1 Introduction

In the next part of the course, we will be working with the Scheme programming language. In addition to learning how to write Scheme programs, we will eventually write a Scheme interpreter in Project 4!

Scheme is a dialect of the Lisp programming language, a language dating back to 1958. The popularity of Scheme within the programming language community stems from its simplicity – in fact, previous versions of CS 61A were taught in the Scheme language.

2 Primitives

Scheme has a set of atomic primitive expressions. Atomic means that these expressions cannot be divided up.

```
scm> 123
123
scm> 123.123
123.123
scm> #t
True
scm> #f
False
scm> 'a ; this is a symbol
a
```
To define variables:
```
scm> (define a 3)
a
scm> a
3
```

The `define` statement binds a value to a variable (just like the assignment operator in Python); in addition, `define` returns the variable name (in this case, `a`).

More precisely, `define` returns the symbol `a`. As you saw above, when you type `a`, you also get the symbol `a`. This is because when you use the single quote, you’re telling Scheme not to follow the normal rules of evaluation and just have the symbol return as itself.

### 2.1 Questions

1. What would Scheme print?
   ```
   scm> (define a 1)
   `scm> a
   `scm> (define b a)
   `scm> b
   `scm> (define c 'a)
   `scm> c
   ```

Now, just defining variables and printing out primitives isn’t very useful. You want to call functions too:
```
scm> (+ 1 2)
3
scm> (- 2 3)
-1
```
To call a function in Scheme, you first need a set of parentheses. Inside of the parentheses, you give the symbol for the function name, then you give the arguments (remember the spaces!).

Evaluating a Scheme function call works just like Python:

1. Evaluate the operator (the first expression after the (, then evaluate each of the arguments.
2. Apply the operator to those evaluated arguments.

When you evaluate \((+ 1 2)\), you evaluate the + symbol which is bound to a built-in addition function, then you evaluate 1 and 2. Finally, you apply the addition function to 1 and 2.

Some important functions you’ll want to use are:

- \(+, -, *, /\)
- \(eq?, =, >, >=, <, <=\)

### 3.1 Questions

1. What would Scheme print?

   \texttt{smd> (+ 1)}

   \texttt{smd> (* 3)}

   \texttt{smd> (+ (* 3 3) (* 4 4))}

   \texttt{smd> (define a (define b 3))}

   \texttt{smd> a}

   \texttt{smd> b}
There are certain expressions that look like function calls, but don’t follow the rule for order of evaluation. These are called *special forms*. You’ve already seen one — define, where the first argument, the variable name, doesn’t actually get evaluated to a value.

**4.1 If Statements**

Another common special form is the *if* form. An *if* expression looks like:

\[
\text{if} \ <\text{CONDITION}> \ <\text{THEN}> \ <\text{ELSE}>
\]

where <CONDITION>, <THEN> and <ELSE> are expressions. First, <CONDITION> is evaluated. If it evaluates to #f, then <ELSE> is evaluated. Otherwise, <THEN> is evaluated. Every primitive expression that is not False evaluates to “true”.

```sml
scm> (if (< 4 5) 1 2)
1
scm> (if False (/ 1 0) 42)
42
```

**4.2 Boolean operators**

Boolean operators (**and** and **or**) are also special forms because they are short-circuiting operators (just like in Python).

```sml
scm> (and 1 2 3)
3
scm> (or 1 2 3)
1
scm> (or True (/ 1 0))
True
scm> (and False (/1 0))
False
scm> (not 3)
False
scm> (not True)
False
```
4.3 Questions

1. What does Scheme print?

```scheme
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
```

```scheme
scm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
```

```scheme
scm> ((if (< 4 3) + -) 4 100)
```

4.4 Lambdas and Defining Functions

Scheme has lambdas too! The syntax is

```
(lambda (<PARAMETERS>) <EXPR>)
```

Like in Python, lambdas are function values. Also like in Python, when a lambda expression is called in Scheme, a new frame is created where the parameters are bound to the arguments passed in. Then, `<EXPR>` is evaluated under this new frame. Note that `<EXPR>` is not evaluated until the lambda function is called.

```scheme
scm> (define x 3)
x
```

```scheme
scm> (define y 4)
y
```

```scheme
scm> ((lambda (x y) (+ x y)) 6 7)
```

13

Like in Python, lambda functions are also values! So you can do this to define functions:

```scheme
scm> (define square (lambda (x) (* x x)))
square
```

```scheme
scm> (square 4)
```

