# INTERPRETERS 6a

COMPUTER SCIENCE 61A

July 30, 2013

In last week's discussion, we introduced the **Calculator** language, a simple Scheme-based language that supports simple arithmetic operations.

We will be continuing using Calculator as an example to study how interpreters work. In this discussion, we move onto a more full-fledged version of the Calculator interpreter that closely resembles Project 4: the Scheme interpreter.

## 1 Warmup

1. Describe what *tokenization* does. What does it take as input? What does it return as output?

2. Describe what *parsing* does. What does it take as input? What does it return as output?

3. Describe what *evaluation* does. What does it take as input? What does it return as output?

# 2 Tokenization

## 2.1 Concept

In its broadest sense, tokenization takes a string of user input and converts it into a sequence of tokens. There are a couple of details every interpreter needs to determine:

- **What counts as a token?** In Calculator, the only valid tokens are parentheses, numbers (e.g. `3`, `5.5`), and arithmetic operators (e.g. `+`, `*`).
- **What type of sequence will contain the tokens?**

In this class, we don't focus on *how* the tokenization process happens. Instead, the important takeaway is *what* the tokenization returns, and how to interact with it.

In `minicalc` (the first interpreter we saw, in Discussion 5b), the tokenizer returns a **Python list** of tokens. From an educational standpoint, we have already been using lists for a while in this class, so it is (presumably) more familiar to you, the student.

## 2.2 Buffers

In `scalc` (the Calculator interpreter we introduced today), the tokenizer returns a `Buffer` **object**. A `Buffer` object is similar to a Python list, but only supports two methods:

- `pop`: the `Buffer` class's version of pop takes exactly 0 arguments, and removes the **first** token from the Buffer (e.g. removes from the front of the Buffer).
- `current`: returns the first token in the Buffer, but does not remove it from the Buffer.

`Buffer` objects are not built-in to Python. We have implemented a `Buffer` class in both `scalc` and in the Project 4 Scheme interpreter.

## 2.3 Questions

1. What would Python print, assuming the tokenizer is analyzing Calculator input?

```
>>> buffer = Buffer(tokenize_line('(+ 3 4)'))
>>> buffer.current()



>>> buffer.pop()



>>> buffer.current()
```

```
>>> buffer = Buffer(tokenize_line('+ ) * 4'))
>>> # buffers don't care about syntactic correctness
>>> token = buffer.pop()
>>> token


>>> buffer.pop()
```

# 3 Parsing

## 3.1 Concept

In an interpreter, the parser takes a sequence of tokens (from the tokenizer) and converts it into a data structure that the evaluator (seen later on) can understand.

In `minicalc` (the interpreter from Discussion 5b), the parser consisted of two functions: `read_exp` and `read_tail`.

```
def read_exp(tokens):
    """In minicalc, tokens is a Python list"""
    ...
    token = tokens.pop(0)
    if token == '(':
        exp = read_tail(tokens)
        ...

def read_tail(tokens):
    if tokens[0] == ')':
        tokens.pop(0)
        return nil
    return Pair(read_exp(tokens), read_tail(tokens))
```

In `scalc` and the Project 4 Scheme interpreter, the parser is similarly composed of two functions: `scheme_read` and `read_tail`.

```
def scheme_read(src):
    """In scalc and scheme, src is a Buffer object"""
    ...
    val = src.pop()
    ...
    if val == '(':
        return read_tail(src)
    ...

def read_tail(src):
    ...
    if src.current() == ')':
        src.pop()
        return nil
    first = scheme_read(src)
    rest = read_tail(src)
    return Pair(first, rest)
```

Notice that the two versions of the parser look very similar. Try to see which parts correspond to each other!

## 3.2 Mutual Recursion

Recall that *mutual recursion* refers to two (or more) functions that call continually call each other. You'll notice that `scheme_read` and `read_tail` are mutually recursive — this allows their implementation to be relatively straightforward. The procedure is as follows:

1. If `scheme_read` sees a `'('`, it calls `read_tail`
2. `read_tail` then calls `scheme_read` to parse the first complete Scheme expression in the Buffer. This becomes the `first` part of the resulting `Pair`. Remember that `scheme_read` *removes* tokens from the Buffer!
3. `read_tail` then calls itself recursively to parse the `rest` of the `Pair`.

### 3.3 Questions

1. For each of the following lines of input, determine what `scheme_read` would return.

```
>>> scheme_read(Buffer(tokenize_line('4')))
```

```
>>> scheme_read(Buffer(tokenize_line('(+ 3 4)')))
```

```
>>> scheme_read(Buffer(tokenize_line('(+ (- 5 4) 3)')))
```

2. For the following Buffer of tokens determine how many times `scheme_read` is called, and how many times `read_tail` is called. The first one is done for you.

```
>>> '(', '+', 3, 4, ')'
```

`scheme_read`: 4

`read_tail`: 4

```
>>> 4
```

```
>>> '(', '+', '(', '-', 4, 3, ')', 5, ')'
```

## 4 Evaluation

### 4.1 Concepts

In the interpreter, the evaluator takes its input from the parser and computes a value based on the rules of the language. In Calculator, the evaluator takes an expression (e.g. a `Pair` object) from the parser (`scheme_read`), and computes an arithmetic operation.

In minicalc, the evaluator consists of two functions:

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(exp.car, map_rlist(calc_eval, exp.cdr))
    else:
        return exp

def calc_apply(op, args):
    if op == '+':
        ...
    elif op == '-':
        ...
```

In scalc, the evaluator is similarly composed to of two functions:

```
def calc_eval(exp):
    if type(exp) in (int, float):
        ...
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    ...

def calc_apply(op, args):
    ...
    if op == '+':
        ...
    elif op == '-':
        ...
```

Again, try to figure out which parts correspond to each other! One thing you'll notice is that the Pair objects used by scalc have different names for the first and the rest than the Pairs used in minicalc.

## 4.2 Mutual Recursion...?

In both minicalc and scalc, the calc_apply function is simple enough that it doesn't make a mutually recursive call to calc_eval. However, in more sophisticated interpreters (like the Scheme interpreter in Project 4), the apply function will make a mutually recursive call to the eval function.

### 4.3 Questions

1. For each of the following lines, determine how many times `calc_eval` and `calc_apply` are called.

```
>>> '4'
```

```
>>> '(+ 2 3)'
```

```
>>> '(+ 2 (- 3 4) 5)'
```

2. In Discussion 5b, we implemented the `and` special form. Here, we'll implement the `or` special form. First of all, why are `and` and `or` considered special forms?

3. `calc_eval` has been modified to call a function `do_or_form`, which handles the `or` operator. Implement `do_or_form` so that it works.

```
def calc_eval(exp):
    ...
    elif isinstance(exp, Pair):
        if exp.first == 'or':
            return do_or_form(exp.rest)
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)

def do_or_form(exp):
    "*** YOUR CODE HERE ***"
```