meta vs object language)
- $\mathcal{E}[[0]] = 0$, $\mathcal{E}[[1]] = 1$

What is the **meaning** of a longer **number**?

- If we have a number n and we “add” another digit b , n will be “doubled”
- $\mathcal{E}[[nb]] = \mathcal{E}[[n]] \times 2 + \mathcal{E}[[b]]$

Example: $\mathcal{E}[[101]] = \mathcal{E}[[10]] \times 2 + \mathcal{E}[[1]] = (\mathcal{E}[[1]] \times 2 + \mathcal{E}[[0]]) \times 2 + \mathcal{E}[[1]]$
 $= (1 \times 2 + 0) \times 2 + 1 = 5$

Simple Expressions

Syntax: $e ::= n | e + e | e - e$ $n ::= nb | b$ $b ::= 0 | 1$

- so $0 + 0$, $1 + 1$, and $101 + 010$, are expressions

What is the **meaning** of an **expression**?

- The **sum** or **difference** of the meanings of **component** expressions
- $\mathcal{E}[[e_1 + e_2]] = \mathcal{E}[[e_1]] + \mathcal{E}[[e_2]]$ (different plus signs!)
- $\mathcal{E}[[e_1 - e_2]] = \mathcal{E}[[e_1]] - \mathcal{E}[[e_2]]$

Example: $\mathcal{E}[[101 + 010]] = \mathcal{E}[[101]] + \mathcal{E}[[010]] = \dots = 5 + \mathcal{E}[[010]]$
 $= \dots = 5 + 2 = 7$

Variables

Syntax:

$$e ::= v | n | e + e | e - e \quad n ::= nb | b \quad b ::= 0 | 1 \quad v ::= x | y | z | \dots$$

- so x is supposed to be a **value** that depends on the current **state**
- State: Variable \rightarrow Value, e.g. $s(x) = 10$ means x has the value 10

The **meaning** of **expressions** now depends on **state**

- $\mathcal{E}[[e]](s)$ stands for “value of expression e in state s ”
- new rules for variables: $\mathcal{E}[[x]](s) = s(x)$, $\mathcal{E}[[y]](s) = s(y)$, ...
- other rules basically unchanged: $\mathcal{E}[[0]](s) = 0$,
 $\mathcal{E}[[e_1 + e_2]](s) = \mathcal{E}[[e_1]](s) + \mathcal{E}[[e_2]](s)$, ...

The While Language

We consider a **simple** imperative programming language

- Syntax:
 $c ::= x := e \mid c; c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c$
- Assume that **booleans** and **comparisons** have been added...
- ... that x and e have appropriate type.
- This language is **turing-complete** just like the λ -calculus!

Example program (sums numbers from 0 to z into y):

```
x := 0;
y := 0;
while x <= z do
  ( y := y + x; x := x + 1 )
```

States & Meaning

State : Variables \rightarrow Values

(Idealized: ignore memory locations & unlimited variables)

Meaning-of-a-Program : State \rightarrow State

In our case:

Command : State \rightarrow State

This is a partial function, of course.

Assignments

Modeling assignments requires “changing” the state somehow

- Initial state assumed given, e.g. all variables are 0 or `false`
- To generate a “new” state, use **modify** function

- $$\text{modify}(s, x, a)(v) = \begin{cases} a & \text{if } v = x \\ s(v) & \text{otherwise} \end{cases}$$

or, more fancy,

$$\text{modify}(s, x, a) = \lambda v \in \text{Variables} . \text{if } v = x \text{ then } a \text{ else } s(v)$$

- “the state $\text{modify}(s, x, a)$ is like s except that x now has value a ”. (We just compose functions to obtain the effect.)

A rule for assignment commands

- $\mathcal{C}[\mathbf{x} := e](s) = \text{modify}(s, x, \mathcal{E}[e](s))$

Example: Given $s_0(x) = 0$ initial state

- $\mathcal{C}[\mathbf{x} := 10](s) = \text{modify}(s, x, \mathcal{E}[10](s)) = \text{modify}(s, x, 10) = s_1$
- Now $s_1(x) = 2$ as we would expect...

Sequencing

A **sequence** of commands requires **ordering** state transitions

- $\mathcal{C}[[c1; c2]](s) = \mathcal{C}[[c2]](\mathcal{C}[[c1]](s))$
- from s_0 command $c1$ leads to s_1
- from s_1 command $c2$ leads to s_2

Example: Given $s_0(x) = 0$ initial state

- $\begin{aligned} &\mathcal{C}[[x := 10; x := x + 1]](s_0) \\ &= \mathcal{C}[[x := x + 1]](\mathcal{C}[[x := 10]](s_0)) \\ &= \mathcal{C}[[x := x + 1]](\text{modify}(s_0, x, \mathcal{E}[[10]](s_0))) \\ &= \mathcal{C}[[x := x + 1]](\text{modify}(s_0, x, 10)) = \mathcal{C}[[x := x + 1]](s_1) \\ &= \text{modify}(s_1, x, \mathcal{E}[[x + 1]](s_1)) \\ &= \text{modify}(s_1, x, \mathcal{E}[[x]](s_1) + \mathcal{E}[[1]](s_1)) \\ &= \text{modify}(s_1, x, 10 + 1) \\ &= \text{modify}(s_1, x, 11) \\ &= s_2 \end{aligned}$

Infinite Loops

Denotational Semantics unambiguously associates a partial function from states to states with each program.

$$\mathcal{C}[\text{while } x=x \text{ do } x := x](s)$$

does not terminate for any state. Therefore it is not defined in any state.

Final Remarks

Often λ -**calculus** is used as **meta-language**.

Thereby, denotational semantics provide a way to **translate imperative programs** into **functional programs**.

Thank You!