



EECS 490 – Lecture 6

Functions and Introduction to Scheme

1

Announcements

- ▶ Project 1 due today at 8pm
- ▶ Homework 2 due Friday 9/29 at 8pm
- ▶ Project 2 due Friday 10/6 at 8pm

Scheme Syntax

- From R5RS spec:

$\langle \text{datum} \rangle \rightarrow \langle \text{simple datum} \rangle \mid \langle \text{compound datum} \rangle$

$\langle \text{compound datum} \rangle \rightarrow \langle \text{list} \rangle \mid \langle \text{vector} \rangle$

$\langle \text{list} \rangle \rightarrow (\langle \text{datum} \rangle^*) \mid (\langle \text{datum} \rangle^+ . \langle \text{datum} \rangle) \mid \langle \text{abbreviation} \rangle$

$\langle \text{abbreviation} \rangle \rightarrow \langle \text{abbrev prefix} \rangle \langle \text{datum} \rangle$

$\langle \text{abbrev prefix} \rangle \rightarrow ' \mid ' \mid , \mid , @$

- List can be

- Zero or more datums in parentheses
- Parentheses containing one or more datums, a period, and a single datum
- A quotation character followed by a datum

Scheme Syntax

- From R5RS spec:

$\langle \text{datum} \rangle \rightarrow \langle \text{simple datum} \rangle \mid \langle \text{compound datum} \rangle$

$\langle \text{compound datum} \rangle \rightarrow \langle \text{list} \rangle \mid \langle \text{vector} \rangle$

$\langle \text{list} \rangle \rightarrow (\langle \text{datum} \rangle^*) \mid (\langle \text{datum} \rangle^+ . \langle \text{datum} \rangle) \mid \langle \text{abbreviation} \rangle$

$\langle \text{abbreviation} \rangle \rightarrow \langle \text{abbrev prefix} \rangle \langle \text{datum} \rangle$

$\langle \text{abbrev prefix} \rangle \rightarrow ' \mid ' \mid , \mid , @$

Vexing Parse

- In languages with complex syntax, such as C++, ambiguity cannot be avoided in the grammar
 - External rules are specified to disambiguate fragments

```
struct foo {
    foo() {
        cout << "foo::foo()" << endl;
    }
    foo(int x) {
        cout << "foo::foo(" << x << ")" << endl;
    }
    void operator=(int x) {
        cout << "foo::operator=(" << x << ")" << endl;
    }
};
```

```
int a = 3, b = 4;
```

```
int main() {
    foo(a);           // equivalent to foo a;
    foo(b) = 3;       // equivalent to foo b = 3;
}
```

C++ disambiguates in favor of declarations

Names can be parenthesized in declarations

```
foo::foo()
foo::foo(3)
```

Most Vexing Parse

- A most vexing example:

```
struct bar {
    bar(foo f) {
        cout << "bar::bar(foo)" << endl;
    }
};
```

Nothing
printed

C++ disambiguates in favor
of function declarations

```
bar c(foo()); // equivalent to bar c(foo);
```

- Clang warning:

foo.cpp:28:8: warning: parentheses were disambiguated as a function declaration

[-Wvexing-parse]

```
bar c(foo()); // equivalent to bar c(foo);
```

~~~~~

**foo.cpp:28:9: note: add a pair of parentheses to declare a variable**

```
bar c(foo()); // equivalent to bar c(foo);
```

^

( )

# Agenda

- Keyword and Default Arguments
- Variadic Functions
- Parameter Passing
- Introduction to Scheme

# Keyword Arguments

- In most languages, names are not specified for arguments when calling a function

- Arguments are bound to parameters in order

```
void foo(int x, int y);  
foo(3, 4);
```

- Some languages allow arguments to be passed to specific parameters, allowing them to be given in a different order and serving as documentation

```
def foo(x, y):  
    print(x, y)
```

```
>>> foo(y = 3, x = 4)  
4 3
```



# Arguments in Swift

- Swift and Objective-C require argument names for most arguments, as well as that they are passed in the same order as the parameters

```
func greet(name: String, withGreeting: String) {  
    print(withGreeting + " " + name)  
}  
greet(name: "world", withGreeting: "hello")
```

- Functions can specify separate internal and external names for a parameter
- Argument names used in function-overload resolution

```
func foo(a: Int) { ... }  
func foo(b: Int) { ... }  
foo(a: 3)
```

# Default Arguments

- Some languages allow a function definition or declaration to provide a default argument for a parameter
- Allow a function to be called without an argument value for the parameter

