# CS450 - Structure of Higher Level Languages

Introduction

September 14, 2020

# Contact Information

- Instructor: Nurit Haspel
- `http://www.cs.umb.edu/~nurith`
- **email:** nurit.haspel@umb.edu
- Course webpage: `http://www.cs.umb.edu/cs450/`
- Office hours – Tu Th 12:30-2:00 on the following link: `https://us.bbcollab.com/guest/c264634eb35b46818b823d825eb2393a`.
- To access the classroom on Blackboard, go to "Blackboard Collaborate Ultra" on the left.
- Go into the course room (on top).
- **Course schedule:** Mo We 5:30-6:45 On Blackboard
- Lectures will be live and recorded.

# Course Description

- The syntax and semantics of higher-level languages are treated. Formal specifications of syntax and models of semantics will be used.
- **Topics:** mechanisms for parameter passing, scoping, dynamic storage allocation and systems interfacing. Both compiled and interpreted languages will be used as examples.
- The language of instruction is Scheme, a dialect of LISP, which is taught in the course.
- See the course Syllabus at `http://www.cs.umb.edu/cs450/Syllabus.pdf`
- Prerequisites are CS310 (Advanced Data Structures and Algorithms) and CS220/Math320 (Applied Discrete Mathematics) or permission from the instructor.

# Administrative Stuff

- A homework assignment will be given every 1–2 weeks (80–90% of the grade).
- There will be 1-2 online quizzes (10–20% of the grade).
- No final exam. Therefore, I will be **very** strict with academic dishonesty.
- I will use a plagiarism detection software and you will get an F if you get caught (and you will get caught).
- I will use Piazza to communicate with the class.
- Homework should be submitted through Gradescope.

# Checklist

- Download Dr. Racket: `racket-lang.org`.
- I added you to Piazza.
- Register to Gradescope with your UMB e-mail. Code was (or will be) provided on Piazza
- Make sure you can access Blackboard and know how to log in to the class space.
- Make sure you check your UMB e-mail regularly.

# Scheme

- *Scheme* is a dialect of Lisp (**Lis**t **P**rocessing).
- It is the second oldest programming language, after Fortran.
- Amazingly enough, however, both are still used:

  Fortran: used for scientific applications, number crunching, massive amounts of data. Compilers for Fortran can generate extremely efficient code.

  Lisp: used for academic research into programming language design, computational models, prototyping, AI research, natural language studies, ....

# Scheme

- Languages such as Fortran, Algol, Pascal, C, C++, Java, Perl, Python are examples of *imperative* programming languages.
- They are based on instructions.
- Lisp has some imperative constructs, but its soul is *functional*.
- In particular, it developed originally from a computing model called the *lambda calculus*, which was invented in 1936 by Alonzo Church to help in investigating what functions could be regarded as computable.

# Church-Turing Hypothesis

Those functions which are expressible in lambda calculus (which turn out to be the same as all those functions which are computable by Turing machines) constitute exactly the class of functions which we naturally regard as *computable*.

- To program in a functional language such as Scheme, you really have to forget everything you think you know about programming in Java, or C++, or Python, and start from the beginning.
- As we get deeper into Scheme, we will see how it can model different kinds of languages for us.

# Some First Things to Know About Lisp

- Lisp is typically interpreted rather than compiled.
- Parentheses always have meaning in Lisp, and it is *not* the meaning you are used to.
- Identifiers (e.g., symbols) are lexically more diverse than in most programming languages. For instance
  - *a is a symbol.
  - * a is two symbols (* followed by a).
  - $+$ is pre-defined to be the addition operator. But it could be redefined to be something else.
  - On the other hand, signed numbers are a special case:

$+3$ is the number 3

$-3$ is the number $-3$

# Some first things to know about Lisp

- Lisp is an expression language. In its pure form, there are no imperative statements. Instead, the interpreter

  reads an expression,
  evaluates it, and
  prints the resulting value (which might be another expression).

- That is, the interpreter itself (which is of course a computer program) is implemented as what usually called a **read-eval-print** loop.

