Appendix of Useful Functions

0.1 Tuples

Length: The function `len` returns the length of a tuple. For example, `len((3, 4, 5))` evaluates to 3.

Item Selection: The expression `tup[pos]` evaluates to the item at position `pos` of tuple `tup`. For example, if `tup` is `(1, 2, 3, 4)`, then `tup[0]` evaluates to `1` and `tup[2]` evaluates to `3`.

Slicing: The expression `tup[start:end:size]` evaluates to a smaller tuple that contains the items from `start` to `end` (excluding the `end`), with a step size of `size`. All arguments are optional. For example, if the tuple `tup` is `(1, 2, 5, 7, 8, 10)`, then `tup[0:3]` returns the tuple `(1, 2, 5)` and `tup[0:4:2]` returns the tuple `(1, 5)`.

Concatenation: Tuples can be concatenated into a larger tuple, which contains the elements of both tuples, using the `+` operator. For example, `(0, 1, 2) + (4, 5)` yields the tuple `(0, 1, 2, 4, 5)`.

0.2 Ranges

The expression `range(start, end, size)` returns an iterable "range object" (*not* a tuple), which is a sequence that contains the integers from `start` to `end` (excluding the `end`), with a step size of `size`. The third argument is optional. If the first and third arguments are not provided, the start value is assumed to be 0. For example, the statement

```
for val in range(0, 5):
    print(val)
```

will print the values `0`, `1`, `2`, `3` and `4`.

0.3 Map, Filter, Reduce

`map(func, seq)` applies the function `func` to the items in the sequence `seq` and returns the results in an iterable "map object" (*not* a tuple). You would use the `tuple` constructor to create a tuple of these results. For example, `tuple(map(lambda x: x*x, (1, 2, 3, 4)))` applies the squaring function to all of the items in the sequence (here, a tuple), and returns the tuple `(1, 4, 9, 16)`.

`map` can also take in more than one sequence, in which case the function `func` is applied first to the first elements of all sequences, then to the second elements of all sequences, and so on. For example, `tuple(map(lambda x, y, z: x+y+z, (1, 2, 3), (4, 5, 6), (7, 8, 9)))` returns the tuple `(1+4+7, 2+5+8, 3+6+9)` or `(12, 15, 18)`.

`filter(pred, seq)` filters out the items in the sequence `seq` that do not satisfy the predicate `pred` and returns the remaining items in an iterable "filter object" (*not* a tuple). You would use the `tuple` constructor to create a tuple of these items. For example, `tuple(filter(lambda x: x>5, (3, 5, 7, 9, 13)))` filters out all of the items in the sequence (here, a tuple) that are not greater than 5, returning the tuple `(7, 9, 13)`.

`reduce(func, seq, initial_value)` returns the result of applying `func` cumulatively to the items of a sequence `seq` from left to right, starting with the `initial_value`. For example, `reduce(lambda x, y: x+y, (1, 2, 3, 4), 10)` returns the value `((((10 + 1) + 2) + 3) + 4)`. The argument `x` maintains the value accumulated so far, and the argument `y` will be iterated through the elements of the sequence from left to right.

If `initial_value` is not provided, `reduce` will use the first element of the sequence as the initial value; if the sequence is empty, and no initial value is provided, `reduce` will return an error.

0.4 Immutable Recursive Lists (IRLists)

`make_irlist(first, rest=empty_irlist)`: Returns a new IRList by prepending item `first` to the front of the IRList `rest`, which is the empty IRList (`empty_irlist`) by default.

`irlist_first(irlist)`: Returns the first item in the IRList `irlist`.

`irlist_rest(irlist)`: Returns an IRList of all items *except* the first item in the IRList `irlist`.

`could_be_irlist(thing)`: Returns `True` if `thing` could be an IRList, and `False` otherwise.

`irlist_populate(*items)`: Returns a new IRList populated with the items provided in `items`.

`irlist_len(irlist)`: Returns the length of the IRList `irlist`.

`irlist_select(irl, pos)`: Returns item at position `pos` (counting from zero) of the IRList `irl`.

`irlist_insert(irlist, index, item)`: Returns a new IRList where the item at position `index` (counting from zero) is replaced with the new item `item`.

`irlist_remove(irlist, index)`: Returns a new IRList where the item at position `index` (counting from zero) of IRList `irlist` is removed.

`irlist_append(irlist1, irlist2)`: Returns a new IRList that is the result of adding the elements in IRList `irlist2` to the end of IRList `irlist1`.