```
void foo(int x, int y = 0);  
foo(3); // equivalent to foo(3, 0)  
foo(3, 4);
```

- Parameters with default arguments generally have to be at the end of the parameter list
- Evaluation rules
  - Evaluated in definition environment in most languages
  - Most languages evaluate default argument each time the function is called

# Python Default Arguments

- Python differs from most languages in that the default argument is evaluated only once at definition time

```
def foo(x, y = []):  
    y.append(x)  
    print(y)
```

```
>>> foo(3)  
[3]  
>>> foo(4)  
[3, 4]
```

# C/C++ Default Arguments

- Default arguments can be provided in any declaration of a function, including its definition
- Multiple visible declarations may not provide a default argument for the same parameter, even if it is the same
- The set of default arguments is the union of all visible declarations in the same scope

```
int foo(int x, int y = 4);  
int foo(int x = 3, int y) {  
    return x + y;  
}
```

- C++ templates also can have default arguments

# Overloading as Alternative

- Some languages, such as Java, rely on function overloading to provide the same behavior as default arguments

```
static void foo(int x, int y) {  
    System.out.println(x + y);  
}
```

```
static void foo(int x) {  
    foo(x, 0);  
}
```



"Default"  
argument of 0

# Variadic Functions

- Functions that can be called with a variable number of arguments, also referred to as *varargs*
- Arguments often packed into a container such as a tuple or array
- Arguments may be required to be of the same type, or can be of different types
- Example in Java:

```
static void print_all(String... args) {  
    for (String s : args) {  
        System.out.println(s);  
    }  
}
```

All Strings,  
packaged  
into array

```
print_all("hello", "world");
```

Java also allows an array to be passed into a variadic parameter.

# Varargs in Python

- Python allows both variadic simple arguments as well as keyword arguments
- Simple variadic arguments packaged into tuple
- Variadic keyword arguments packaged into `dict`

```
def print_args(*args, **kwargs):  
    print(args)  
    print(kwargs)
```

```
>>> print_args(3, 4, x = 5, y = 6)  
(3, 4)  
{'x': 5, 'y': 6}
```

## Unpacking Sequences and Dictionaries

- Python has operators for unpacking sequences and dictionaries
- Can be used where a value list is required

```
>>> print_args(*[3, 4], **{ 'x': 5, 'y': 6})  
(3, 4)  
{'x': 5, 'y': 6}
```

Unpacks  
sequence

Unpacks  
dictionary



# Varargs in C/C++

- C and C++ provide a varargs mechanism that is low level and can be unsafe

```
#include <stdarg.h>
int sum(int count, ...) {
    va_list args;
    int total = 0;
    int i;
    va_start(args, count);
    for (i = 0; i < count; i++) {
        total += va_arg(args, int);
    }
    va_end(args);
    return total;
}
```

Relies on caller to  
pass correct count

Relies on caller to  
pass right types

# Parameter Passing

- Arguments and parameters are a means of communication between a function and its caller
- A parameter may be used only for input, only for output, or for both
- Semantics of parameters determined by *call mode* of function
  - Call by value
  - Call by reference
  - Call by result
  - Call by value-result
  - Call by name

# Call by Value

- ▶ A parameter represents a new variable in the frame of a function invocation
- ▶ Argument value is copied to parameter variable
- ▶ Parameter can only be used for input

```
void foo(int x) {  
    x++;  
    cout << x << endl;  
}
```

```
int y = 3;  
foo(y);           // prints 4  
cout << y << endl; // prints 3
```

# Call by Reference

- Requires l-value as argument<sup>1</sup>
- Parameter name is bound to argument object
- Parameter can be used for input and output
- No separate storage for parameter

```
void swap(int &x, int &y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

```
int x = 3, y = 4;  
swap(x, y);           // x now 4, y now 3
```

<sup>1</sup>In C++, const l-value references can bind to r-values

# Simulating Call by Reference

- Pointers can be used to simulate call by reference
- However, function is still call by value, since parameters correspond to new pointer variables

```
void swap(int *x, int *y) {  
    int tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

```
int x = 3, y = 4;  
swap(&x, &y);      // x now 4, y now 3
```

# Call by Result

- Argument must be l-value
- Parameter is a new variable with its own storage
- Parameter is **not** initialized with argument value
- Upon return of the function, parameter value is copied to argument object
- Can only be used for output

```
void foo(result int x) {  
    x = 3;  
    ...  
    x++;    // x is now 4  
}
```

```
int y = 5;  
foo(y);    // y is now 4
```

This is not C++! C++ does not have call by result.

# Call by Value-Result

- Combination of call by value and call by result
- Argument must be l-value
- Parameter is a new variable with storage, initialized with argument value
- Upon return, value of parameter is copied to argument object

