This exam begins for you when you open (or peek inside) this packet. It ends at the start of class on Monday 2008 November 3. Between those two times, you may work on it as much as you like. I recommend that you get started early and work often. The exam is open-book and open-note, which means, precisely:

- You may freely consult all of this course's material: the Miller and Ranum textbook, your class notes, your old homework and exam, and the materials on the course web site. For example, if the course web site links to an applet at another site, then you may view that applet, but you may not view any other materials on that site. If you missed a class and need to copy someone else's notes, do so before either of you begins the exam.
- You may not consult any other papers, books, microfiche, film, video, audio recordings, Internet sites, etc. You may use a computer for these three purposes: viewing the course web site materials, editing and running Python programs, and e-mailing with me. You may not share any materials with another student.
- You may not discuss the exam in any way (spoken, written, pantomime, semaphore, etc.) with anyone but me until everyone has handed in the exam — even if *you* finish earlier. During the exam you will inevitably see your classmates around campus. Please refrain from asking even seemingly innocuous questions such as "Have you started the exam yet?" If a statement or question conveys any information, then it is not allowed; if it conveys no information, then you have no reason to make it.

During the exam you may want to ask me questions. You may ask clarifying questions for free. If you believe that the statement of a problem is wrong, then you should certainly ask for clarification. You may also ask for hints, which cost you some points, to be decided by me as I grade your paper. I will not give you a hint unless you unambiguously request it. I will try to check my e-mail over the weekend, but there is always some lag, and discussing technical material over e-mail is not easy.

Your solutions should be thorough, self-explanatory, and polished. When asked to "explain" something, explain at a level suitable for a classmate, and explain thoroughly enough that I am convinced that you fully understand the material. If you cannot solve a problem, write a *brief* summary of the approaches you've tried; this may earn partial credit. Submit your solutions electronically, as usual.

Partial credit is often awarded. Exam grades are loosely curved — by this I do not mean that there are predetermined numbers of As, Bs, Cs to be awarded, but rather that there are no predetermined scores required for grades A, B, C.

Good luck!

AVL Trees

For this problem we will use the following specification for the ADT `AVLTree`. First, `AVLTree` is a subclass of the ADT `BinarySearchTree` defined in class. The rest of the ADT is:

- `AVLTree(key, value)` creates an `AVLTree` node storing the given key-value pair. It is assumed that all keys are numbers.
- `getRootKey()` and `setRootKey(key)` access the key stored in the tree node.
- `put(key, value)` puts the key-value pair into the tree. If the key is already in the tree, then its value is updated to the new value. This method returns the root node of the tree.
- `get(key)` returns the value associated to the given key. If the key is not in the tree, then it returns `None`.
- `delete(key)` deletes the key (and its associated value) from the tree. If the key is not in the tree, then it does nothing. This method returns the root node of the tree.

In the file dicthasbst.py on the course web site, the `Dictionary` ADT is implemented using `BinarySearchTree`. Suppose that you wrote another implementation of `Dictionary` using `AVLTree` in a file dicthasavl.py. (I am not asking you to write this implementation.) In a plain text file avl.txt, type your answers to these questions.

1. How would the code for the dicthasavl.py implementation of `Dictionary` differ from that of the dicthasbst.py implementation?
2. In detail, compare the efficiencies of the two implementations.
3. If another programmer, who was using the dicthasbst.py implementation, wanted to switch to the dicthasavl.py implementation, then what instructions would you give her about making the transition?
4. Why do `put()` and `delete()` return the root of the tree?

Priority Queues

A priority queue is a data structure for storing key-value pairs, as follows. A key is called a *priority*, and it is a number. A value can be any Python object. The priority queue acts much like a queue, except that high-priority pairs are always dequeued before low-priority ones. Pairs of equal priority are handled in a FIFO manner, as in a regular queue. You partially implemented a priority queue on Exam 1. To be precise, the four methods of the `PriorityQueue` ADT are:

- `PriorityQueue()` returns an empty priority queue.
- `enqueue(priority, value)` puts the priority-value pair into the queue.
- `dequeue()` deletes and returns the highest-priority value (with ties broken FIFO). If there are no pairs in the queue, then it does nothing.

- `size()` returns the number of pairs currently in the queue.

In a file priorityqueue.py, implement `PriorityQueue` using a binary heap. I advise you to follow the binary heap implementation in our textbook, but you will need to make various modifications. Thoroughly comment your code so that a reader can understand how it works (or is intended to work). Also, at the bottom of the file include test code that demonstrates that your priority queue works.

Parsing

In class and in homework we've studied scanning, parsing, and evaluation of fully parenthesized algebraic expressions (FPAEs). In this problem you will parse a different language, called SLAM, that is simpler than FPAEs in some ways and more complicated in others. There are several kinds of SLAM expressions:

- There are numbers, such as `3` and `-1.16`.
- There are built-in mathematical operators/functions; they are `+`, `-`, `*`, `/`.
- There are user-defined functions, such as `foo` or `push`, that the user may design to take any number of arguments.
- There are compound expressions of the form

  $$(\; func \; arg_1 \; arg_2 \; \ldots arg_n \;)$$

  where $func$ is any built-in or user-defined function/operator, and all of the arguments arg_i are either numbers or compound expressions.

Notice that compound expressions are always in prefix form. Here are some examples, along with comparisons to standard mathematical/Python notation:

- `( + 3 5 )` is equivalent to the infix expression `3 + 5`.
- `( + 3 5 -11 )` is equivalent to `3 + 5 + -11`.
- `( / 3 5 -11)` is equivalent to `3 / 5 / -11`, which would usually be written `( 3 / 5 ) / -11`.
- `( * ( / 3 5 ) (- 4 7 ) )` is equivalent to `(3 / 5) * (4 - 7)`.
- If `quack` is a user-defined function that takes three arguments, then `( quack 3 (+ 2 7) -5 )` is equivalent to `quack(3, 2 + 7, -5)`.

SLAM expressions are scanned (tokenized) in exactly the same way as FPAEs. Your first job is to write a parser, that takes in a list of tokens and generates a parse tree (as you did in homework). Notice that the tree may not be a binary tree. Write this parser in a file called slam.py. Also implement these Python functions:

- `factorial(n)` computes the factorial function, for any nonnegative integer n.
- `power(a, n)` computes a^n for any number a and any nonnegative integer n.

Also write an evaluator, that takes in a SLAM parse tree and evaluates it. Your evaluator should be able to handle all built-in SLAM operators/functions, as well as the user-defined functions `factorial` and `power`. For example, parsing and evaluating

```
( factorial ( power ( * ( + 6 -3 ) 1 ) 2 ) )
```

should return 362880. Write test code to demonstrate that your parser and evaluator work correctly. Also make sure that your code is commented/documented well.