`irlist_map(func, irlist)`: Returns a new IRList that contains the results of applying `func` to every item in IRList `irlist`.

`irlist_filter(pred, irlist)`: Returns a new IRList that contains the items in IRList `irlist` that satisfy the predicate `pred`.

`irlist_str(irlist)`: Returns a string representation of the IRList `irlist`.

0.5 Immutable Dictionaries (IDicts)

`make_idict(*mappings)`: Returns a new IDict from the tuple of key-value pairs provided in `mappings`.

`idict_keys(idict)`: Returns a tuple of the keys belonging to the IDict `idict`.

`idict_values(idict)`: Returns a tuple of the values that are mapped to in the IDict `idict`.

`could_be_idict(thing)`: Returns `True` if `thing` could be an IDict, and `False` otherwise.

`idict_select(idict, key)`: Returns the value that the key `key` maps to in the IDict `idict`, or `None` if the key is not present in `idict`.

`idict_insert(idict, key, value)`: Returns a copy of the IDict `idict`, updated with the key `key` now mapped to value `value`.

`idict_len(idict)`: Returns the number of mappings in the IDict `idict`.

`idict_items(idict)`: Returns a tuple of key-value pairs stored in IDict `idict`.

`idict_str(idict)`: Returns a string representation of the IDict `idict`.

0.6 Immutable Trees (ITrees)

`make_itree(datum, children=())`: Returns a new ITree from the datum provided as `datum` and the tuple of children ITrees provided as `children`.

`itree_datum(itree)`: Returns the datum of the ITree `itree`.

`itree_children(itree)`: Returns the tuple of children ITrees of the ITree `itree`.

`itree_items(itree)`: Returns a tuple of all the items in the ITree `itree`.

`itree_size(itree)`: Returns the number of items in the ITree `itree`.

`is_leaf(itree)`: Returns `True` if the ITree `itree` is a leaf (or has no children).

`itree_map(func, itree)`: Returns a new ITree that results from applying the function `func` to each of the items in the ITree `itree`.

`itree_str(itree)`: Returns a string representation of the ITree `itree`.

0.7 Mutable Lists

Lists, just like tuples, support `len`, item selection, slicing and concatenation. Slicing produces a new list if used anywhere on the right-hand side of an assignment expression. Concatenation also produces a new list.

Updates: Specific positions in the list can be updated. For example, `lst[2] = 5` updates the second element (counting from zero) of the list `lst` to `5`, but *does not create a new list*. Slices of a list can also be updated. For example, `lst[2:5] = [3, 4, 'foo']` replaces elements 2, 3 and 4 of list `lst` with the items `3`, `4` and `'foo'` respectively. The size of the list on the right does not have to match the number of elements in the slice on the left: if the list on the right is larger, then the list expands; if the list is smaller, the list shrinks.

Other Methods: The methods below *mutate* the list on which they are called, and do not create a new list:

`lst.append(item)`: Appends the item `item` to the list `lst`.

`lst.extend(m)`: Extends the list `lst` using the items in the list `m`.

`lst.count(item)`: Counts the number of times `item` appears in the list `lst`.

`del` `lst[pos]`: Deletes an item at position `pos` of the list `lst`.

`lst.index(item)`: Returns the index (starting from zero) of the first occurrence of `item` in the list `lst`, or a `ValueError` if the item is not present in the list.

`lst.insert(pos, item)`: Inserts the item `item` at position `pos` of the list `lst`.

`lst.pop(pos)`: Removes and returns the item at position `pos` of the list `lst` (or at the last position if no argument is provided), or returns a `ValueError` if the list is empty.

`lst.remove(item)`: Removes the first occurrence of `item` in the list `lst`, or returns a `ValueError` if the item is not present.

`lst.reverse()`: Reverses the items of the list `lst`.

`lst.sort()`: Sorts the items of the list `lst`.

0.8 Generator Expressions and List Comprehensions

The *generator expression* `e` **`for`** `v` **`in`** `i` returns an iterable object of the values of the expression `e` (which may or may not involve `v`) when `v` is successively assigned the items in the iterable object `i`. For example, the expression `tuple(x * x for x in (1, 2, 3))` returns a tuple constructed as follows: each item in the tuple `(1, 2, 3)` is, in turn, assigned to the variable `x`. Then, the expression `x * x` is evaluated and the result stored. This process yields the tuple `(1, 4, 9)`. The expression `e` **`for`** `v` **`in`** `i` **`if`** `b` restricts the evaluation of the expression to items in `i` that satisfy the Boolean expression `b`. So, for example, the expression `tuple(x * x for x in (1, 2, 3, 4, 5) if x % 2 == 0)` returns the tuple `(4, 16)`.