```
int foo(v/r int x, v/r int y) {  
    x++;  
    return x - y;  
}
```

```
int z = 3;  
print(foo(z, z));    // prints 1
```

Again, not C++! Final value of *z* depends on whether it is copied from first or second parameter in the given language

# Call by Name

- ▶ Any expression provided as argument
- ▶ Parameter name is replaced by argument expression everywhere in the body
- ▶ Expression computed whenever it is encountered in body

```
void foo(name int x) {  
    print(x);  // becomes print(++y)  
    print(x);  // becomes print(++y)  
}
```

```
int y = 3;  
foo(++y);  // prints 4, then 5; y is now 5
```

!C++; Mutating expressions should not be passed by name, since behavior would depend on implementation details



# Thunks

- In call by name, expression must be computed in its own environment

```
void bar(name int x) {  
    int y = 3;  
    print(x + y); // becomes print(y + 1 + y)  
}
```

```
int y = 1;  
bar(y + 1); // should print 5, not 7
```

- This is accomplished with a *thunk*, a compiler-generated local function that packages the expression with its environment

# Python is Call by Value

- Call by value is most common mode, followed by call by reference
- Python and Java are not call by reference
  - They combine call by value with reference semantics
  - This is sometimes called "call by object reference"

```
def swap(x, y):  
    tmp = x  
    x = y  
    y = tmp
```

```
>>> x, y = 1, 2  
>>> swap(x, y)  
>>> x, y  
(1, 2)
```

x and y are new  
variables with their  
own storage

- ▶ We'll start again in five minutes.

# Running Scheme

- We recommend Racket
  - <https://download.racket-lang.org/>
  - Includes DrRacket IDE and command-line `plt-r5rs` interpreter
- Online interpreter for simple examples
  - <https://repl.it/languages/scheme>
- Be aware that most interpreters are not fully R5RS compliant, so we recommend sticking to Racket for homework/project development

# Call Expressions

- Everything is an expression in Scheme
- Simple expressions: literals, names
- Compound expressions consist of a parenthesized list

- Call expressions:

`(function arg1 arg2 ... argN)`

- Examples:

`(+ 3 4)`

`(+ (* 3 5) (- 10 6))`

`(quotient 10 2)`

Integer division

Order of evaluation of subexpressions is not defined

9/21/17

# Conditionals

- Special forms have their own evaluation rules
- Conditional evaluates *test*, then evaluates *then* expression if true, otherwise the *else* expression if provided

`(if <test> <then_expr> <else_expr>)`

- Value of whole expression is value of then or else expression
  - If test is false and no else expression, then value is unspecified
- Only `#f` is a false value, all other values are true

# Definitions and Blocks

- Variables can be defined in the current frame using `define`

```
(define <name> <expr>)
```

- In standard Scheme, this can only be at the top level or at the beginning of a block
  - We won't require this to be enforced in the project

- Blocks can be introduced with `let`

```
(let ((<name1> <expr1>) ... (<nameN> <exprN>))  
  <body_expr1> <body_expr2> ... <body_exprN>)
```

`let` can be considered syntactic sugar for `lambda` definition and application.

# Functions

- Functions can also be defined using `define`

```
(define (<name> <param1> ... <paramN>)  
  <body_expr1> ... <body_exprN>)
```

- Anonymous functions can be defined using `lambda`

```
(lambda (<param1> ... <paramN>)  
  <body_expr1> ... <body_exprN>)
```

- Then the `define` form is equivalent to

```
(define (lambda (<param1> ... <paramN>)  
  <body_expr1> ... <body_exprN>)  
  <name>)
```



# Pairs

- Pairs are a fundamental mechanism for combining data

- Construct pair using cons

```
> (define x (cons 1 2))
```

```
> x
```

```
(1 . 2)
```

Dot denotes pair where  
the second is not a list

- Access the first and second with car and cdr

```
> (car x)
```

```
1
```

```
> (cdr x)
```

```
2
```

# Lists

- A list is a sequence of pairs terminated by an empty list
- An empty list is denoted by '()', and in our implementation, by the non-standard `nil`

```
> (define y (cons 1 (cons 2 (cons 3 '()))))  
> y  
(1 2 3)  
> (car y)  
1  
> (cdr y)  
(2 3)  
> (cdr (cdr (cdr y)))  
()
```

Also `(cdddr y)`  
in standard  
Scheme

# Symbolic Data

- ▶ In Scheme, both code and data share the same representation
- ▶ Quotation specifies that what follows should be treated as data and not evaluated

```
> (define x 3)
```

```
> x
```

```
3
```

```
> 'x
```

```
x
```

```
> '(hello world)
```

```
(hello world)
```

Equivalent to  
(quote x)

