SICP Notes 01

https://mitpress.mit.edu/sicp/
http://www.aduni.org/courses/sicp/index.php

Geoffrey Matthews

Department of Computer Science
Western Washington University

March 31, 2015

Every powerful language has

- primitive expressions: the simplest entities, such as 3 and +
- means of combination: buidling compound elements from simpler ones such as
  `(+ 3 4)`
- means of abstraction: a way for naming compound elements and then manipulating them as units such as
  `(define square (lambda (x) (* x x)))`

Scheme does the following three things:

- Reads an expression
- Evaluates it to produce a value
- Prints the value

The returned value has a small set of types, including number, boolean and procedure. (Later, we'll see symbol, pair, vector, and promise (stream).)

There are 4 types of expressions:

- Constants: numbers, booleans. Examples: `4 3.141592 #t #f`
- Variables: names for values. We create these using the special form `define`
- Special forms: have special rules for evaluation. In addition, you may not redefine a special form.
- Combinations: `(<operator> <operands>)`. These are sometimes called "function calls" or "procedure applications."

The first two types of expressions (constants and variables) are primitive expressions – they have no parentheses. The second two types are called compound expressions – they have parentheses.

Mantras

- Every expression has a value (except for errors, infinite loops and the `define` special form)
- To find the value of a combination:
  - Find values of all subexpressions in any order
  - Apply the value of the first to the values of the rest
- The value of a `lambda` expression is a procedure

Lambda

When you hear the words "write a procedure," you should think of `lambda`. Lambda is a special form that creates a procedure. There are three parts to the lambda expression:

- lambda
- parameter list
- body

For example, let's write a procedure to square a number:
`(lambda (x) (* x x))`

- (x) is the parameter list. In this case, we only have one parameter.
- (* x x) is the body of the lambda. The body of the lambda will not be evaluated until the procedure is applied.

How do we use this procedure that we just wrote? Well, we can write `((lambda (x) (* x x)) 3)` This will return 9.

Define

- If we want to square 9, we can write
  `((lambda (x) (* x x)) 9)`
- But we don't want to have to keep typing the procedure over and over.
- This leads us to another special form: `define`

Define

Define is a special form that allows us to name objects. Define has three parts:

- define
- name
- the object you want the name to be bound to

Here are some examples:

- `(define pi 3.141592)`
- `(define four 8)`

Define

- We can use define to name our procedure we wrote above to allow us to use it without having to retype the lambda expression over and over.
  `(define square (lambda (x) (* x x)))`
- Now we can write
  `(square 9)`
- Instead of
  `((lambda (x) (* x x)) 9)`

Special forms

Special forms are those expressions that begin with an open parenthesis followed by one of the 15 "magic words":

`and`	`define`	`let`	`quasiquote`
`begin`	`do`	`let*`	`quote`
`case`	`if`	`letrec`	`set!`
`cond`	`lambda`	`or`	

Special Forms Have Special Rules

- Recall from the mantras that to find the value of a combination, you find the values of all of the subexpressions in any order.
- With special forms, this is not done.
- The order of the evaluation of the subexpressions is specified for each special form.

and

`(and <exp1> <exp2> ...)`

- `and` evaluates the expressions one at a time in left to right order.
- As soon as one of the expressions evaluates to `#f` (false), the value of the and expression is `#f` (false) and none of the remaining expressions are evaluated.
- If none of the expressions evaluates to `#f` (false), `#t` (true) is returned.

or

`(or <exp1> <exp2> ...)`

- `or` evaluates the expressions one at a time in left to right order.
- As soon as one of the expressions evaluates to anything other than `#f` (false), the value of the or expression is returned and none of the remaining expressions are evaluated.
- If none of the expressions evaluates to something other than `#f`, then `#f` (false) is returned.

if

`(if <predicate> <consequent> <alternative>)`

- The predicate is evaluated.
- If it is anything other than `#f`, the value of the consequent will be returned.
- If it is `#f`, the value of the alternative will be returned.
- In this special form, never will the consequent and alternative both be evaluated.

cond

```
(cond (<pred1> <exp1>)
      (<pred2> <exp2>)
      ...
      (else <expn>))
```

- The first predicate is evaluated.
- If it is anything other than `#f`, the value of the first expression will be returned.
- If it is `#f`, the second predicate will be evaluated.
- The computer will continue to evaluate the predicates until one is something other than `#f`.
- The value of the expression corresponding to the non-`#f` predicate will be returned.
- Note that the `else` will always be true (*i.e.* not `#f`).