List comprehensions store the results of generator expressions in a list: `[e` **`for`** `v` **`in`** `i]` or `[e` **`for`** `v` **`in`** `i` **`if`** `b]`.

0.9 Mutable Dictionaries

Instantiation and Initialization: A new dictionary is instantiatied and initialized using the expression { `<key 1>: <value 1>, <key 2>: <value 2>, ...` }. For example, the statement `d = { 'foo': 5, 'bar': 6 }` assigns, to the variable `d`, a dictionary that associates the key `'foo'` with `5` and `'bar'` with `6`.

Dictionaries can also be instantiatied and initialized with either a tuple or a list of key-value pairs. For example, the expression `d = ([('foo', 5), ('bar', 6)])` evaluates to a dictionary that associates the key `'foo'` with `5` and `'bar'` with `6`.

Key Selection: The expression `d[key]` evaluates to the value associated with key `key` in the dictionary `d`, or a `KeyError` if not present.

Key Assignment and Update: The expression `d[key] = val` either updates the value associated with the key `key` in the dictionary `d` to the new value `val` if the key is already present, or associates the key `key` with the value `val` if the key is not present.

Deletion: The statement **`del`** `d[key]` removes the key `key` and the associated value from `d`.

Key Membership: The expression `k` **`in`** `d` evaluates to `True` if the key `k` and an associated value is present in the dictionary `d`, and `False` otherwise.

Keys and Values: The expression `d.keys()` returns an iterable object that contains the keys in the dictionary `d` and the expression `d.values()` returns an iterable object that contains the values in the dictionary `d`.

0.10 Mutable Recursive Lists (RLists)

An abbreviated definition of the class `RList`, used to represent mutable recursive lists, is provided below.

```
class RList:
    """A recursive list consisting of a first element and the rest."""

    class EmptyRList:
        def __len__(self):
            return 0

        def __add__(self, other):
            if other is not RList.empty():
                return RList(other.first, self + other.rest)
            return RList.empty()

    # The one and ONLY instance of the EmptyRList class.
    the_empty_rlist = EmptyRList()
```

```
    @staticmethod
    def empty():
        """Returns the empty RList."""
        return RList.the_empty_rlist

    def __init__(self, first, rest=the_empty_rlist):
        self.first = first
        self.rest = rest

    @staticmethod
    def populate(*items):
        """Populates a new RList with the items provided."""
        # Body removed for sake of brevity.

    def __len__(self):
        """Returns the length of the current RList.
        The builtin function len(x) calls x.__len__()."""
        return 1 + len(self.rest)

    def __add__(self, other):
        """Returns the RList containing the elements of self followed by the
        elements of other. The operator x + y calls x.__add__(y)."""
        return RList(self.first, self.rest + other)
```

0.11 Environment Diagram Rules

Variable Lookup

1. Start with the current frame.
2. Look in the frame for a binding from the variable name to a value. If it is not in the current frame, search in the parent frame.
3. Once a value is found, return to the current frame. If the value is not found, signal an error.

Variable Assignment

1. Evaluate the right-hand side and continue drawing the environment diagram as you evaluate.
2. Create a binding in the current frame from the variable name to the value of the right-hand side, only if there is no binding for the variable in the current frame. Otherwise, update the binding in the current frame.
   Special case: If the variable is `nonlocal`, update the binding in a parent frame, if there is already a binding in a parent frame that is *not* the global frame. Otherwise, make a binding in the current frame. The keyword **`global`** is used if you want to update a binding in the global frame.

Function Creation

1. Create a double bubble. Point the left bubble to a list of parameters and the function body, and the right bubble to the current frame.
2. If the function was created for a **`def`** statement, create a binding from the function name to the double bubble in the current frame.

Function Call

1. Evaluate the operator and check that it produces a function.
   - If this was a built-in function, treat it like "magic", where the built-in function is defined in global and not shown.
   - Otherwise, locate the double bubble for the user-defined function.
2. Evaluate the operands (the arguments to the function).
3. Apply the function on its arguments.
   - If this was a built-in function, treat it like a black box and return the resulting value.
   - Otherwise, if it is a user-defined function:
     (a) Draw a new frame.
     (b) Create bindings in the new frame from the argument names to the values found earlier.
     (c) Have the new frame extend the frame that the right bubble points to.
     (d) Make this new frame the current frame (called "stepping into" the new frame) and evaluate the function body pointed to by the left bubble.
     (e) Once the function body is evaluated, return the value and restore the current frame to what it was before we stepped into the new frame.