# Racket and Scheme

- The racket language is fully documented here: `https://docs.racket-lang.org/reference/index.html`
- Racket is a descendant of scheme – there are some differences.
- We will be using (mostly) the old scheme (R5RS) to comply with the book, with small modifications.
- The racket interpreter supports scheme for backward compatibility.
- See demo for the racket package.
- You can also run from the command line.

$\Longrightarrow$ 27

27

$\Longrightarrow$

Here is a more substantive example, illustrating a procedure application (which is what most of the rest of us call a "function call"):

| Expression | Evaluates to |
|---|---|
| (+ 7 4) | 11 |

# Reading this Expression

- The parentheses indicate a function call (in this case; they can also indicate a special form, which we will discuss later).
- The first element of the list inside the parentheses is the procedure – in this case, it is the procedure indicated by $+$, which is addition.
- The remaining elements of the list inside the parentheses are the arguments of the function.
- Note that in effect Scheme implements prefix notation for all functions, even for functions that we would normally write as binary infix operators, like $+$.

# More Examples

| Expression | Evaluates to |
|---|---|
| `(- 7 4)` | `3` |
| `(-7 4)` | `??? (error)` |
| `(* 2 3)` | `6` |
| `(/ 10 2)` | `5` |
| `(/ 10 3)` | `3.33333 or 10/3` |

These are called *compound expressions*. Something like (function arg1 arg2 ...) is called a *combination*.
`(+ (* 3 5) (- 10 6))` evaluates to 19
and so on, to even more complicated expressions.

# Variables

| Expression | Evaluates to |
|---|---|
| `(define size 2)` | `size` |

- creates a variable whose value is 2.
- Actually, normally the value never changes, so in some sense this is not a "variable". Technically, this expression creates a *symbol* (`size`) which is *bound* to the value 2.
- Also note that **define** is not a function, even though syntactically it looks like one.
- Instead, it is an example of what is called a **special form**.
- We'll see more about this as we go on.

# Variables

| Expression | Evaluates to |
|---|---|
| `size` | 2 |
| `(+ size 3)` | 5 |

This is bad practice, but possible:

| | |
|---|---|
| `(define + 3)` | + |
| `(- 7 +)` | 4 |

Of course, in such a case, we could no longer add anything!

# Expressions

There are no statements in Scheme! Everything is an expression.

- Simple expression: There are two kinds of simple expressions:
  - constant: (e.g., `17`, `-6`, `+2.11`, `"hello"`, `#t`, `#f`)
  - variable name (e.g., `abc`, `count`, `...`)
- Compound expression: ("compound" in this context just means "not simple").
- A **compound expression** is a sequence of expressions (each of which may be a simple or compound expression) enclosed in parentheses: `(expr1 expr2 ...)`.
- There are two kinds of compound expressions:
  - Procedure call (e.g., `(+ 5 -11)`, `(square 6)`)
  - Special form (e.g., `(define x 2)`)

# Functions

- Procedures are either
  - primitive (i.e., pre-defined in the language, like +), or
  - user-defined (e.g., `square`)
- In some languages, there is a difference between a *procedure* and a *function*.
- In Scheme they mean the same thing. A *procedure application* is just a fancy way of saying a *function call*.

# User Defined Procedures

**User-defined procedures** are created by an extension of **define**:

```
(define (square x) (* x x))
```

We know we are defining a procedure and not a variable because a parenthesis comes after the **define**. This expression is scanned as follows:

```
(define    (square          x)          (* x x))

           name of      formal          body of
           procedure  parameter         procedure
```

- Of course, there can be more than one formal parameter.
- Also, the body of the procedure can consist of more than one expression.
- We will see examples of this later.

# User Defined Procedures

- In case of more than one expression, all the expressions in the body are evaluated, in the order written.
- The value of the final one is the value of the procedure application.

