# CSC347 Concepts of Programming Languages

## Lecture Notes: Scheme

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

## Functional Programming: Programming with Pure Functions and Immutable Data

In this lecture, we explore the following questions:

❓ How is a functional programming language built entirely from expressions?

### Learning Goals

🎓 Understand basic Scheme: expressions, cons cells, lists, functions, and recursion

## Scheme

We start making our way into functional programming by exploring how functional programs are built entirely from (pure) expressions and operate primarily with immutable data. These concepts were introduced in the late 1950s, early 1960s first with Lisp and got later advanced into Scheme and the many dialects (e.g., Common Lisp, Scheme, Clojure, Racket). The original motivation behind the design of these languages was the ability to implement logic reasoning and AI applications (think knowledge representation and reasoning, not machine learning). These languages pioneered a plethora of programming language concepts:

- garbage collection
- first-class, higher-order, nested functions
- read-eval-print loop (REPL) including runtime compilation
- macro system
- multiple dispatch and multi-methods

In this course, we will use Scheme to illustrate the language concepts. To give you a bit of a feel of the language, below is a snippet of code that calculates the length of a linked list recursively.

```
1 ; calculates the length of a linked list recursively
2 (define (length l)
3     (if (equal? l ())
4     0
5     (+ 1 (length (cdr l)))))
```

The designers of Scheme opted for fully-parenthesized expressions in prefix notation, instead of defining operator precedence rules. The benefit is that parsing becomes extremely easy; the downside is that we have to read and write lots of silly parentheses.

**Expressions.** Expressions in Scheme are built from atoms: number literals `5`, string literals `"hello world"`, and symbols `'helloworld`. Arithmetic expressions use prefix notation, must be parenthesized, and are n-ary evaluated in the listed order.

```
(+ 10 5 2) ; result: 17
(- 10 5 2) ; result: 3
(* 10 5 2) ; result: 100
(/ 10 5 2) ; result: 1
```

Writing expressions in prefix notation and fully parenthesized simplifies parsing. In prefix notation, the operator is listed *before* the operands (`+ 1 2`). As a result, at the time the parser reads operands their expected type is already known since the operator came first. In infix notation, the operator is listed *between* the operands (`1 + 2`), so that the parser first reads an operand without knowing what type is expected. In postfix notation, the operator comes *after* the operands (`1 2 +`), so all operands have to be parsed without knowing expected types.

Functions in Scheme are defined in expressions: `(define (square n)(* n n))`. The operator `define` takes as first argument a list of function name and argument names, followed by the function definition (a sequence of expressions). To invoke a function, we simply write an expression consisting of the function name followed by argument expressions: `(square 5)`. We can invoke functions multiple times: `(square (square 5))`. The general form of function definition is below.

```
(define (f param_1 param_2 ... param_m) (e_1 e_2 ... e_n))
```

The function `f` takes $m$ arguments. Its definition is a sequence of $n$ expressions, where the first $n-1$ expressions are evaluated for their side effect and expression `e_n` determines the result of the function. No `return` keyword is necessary and no statements, Scheme only uses expressions. Optionally, we can use the keyword `begin` to mark the sequence of expressions: `(define (f param_1 param_2 ... param_m)(begin e_1 e_2 ... e_n))`. A function call expression `(f M N)` is evaluated by first evaluating expression $M$ to value $U$, then evaluating expression $N$ to value $V$, and then invoking function $f$ with values $U$ and $V$. Note that the parentheses are mandatory! The operator `define` is a special form, not a function, so this convention does not apply.

Boolean and conditional expressions include the literals `#t` and `#f`, the operator `=` to test numeric equality of numbers `(define (isZero n)(= n 0))`. Conditional expression `if` is a non-strict special form.

```
(define (safe-divide m n)
  (if (= n 0)
    "divide by zero"
    (/ m n)))
```

Remember that non-strict evaluation means that arguments are evaluated on demand, in this case the Boolean condition and, depending on its outcome, one of the branches. Recursive functions are common in Scheme, and in fact throughout functional programming. Below, we implement computing the factorial of a number $n$ in Scheme using a conditional expression.

```
(define (fact n)
  (if (<= n 1)
```

```
3       1
4       (* n (fact (- n 1)))))
```

Compare this to the C factorial using conditional expressions.

```
1  int fact(int n) {
2      return (n <= 1) ? 1 : n * fact(n-1);
3  }
```

We notice some differences, most obviously prefix vs. infix notation and a more readable conditional expression in Scheme. Another obvious difference is that in C we need to annotate types, while in Scheme types are *inferred* (more about this in a later lecture).