Lists, Tuples and Dictionaries

Lists and tuples are both represented by box-and-pointer diagrams. We annotate the box-and-pointer diagram with `L` to the side for lists and `T` for tuples. Dictionaries are represented by ovals, with keys bound to values inside the oval.

0.12 OOP Implementation

```
def make_class(attributes, base_class=None):
    """Return a new class."""

    def get_value(name):
        """Get a class attribute."""
    def set_value(name, value):
        """Set a class attribute."""
    def new(*args):
        """Create an instance of the class."""

    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls

def make_instance(cls):
    """Return a new object instance."""

    def get_value(name):
        """Get an instance attribute."""
    def set_value(name, value):
        """Set an instance attribute."""

    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance

def bind_method(value, instance):
    """Return value or a bound method if value is callable."""

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args."""
```

0.13 Streams

`Stream(first, compute_rest, empty=False)`: Constructs a stream with the given first element `first` and a function to compute the rest of the stream `compute_rest`. `empty` is true if the stream is empty.

`s.first`: Returns the first of the Stream `s`.

`s.rest`: Returns the rest of the Stream `s`.

`Stream.the_empty_stream`: The empty stream.

`show_stream(strm, count=10)`: Compute and print out the first `count` number of items (default 10).

`add_streams(s1, s2)`: Returns the stream where each element is the sum of the elements at the corresponding positions in the two streams. The result is as long as the shorter of the two streams.

`mul_streams(s1, s2)`: Returns the stream where each element is the product of the elements at the corresponding positions in the two streams. The result is as long as the shorter of the two streams.

`stream_map(fn, *strms)`: Map a function over a (collection of) stream(s).

`stream_filter(pred, strm)`: Filter items from a stream using the predicate `pred`.

`stream_interleave(s1, s2)`: Interleave the items from the two streams provided.

`make_stream_of(n)`: Returns an infinite stream of the same number `n`.

`integers_starting_from(n)`: Returns a stream of integers starting from the number `n`.

0.14 Logic Programming in PyGic

Appending two lists:

```
fact append(<>, ?z, ?z)
rule append(<?x | ?u>, ?v, <?x | ?w>):
    append(?u, ?v, ?w)
```

Reversing a list:

```
fact reverse(<>, <>)
rule reverse(<?first | ?rest>, ?rev):
    reverse(?rest, ?rest_rev)
    append(?rest_rev, <?first>, ?rev)
```

Checking if two lists are equal:

```
fact equal_lists(<>, <>)
rule equal_lists(<?x | ?rest1>, <?x | ?rest2>):
    equal_lists(?rest1, ?rest2)
```

Checking if an element is in a list:

```
fact member(?x, <?x | _>)
rule member(?x, <_ | ?rest>):
    member(?x, ?rest)
```

0.15 Scheme

cons: Constructs a pair where the first argument is the first item in the pair, and the second argument is the second item. For example:

```
STk> (cons 1 2)
(1 . 2)              ; The dot stands for pair: this is the pair 1, 2.
STk> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

list: Constructs a list from all of its arguments. For example:

```
STk> (list 1 2 3)
(1 2 3)
```

`car`, `cdr`: Return the first item and the second item in a Scheme pair respectively. For example:

```
STk> (car '(1 . 2))
1
STk> (car '(1 2 3))
1
STk> (cdr '(1 . 2))
2
STk> (cdr '(1 2 3))
(2 3)
```

`if`: Similar to the `if` statement in Python. It evaluates the second argument if the first argument evaluates to true. Otherwise, it evaluates the third argument.

```
STk> (if (= 2 2) (+ 2 3) (- 5 1))
5
STk> (if (= 2 3) (/ 1 0) (+ 1 1))   ; does not error
2
```

`define`: Defines either a Scheme function or a variable. It takes two (or more) arguments. If the first argument is a symbol, then Scheme defines a variable. If the first argument is a Scheme list, then the first element is the name of the function, and the rest of the elements are the names of the arguments. The rest of the arguments to the `define` form are then evaluated when the function is called.

```
STk> (define (f x) (* x 2))
STk> (f 3)
6
STk> (define g 2)
STk> g
2
```

True and False: In Scheme, everything is considered true (or `#t`), except for `#f`.