This expression can even easily be translated into English, as follows:

```
(define  (square      x    )   (*        x        x   ))

  To      square   something,   multiply  it by itself
```

# User Defined Procedures

Now with this definition, we can do the following:

| Expression | Evaluates to |
|---|---|
| `(square 10)` | `100` |
| `(square (+ 2 5))` | `49` |
| `(square (square 3))` | `81` |

Now we can make a further definition:

```
==> (define (sum-of-squares x y) (+ (square x) (square y)))
==> (sum-of-squares 3 4)
25
==>
```

(Note that we are using hyphens in the middle of symbol names. This is OK in Scheme. It wouldn't work in C or Java.)

# User Defined Procedures

Let's go farther:

```
==> (define (F A) (sum-of-squares (+ A 1)(* A 2)))
==>
```

- Exactly how does the Scheme interpreter evaluate something like `(F 5)`?
- We'll get to that shortly.
- First, let's handle conditional expressions.

# Conditionals

We have two special symbols for truth values (also called "Booleans"):

`#t` (means true)

`#f` (means false)

- This is much better than C and languages derived from it, like Java, which really don't have a good notion of Boolean values.
- In Scheme, an expression in a Boolean context is regarded as true if and only if it evaluates to *anything except* `#f`.
- So in particular, any number represents true, even 0. This may surprise you.

| Expression | Evaluates to |
|---|---|
| `(< 2 3)` | `#t` |
| `(= +1 1)` | `#t` |
| `(< 1 -1)` | `#f` |

# Conditional Special Forms

There are two special forms that we use with Boolean expressions: `if` and `cond`. Here are some examples:

```
(if (< 2 3)
    5
    7)
```

Only evaluate the expression selected – that's why it's a special form. For instance,

```
(if (< 2 3)
    5
    (/ 10 0))
```

This does not generate an error. But if **if** were a function (rather than a special form name), it *would* generate an error.

# Conditional Special Forms

```
(define (abs x)
   (cond ((> x 0) x)  ;;; the last x could have been
         ((= x 0) 0) ;;;  a sequence of expressions
         ((< x 0) (- x)) )) ;;; note unary minus
```

- Evaluate only the sequence selected.
- **if** and **cond** are expressions. For instance,

```
(define a 3)
(define b (+ a 1))
(+ 2 (if (< a b) b a))
((if (< a b) + -) a b)
```

# Evaluate a Procedure Application

- Here is the algorithm that the Scheme interpreter uses to evaluate procedure calls.
- It is called *applicative-order evaluation*. It is equivalent to *call-by-value*.

1. **Evaluate** (in any order) all the expressions in the list.
2. **Apply** the procedure (which is the evaluated first expression) to the arguments (which are the rest of the evaluated expressions).

- So we see that a function evaluates all its arguments unconditionally.
- This may seem trivial or obvious, but it actually isn't.
- We'll be talking a lot more about this as the course goes on.

# Evaluate a Procedure Application

- The reason that `if`, `cond`, and `define` are called *special forms* is that even though they look like functions (in that they are the first elements of lists inside parentheses), they act differently.
- For instance, they may not evaluate all their arguments like functions do. And they may not return a value. (The *define* special form, for instance, does not evaluate both its arguments – it only evaluates the last one – and it also does not return a value.)
- Here is an example of applicative-order evaluation (Recall some of our user-defined procedures):

```
(define (square x) (* x x))
(define (sum-of-squares x y) (+ (square x) (square y)))
(define (F A) (sum-of-squares (+ A 1)(* A 2)))
```

# Evaluate a Procedure Application

To evaluate `(F 5)`, we proceed like this:

```
(F 5)
(sum-of-squares (+ 5 1)(* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

# Evaluate a Procedure Application

- What we did was actually not quite correct, because it is not quite what we said we would do.
- We really should do something like this: Evaluate `F` and `5` to transform `(F 5)` into the following:

```
("function having one parameter -- call it A --
and whose value is
(sum-of-squares (+ A 1) (* A 2))"    5)
```

- which then becomes (when we apply the function)

```
(sum-of-squares (+ 5 1)(* 5 2))
```

- and then we proceed as before.

# The **lambda** Special Form

- There is a way of doing this: we use another special form. We could have defined

