CS61A Lecture 25

Monday, October 28, 2019

Announcements

- Video-only lecture.
- Midterm 2 results, exam and solution have been posted.
  - Regrade requests due next Monday.
- Homework 6 deadline extended to Friday, November 1.
- Lecture 24 was a special, video-only lecture given by Alan Kay, the father of object-oriented programming. There was no class content in the lecture, so there are no notes for it.

Scheme

You now know how to program in Python. Well, there's still more to learn, technically, as we haven't covered the standard libraries and some minor syntatic features, but by and large you can already go out and build stuff.

Today, we will begin learning how to code in a new programming language called Scheme. Well, it's new to you: Scheme is one of the world's oldest programming languages, but it was chosen for its particular focus on functional programming. The ideas you learn in Scheme will make you a better programmer in any language, because as we said before, concepts transfer between languages.

The best way to learn a new programming language is to watch lectures, then make stuff yourself. That's what lab this week is for.

So far you've learnt about functions and data: the essence of programming. We will now move on to Scheme: we learn it for its history (it was influential in the creation of Python), for its simplicity and beauty, and for the learning potential.

Scheme is a dialect of a language called Lisp, one of the two oldest programming languages that's still used today, and one of the only ones that is still growing.

Alan Kay, the co-inventor of Smalltalk, said it was "the greatest single programming language ever designed." Neal Stephenson, DeNero's favorite sci-fi author, called it "the only computer language that is beautiful." And Brian Harvey, 61A instructor of over two decades, once called it "God's programming language."

The beauty of Lisp is its implicity. The entire language can be learnt in a day, but it can be used to build programs as or even more complex than the ones we've seen so far.

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2, 3.3, true, +, quotient etc.
- Combinations: always contained within parentheses. (quotient 10 2), (not true) etc.

Numbers are self-evaluating: symbols are bound to values. Call expressions include an operator and 0 or more operands in parentheses.

```
scm> (quotient 10 2)
5
; "quotient" names Scheme's built-in
; integer division procedure (what we
; know as functions)
scm> (quotient (+ 8 7) 5)
3
scm> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
      (+ (- 10 7)
         6))
57
\ Scheme doesn't care about indentation and new lines!
```

Since Scheme doesn't care how you separate your expressions, you can type them like above to make them more human-readable. Expressions indented to the same level will be operated on with the built-in interpreter.

In addition to these basic arithmetic procedures that are built-in, there are other ones as well.

```
> (number? 3)
#t
; We will drop the dumb "scm" text in
; front of the arrow bracket to make
; the code more readable.
> (zero? 2)
#f
> (integer? 2.2)
#f
```

The question mark is part of the procedure name. It is a good reminder that it returns a true or false value.

Special Forms

We haven't learned ALL of the Scheme language yet. There are other types of expressions besides call expressions, and they are called special forms (anything that's not a call expression).

If looks just like a compound expression:

```
> (if <predicate> <consequent> <alternative>)
; In other words, if <condition> <if suite>
; <else suite>
```

The evaluation procedure for the Scheme if clause is the same as the evaluation procedure for Python's.

And and or also work similarly to Python's. By default they can take it more than 2 arguments:

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

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

Their evaluation procedure also work similarly to Python's.

The binding symbol is the assignment statement. In Scheme, we bind new values to symbols with the define special form:

```
> (define <symbol> <expression>)
> (define pi 3.14)
> (* pi 2)
6.28
```

Scheme uses the same frame-based model for evaluation of expressions as Python. The symbol "pi" is bound to 3.14 in the global frame.

You also bind new procedures to names with the same define keyword:

```
> (define (<symbol> <parameters>) <body>)
> (define (abs x)
      (if (< x 0)
          (-x)
          x))
> (abs -3)
3
```

In Scheme, we already know how to evaluate call expressions, and that call expressions can contain other call expressions. You can continue to define recursive functions because they will continue to work just fine. In fact, they are everywhere.

Higher-order functions will also continue to work. Scheme is lexically scoped like Python, so you can refer to parameters found in a function's parent frame within the child frame.

Scheme Interpreter

So how do we execute all this Scheme code anyway? You can download Scheme like Python, or you can use an online interpretr. So far, Professor DeNero has been using the interpreter built by past students in Project 4 of the class. Project 4 has you building your own interpreter for the Scheme language in Python!

Lambda Expressions

Lambda expressions create new procedures in Scheme. They evaluate to anonymous procedures.

```
(lambda (<formal parameters>) <body>)
```

Lambda is very important in Scheme! The following two expressions are equivalent:

```
> (define (plus 4 x) (+ x 4))
> (define plus4 (lambda (x) (+x 4)))
```

In a call expression, you can have a lambda expression too:

```
> ((lambda (x y z) (+ x y (square z))) 1 2 3)
```

Lists

Python has many types of built-in data structures: lists, tuples, dictionaries, and sets. among others. Scheme has one: lists.

The Scheme list is essentially a linked list.

Lists were created in the 1950s, back when computer scientists loved using confusing names. They created lists with the following syntax:

- `cons` is a two-argument procedure that creates a linked list
- `car` is a procedure that returns the first element of a list
- `cdr` is a procedure that returns the rest of a list
- `nil` is the empty list.

So, to build this linked list from Python:

```
Link(2,Link.empty)
```

In Scheme, we would type:

```
(cons 2 nil)
```

Scheme lists are written in parentheses with elements separated by spaces. They have the structure of a linked list, but that doesn't show in the display.

```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil)))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
> (cons 1 x)
(1 1 2)
```

When we call `cons`, the first argument can be anything, but the second argument must either be another `cons`, or `nil`.

There are a few built-in procedures for lists:

```
> (list? x)
```

```
#t
> (list? 3)
#f
> (list? (car s))
#f
> (list? nil)
#t
> (null? nil)
#t
```

I can also build a list with the `list` procedure, which is still a linked list but may be more convenient for you.

```
> (list 1 2 3 4)
(1 2 3 4)
> (cons 0 (cdr (list 1 2 3 4)))
(0 2 3 4)
```

Symbolic Programming

Lisp is well-known for introducing the idea of symbolic programming: manipulating lists of symbols which represent things in the world as structured objects.

With Lisp, you could do more than just compute a number, you could manipulate whole equations. For this reason, for many years, Lisp was the standard language for artificial intelligence and any implementation that needed automatic manipulation of mathematical equations (such as symbolic differentiation or auto-verifying proofs).

How do we do this? There is a feature in the language that enables this.

Symbols normally refer to values, but how do we refer to them?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

We've lost all notion of what symbols were used to create this value. Quotation is used to refer to symbols directly. If I make a list with a single quote `a` and a single quote `b`, then I build a list with the symbols `a` and `b`.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

The `'` is shorthand for a special form for a special syntatic feature `(quote <symbol>)`.

Quotation can also be applied to combinations to form lists:

```
> '(a b c)
```

```
(a b c)
```

Notice we can build a list with `c` even though we never defined what it was yet; Scheme expects it may mean something in the future!