16

This can be a bit tedious though. Luckily Scheme has a shortcut: our old friend `define`:

```scheme
scm> (define (square x) (* x x))
square
```

```scheme
scm> (square 5)
```

25

When you do `(define (<FUNCTION NAME> <PARAMETERS>) <EXPR>)`, Scheme will automatically transform it to `(define <FUNCTION NAME> (lambda (<PARAMETERS>) <EXPR>)`. In this way, lambdas are more central to Scheme than they are to Python.
4.5 Let

There is also a special form based around lambda: let. The structure of let is as follows:

```
(let ( (<SYMBOL1> <EXPR1>)
     ...
    (<SYMBOLN> <EXPRN>) )
  <BODY> )
```

This special form really just gets transformed to:

```
(l lambda
 (<SYMBOL1> ... <SYMBOLN>) <BODY>) <EXPR1> ... <EXPRN>)
```

let effectively just binds symbols to expressions, then runs its body. This can be useful if you need to reuse a value multiple times, or if you want to make your code more readable:

```
(define (sin x)
  (if (< x 0.000001)
    x
    (let ( (recursive-step (sin (/ x 3))) )
      (- (* 3 recursive-step)
        (* 4 (expt recursive-step 3))))))
```

4.6 Questions

1. Write a function that calculates factorial. (Note we have not seen any iteration yet.)
   (define (factorial x)
   )

2. Write a function that calculates the \(n\)th Fibonacci number.
   (define (fib n)
   (if (< n 2)
     1
   )

5 Pairs and Lists

So far, we have lambdas and a few atomic primitives. How do we create larger, more complicated data structures? Well, the most important data structure in Scheme is the pair. A pair is an abstract data type with the constructor cons (which takes two arguments), and two selectors, car and cdr (which get the first and second argument respec-
car and cdr don’t stand for anything anymore, but if you want the history go to http://en.wikipedia.org/wiki/CAR_and_CDR.

```
scm> (define a (cons 1 2))
a
scm> a
(1 . 2)
scm> (car a)
1
scm> (cdr a)
2
```

Note that when a pair is printed, the car and cdr elements are separated by a period. Remember, cons always takes in exactly two arguments.

A common data structure that you build out of pairs is the list. A list is either the empty list, which is another primitive represented as ’() or nil, or a cons pair where the cdr is a list. (Note the similarity to Links!)

```
scm> ’()
()
scm> nil
()
scm> (cons 1 (cons 2 nil))
(1 2)
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

Note that there are no dots here. When a dot is followed by a left parenthesis, the dot, left parenthesis, and matching right parenthesis are deleted. You can check if a list is nil with the null? function.

A shorthand for writing out a list is:

```
scm> ’(1 2 3)
(1 2 3)
scm> ’(define (square x) (* x x))
(define (square x) (* x x))
```

You might notice that the evaluation of the second expression looks a lot like Scheme code. That’s because Scheme code is made up of lists. When you quote an expression (like a list), you’re telling Scheme not to evaluate the expression, but instead keep it as is. This is one of the reasons why Scheme is cool – it can be defined within itself!
5.1 Questions

1. Fill in the following to complete an abstract data type for binary trees, in which each node has at most 2 children, left and right:
   
   `(define (make-btree entry left right)
    (cons entry (cons left right)))
   `
   
   `(define (entry tree)
    )
   `
   
   `(define (left tree)
    )
   `
   
   `(define (right tree)
    )
   `

2. Using the above definition, write a function that sums up the entries of a binary tree, assuming that the entries are all numbers.
   
   `(define (btree-sum tree)
   )
   `

3. Define `map`, where the first argument is a function and the second a list. This should work like Python’s `map`.
   
   `(define (map fn lst)
   )`
4. Define reduce, where the first argument is a function that takes two arguments, the second is a starting value, and the third is a list. This should work like Python’s reduce.
\[
\text{(define (reduce fn s lst)}
\]

6 Extra Questions

1. Write a Scheme function that, when given an element, a list, and a position, inserts the element into the list at that position.
\[
\text{(define (insert element lst position)}
\]

2. Write a Scheme function that, when given a list, such as \((1\ 2\ 3\ 4)\), duplicates every element in the list (i.e. \((1\ 1\ 2\ 2\ 3\ 3\ 4\ 4)\))
\[
\text{(define (duplicate lst)}
\]