```
(define F
   (lambda (A) (sum-of-squares (+ A 1)(* A 2))))
```

- Thus, `(lambda (x) (<stuff>))` is an unnamed function of 1 parameter whose body is `(<stuff>)`.
- In fact, `(define (f x y) (<expression in x and y>))` is really turned by the interpreter internally into

```
(define f (lambda (x y) (<expression in x and y>)))
```

# Some Examples

```
==> ((lambda (x) (+ x 3)) 4)
7
==>
```

The trouble is that since this function has no name, it can only be used once. So the idea is that we can use **define** to give it a name:

```
==> (define f (lambda (x) (+ x 3)) )
==> (f 4)
7
==> (f -3)
0
==>
```

and so on.

# Some Examples

If we entered this expression:

```
==> (define (f x) (+ x 3))}
==>
```

then the interpreter actually turns it internally into the previous one:

```
(define f (lambda (x) (+ x 3)) )
```

Here is how `(F 5)` is really evaluated internally by the Scheme interpreter:

# Some Examples

```
(F 5)
;; Now eval F
((lambda (A) (sum-of-squares (+ A 1)(* A 2))) 5)
;; Now apply the function
(sum-of-squares (+ 5 1) (* 5 2))
;; Now eval every element
((lambda (x y) (+ (square x)(square y))) 6 10)
;; Now apply the function
(+ (square 6) (square 10))
;; Now eval every element
(+ ((lambda (x) (* x x)) 6) ((lambda (x) (* x x)) 10))
;; Now apply lambdas
(+ (* 6 6) (* 10 10))
;; Now eval every element
(+ 36 100)
;; Now apply the function
136
```

# Some Examples

```
==> (define four 4)
==> (define (five) 5)
==> four
4
==> five
#<procedure:five>
==> (four)
*** error
==> (five)
5
==> (procedure? four)
#f
==> (procedure? five)
#t
```

# Some Examples

- Note: `procedure?` is a primitive procedure.
- Note that the question mark is just another character in the name.
- The convention in Scheme is that a procedure that evaluates to a Boolean ends in a question mark.

# Normal Order Evaluation

- This is a different method of evaluating expressions.
- It is not the method used in Scheme, but it is very important, and we will see examples of this later on.
- If we changed Scheme so that it used normal-order evaluation, we would evaluate procedure applications like this:

# Normal Order Evaluation

1. **Evaluate** the leftmost subexpression of the list (i.e., the operator).
2. Do the following:
   1. If the procedure that is the value of that expression is primitive, then
      - **evaluate** the other subexpressions (i.e., the arguments), and
      - **apply** the procedure to the resulting evaluated arguments (as usual).
   2. Otherwise (i.e., if the procedure is a user-defined procedure),
      - **apply** the procedure to the *unevaluated* argument expressions.

Normal-order evaluation corresponds to Algol's *call-by-name*.

# Normal Order Evaluation

Let's evaluate `(F 5)` using normal-order evaluation:

```
(F 5)
;; Now eval F
((lambda (A) (sum-of-squares (+ A 1)(* A 2))) 5)
;; Now apply the function
(sum-of-squares (+ 5 1) (* 5 2))
;; Now eval sum-of-squares
((lambda (x y) (+ (square x)(square y))) (+ 5 1)(* 5 2))
;; Now apply fn.
(+ (square (+ 5 1)) (square (* 5 2)))
;; Now eval arguments
(+ ((lambda (x) (* x x)) (+ 5 1)) ((lambda (x) (* x x)) (* 5 2))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

# Normal Order Evaluation

Normal-order evaluation is also called *lazy evaluation* or *delayed evaluation* – we don't actually evaluate anything until we absolutely need to.

- **Basic Fact:** Whenever applicative-order evaluation yields a result, normal-order evaluation yields the same result. But there are cases when normal-order evaluation is more powerful.

For example:

```
(define f (lambda (x y) x))

(f 4 (/ 2 0))
```

The reason that Scheme uses applicative-order evaluation is:

- It is easier to implement. This is a minor reason.
- It leads to much more efficient code. This is the main reason.