Introduction

The midterm exam will be given during the class time (8:30 to 9:20 AM) on Wednesday, November 6. You will have 50 minutes for the exam.

The exam will be closed book, no notes, no calculators. Review the homework questions and the class notes, and read this document carefully, and you will do fine on the exam.

The exam is based on the lectures and the homework. The questions will be similar to the questions on the homework with several small programming problems.

Scheme

Programming that makes extensive use of assignment is known as *imperative programming*. The order of assignments changes the operation of the program because the state is changed by assignment. Programming without the use of assignment statements is known as *functional programming*. In such a language, all procedures implement well-defined mathematical functions of their arguments whose behavior does not change. Scheme is heavily oriented towards functional style, and we did not cover the imperative aspects of the language.

Expressions

Expressions in Scheme are either simple or compound. Simple expressions are basic elements such numeric constants and variable names (symbols). Numbers can be integers, rational numbers or floating point. Integers and rational numbers are exact; floating point numbers are inexact. Compound expressions are either a combination or a special form. Combinations take the form (operator operand operand …), where each term can itself be simple or compound. Scheme evaluates a combination by evaluating the operator and each operand, and then applying the operator to the operands. Special forms look the same as combinations, but the operator is recognized by Scheme as a special keyword, and the evaluation of the operands follows the rules for that particular keyword. (This is the reason these are “special” forms; they have their own individual rules for evaluation.)

An expression that yields a true or false value is called a predicate.

Procedures

Procedures are the way to group statements into a single unit. Procedures can be defined using the define special form, or the lambda special form.

Procedures can be defined within the lexical scope of another procedure, in which case they are local to that procedure and can only be accessed from within the procedure (ie, the local name is not visible outside the enclosing procedure block). Similarly, the scope of the names of the formal parameters of each local procedure is the body of that procedure.

Procedures can be defined with zero or more required parameters, plus provision for a variable number of parameters to follow. If the optional parameters are present in a call to the procedure, they are collected and provided to the called procedure as a single list.

Variables defined in the enclosing scope can be referenced from the enclosed procedures, as long as the identifier has not been redefined locally. This is referred to as lexical scoping. Free variables (those that are not bound by the parameter list or a local define) are taken to refer to bindings made by enclosing procedure definitions. The bindings are looked up in the environment in which the procedure was defined.

Using the lambda special form, we can write procedures that operate on other procedures. This is known as applicative programming.

Compound data

In order to build compound structures we need a way to combine elements and refer to them as a single blob. We can write a lambda expression that combines one or more expressions and the resulting combination is a procedure. We can write a cons expression that ties two data elements together and the resulting combination is a pair.

(cons a b) takes a and b as args, returns a compound data object that contains a and b as its parts. We can extract the two parts with accessor functions car and cdr. (pair? z) is true if z is a pair. (null? z) is true if z is nil (ie, the empty list, a null pointer).

You should be able to produce a drawing showing the elements of a compound data structure, given a cons expression or a quoted list. Similarly, given a drawing, you should be able to give and expression that would create that data structure.

The convention for lists in Scheme uses a backbone of pairs, with one pair object per element in the list. The element is the car of the pair; the rest of the list is the cdr of the pair. The last pair has the empty list (null) as its cdr. The elements of a list can also be lists, as for example in our mobiles and symbolic expressions in the homework.

Recursion works well with list structures, and you should be able to write simple list related procedures. Some of the design patterns that we used include the following. “cdr down” in which we process each element in turn by processing the first element in the list, then recursively processing the rest of the list using cdr and a recursive call. “cons up” in which we build a list to return to the caller piece by piece as we go along through an input list.

Input / Output

Input in Scheme can be used to read from the console, a file, or a string. Similarly, output can go to the display, a file, or a string. An input port can be read as a character stream (using read-char) or as an object stream (using read). An output port can be written in human oriented format (using display or write-char) or machine oriented format (using write). Most implementations provide procedures to interface with the host file system, although they are not part of the Scheme standard.

Special Forms

(define (⟨*name*⟩ ⟨*formal params*⟩) ⟨*body*⟩)	Define and name a procedure.
(cond ⟨*clause*$_1$⟩ ⟨*clause*$_2$⟩ ... ⟨*clause*$_n$⟩)	Choose from one of several possible clauses. Each clause is of the form (⟨*predicate*⟩ ⟨*expression*⟩). The last clause can be of the form (else ⟨*expression*⟩).
(if ⟨*predicate*⟩ ⟨*consequent*⟩ ⟨*alternate*⟩) **(if** ⟨*predicate*⟩ ⟨*consequent*⟩)	Choose to do or not do a consequent expression, or choose between a consequent or an alternate expression.
(and ⟨*e*$_1$⟩ ⟨*e*$_2$⟩... ⟨*e*$_n$⟩) **(or** ⟨*e*$_1$⟩ ⟨*e*$_2$⟩... ⟨*e*$_n$⟩) **(not** ⟨*e*⟩)	Logical composition.
(lambda (⟨*formals*⟩) ⟨*body*⟩)	A lambda expression evaluates to a procedure
(let ((⟨*var*$_1$⟩ ⟨*exp*$_1$⟩) (⟨*var*$_2$⟩ ⟨*exp*$_2$⟩)) ⟨*body*⟩)	Special form that sets all the *var*$_i$ to the corresponding *exp*$_i$, then executes the body.
(let* ((⟨*var*$_1$⟩ ⟨*exp*$_1$⟩) (⟨*var*$_2$⟩ ⟨*exp*$_2$⟩)) ⟨*body*⟩)	Special form that sets all the *var*$_i$ to the corresponding *exp*$_i$, then executes the *body*. *exp*$_{i+1}$ can refer to all preceding *var*$_i$.
(begin ⟨*exp*$_1$⟩ ⟨*exp*$_2$⟩ … ⟨*exp*$_n$⟩)	Evaluate the *exp*$_i$ in sequence from left to right
(quote ⟨datum⟩) *or* **'⟨datum⟩**	Tells Scheme to treat the given expression as a data object directly, rather than as an expression to be evaluated.