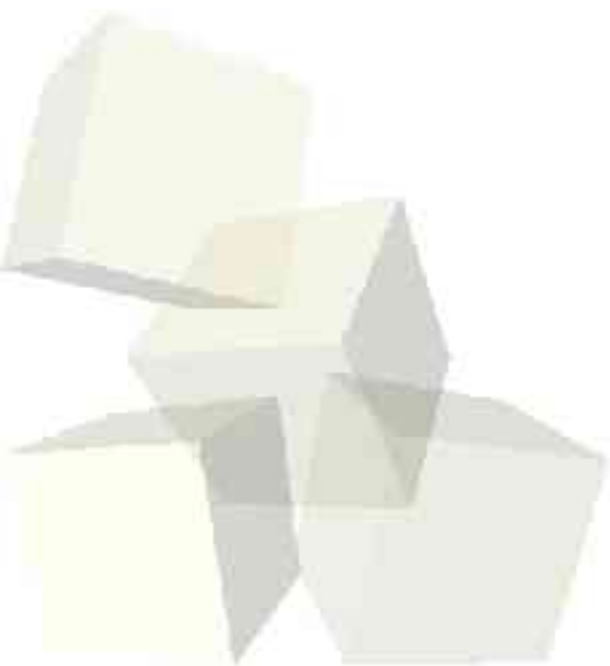




Procedures and Recursion 1

9-1-2004





Opening Discussion

- What did we talk about last class?
- Only one person actually did the whole minute essay in the way I was hoping for. Let's look at what it requires.
- As you should have seen, the first assignment has been posted on the website.





Functions

- Being a functional language, Scheme is heavily based on the concept of functions. Last class we saw some built-in functions like `+`, `car`, and `cons`. We also saw how Scheme uses prefix notation. Instead of doing `f(x,y)` we do `(f x y)`.
- Functions in Scheme are called lambda expressions going back to their basis on the lambda calculus. We can create on this way:
 - ♦ `(lambda (parameter ...) body)`



Calling a Function

- Any function can be called the same way we did with the functions we learned about last class. So we could use a function that picks the second element of a list like this:
 - ♦ `((lambda (list) (car (cdr list))) '(4 5 6))`
- This would return 5 to us. Obviously is it less than ideal to write functions every time we need them. Instead, we need a way to bind them to names like we did yesterday with `define`.




Defining Names for Functions

- There are two ways to bind a lambda function to a name. The most obvious is like this:
 - ♦ (define *name* (lambda (*param ...*) *body*))
- This is basically the usage of define we saw before with the value being a lambda expression.
- There is also a shorthand version that doesn't explicitly include the lambda keyword:
 - ♦ (define (*name param ...*) *body*)



More Predefined Functions

- Scheme provides a shorthand for successive applications of car and cdr. Between the c and r you can place up to 4 a's or d's. So caar is the car of the car and cddr is the cdr of the cdr.
 - The list function takes any number of arguments and returns a list that has them in order.
- 



Conditional Expressions

- Scheme provides two methods of conditional execution. The simpler of the two is pretty much like the if statement that you are used to from other languages, but with a semantic exception.
 - ♦ *(if condition consequence alternative)*
- The difference comes from the fact that we don't have side effects. This if is more like the ternary operator `?:` in C/C++/Java. It “returns” the value of either the consequence or alternative. The latter can be left out.



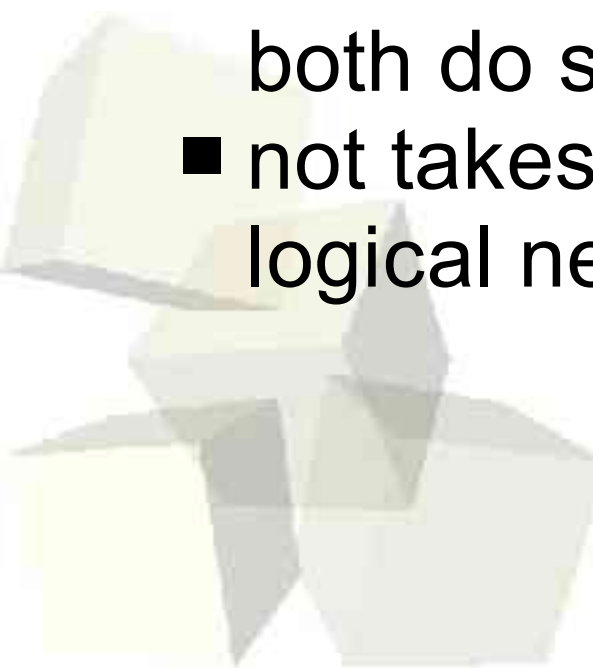
The cond Expression

- Scheme also provides a way of choosing between multiple options. This is the cond expression.
- (cond
- (*condition1 expression1*)
- (*condition2 expression2*)
- ...
- (*conditionN expressionN*)
- (*else alternative*))
- This returns the value of the expression following the first true condition.



Boolean Expressions

- We have seen predicates that could be used for the conditions in if and cond. Sometimes we want to build more complex logic. For the we have the operators and, or, and not.
- and and or can take multiple arguments and both do short-circuit evaluation.
- not takes a single argument and returns the logical negation of it.



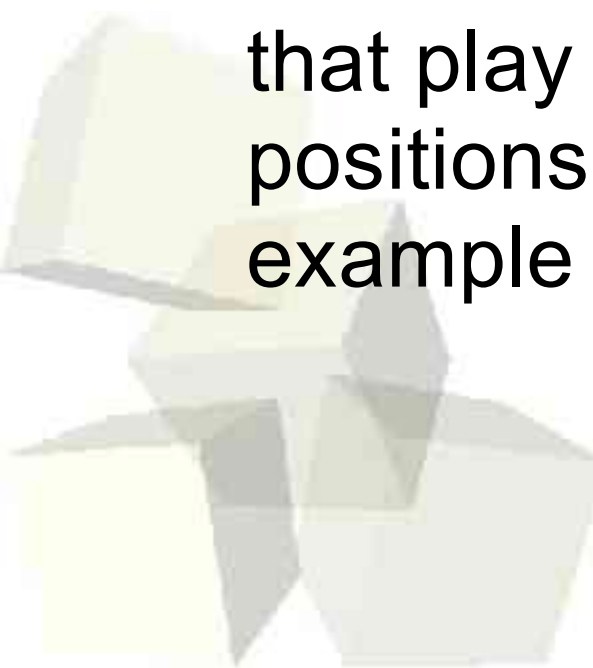


Recursion

- Recursive functions are functions that call themselves. We can use these functions in most language, but many people shy away from them, even when they are the easiest way to do things. Because of the lack of side effects, recursion is the primary way we get anything to happen multiple times in Scheme and other functional languages.
- Instead of a loop variable that is altered, we have a function that calls itself with an altered argument and the argument acts like the loop variable.



More Recursion

- Any recursive function must have a conditional to terminate the recursion. There must be some situation that prevents the function from calling itself.
 - In Scheme we can write recursive functions that play with lists when we don't know exact positions of things. Let's write some example functions that do this.
- 



Minute Essay

- Write a function which, given a list, will return a list that contains every other element of the list.
- How comfortable are you with recursion? It can be a tough topic, but you can take comfort that we have at least two more chapters on it.