Lambda expressions (named after Church's lambda calculus) provide binders to define "anonymous" functions. For example, `(lambda (x)(+ x 1))` defines a function that increments the value of its argument. Lambda expressions are useful as arguments to higher-order functions, i.e., functions that take other functions as arguments (compare to function pointers in C).

**Cons Cells.** The primary means of defining data types in Scheme is by combining two pieces of data into a pair called a *cons cell*. We can define homogeneous cons cells, such as pairs of numbers `(cons 1 2)` or pairs of strings `(cons "hello""world)`, and heterogeneous cons cells, such as a pair of a number and a string `(cons 1 "world")`. We use function `car` to extract the first component of a cons cell, and `cdr` to extract its second component.

```
1  (car (cons 1 "world")) ; result: 1
2  (cdr (cons 1 "world")) ; result: "world"
```

The most important use of cons cells is as pairs holding some data plus pointers to other data (`car` position for elements, `cdr` position for next cons cell). With this use, we can build container data structures, such as linked lists and trees. Linked lists can be built using `'()` for the empty list and `cons` to create pairs.

```
1  ()                                          ; the empty list
2  (cons 1 ())                                 ; the list 1
3  (cons 1 (cons 2 (cons 3 (cons 4 '()))))     ; the list 1,2,3,4
```

This notation is quite lengthy, so Scheme knows syntactic sugar to build lists in a less verbose way:

```
1  (quote (1 2 3)) ; special form quote prevents evaluation
2  '(1 2 3)        ; shorthand for quote
3  (list 1 2 3)    ; evaluates args and puts result in a list
4  (list 1 2 (+ 1 2))
```

In order to define algorithms on lists, equality tests are quite handy. Scheme distinguishes between pointer equality `eq?` and structural equality `equal?`. Pointer equality compares two pointers and only returns true when the pointers point to the exact same address, while structural equality traverses two structures and compares the content elementwise.

```
1  (eq? (cons 1 2) '(1 2))          ; returns #f
2  (equal? (list 1 2 3 ) '(1 2 3)) ; returns #t
```

Now that we have the building blocks for defining algorithms on data structures, let's examine a small example of computing the length of a linked list recursively. As usual in a recursive function, we start by identifying the base case (i.e., the smallest amount of work no longer decomposable into pieces) and the recursive case (i.e., how to work on one element and combine it with the rest of the work).

```
(define (length l)
  (if (equal? l '())
    ; base case: empty list has length 0
    0
    ; recursive case: 1 + length of rest
    (+ 1 (length (cdr l)))))
```

A sample evaluation of this function with its recursive calls and intermediate results is illustrated below.

```
(length '(5 6 7))
--> (if (equal? '(5 6 7) ()) 0 (+ 1 (length (cdr '(5 6 7)))))
--> (+ 1 (length (cdr '(5 6 7))))
--> (+ 1 (length '(6 7)))
--> (+ 1 (+ 1 (length '(7))))
--> (+ 1 (+ 1 (+ 1 (length '()))))
--> (+ 1 (+ 1 (+ 1 0)))
--> (+ 1 (+ 1 1))
--> (+ 1 2)
--> 3
```

**Dynamic Types and S-Expressions.** Scheme uses dynamic types and type inference. It provides test operators that allow us to inspect the type of an element.

```
(symbol? 'x)
(number? 1)
(boolean? #t)
(string? "x")
(procedure? (lambda (x) (+ x 1)))
```

The major building blocks of Scheme programs are *symbolic expression* (S-expressions). S-expressions include pairs, numbers, Booleans, strings, symbols, and the empty list. Parsing a Scheme program results in an S-expression, which is then sent to a runtime evaluation procedure called `eval` for evaluation. The special form `quote` causes an expression to be parsed without evaluation.

```
(+ 1 2)          ; 3
'(+ 1 2)         ; the list (S-expression) +,1,2
(cons '+ '(1 2)) ; a pair of a + symbol and the list 1,2
(car '(+ 1 2))   ; + (the first element of the list +,1,2)
```

Within Scheme, we have access to `eval` and can explicitly call it on S-expressions for runtime evaluation.

```
(eval (cons '+ '(1 2)))
```

```
(define (add-all l) (eval (append '(+) l)))
(add-all '(1 2 3))
```

This is especially useful together with `read` that reads S-expressions from the console. Together, these functions allow us to implement interactive applications using a read-eval-print loop (REPL).