

Take-home Midterm
CS 503 2006
Due: 5pm, Tues Oct 31

Overview

This midterm is designed to test the skills and concepts that we've covered so far this semester with easy examples that you can still learn a little bit from, and that we can discuss further in class Nov 2. You are welcome to make reference to any of the materials we've considered in class, including the sample programs posted here, Scheme help, and even *How to Design Programs* and *Wikipedia* (though you shouldn't need them). You can also send questions about questions and Scheme bugs to Matthew—if they're just stupid errors (e.g., use = not eq?) you'll get a response; if they point to a useful clarification to any of the questions an answer will be posted to the web; but you'll get a note declining to answer if the question is something you have to find out for yourself, e.g., it concerns the insight the question tests. Except for the exceptions just mentioned, your answers must be exclusively your own work. Email Matthew a scheme file with the answers Tuesday Oct 31. An easy way to write this up is to extend the posted Scheme template file.

Problem 1.

This question is a general exploration of **lists and representations**.

You can use a list of numbers to represent a polynomial. Each successive element in the list gives the coefficient of the variable x raised to the next higher power. For example, `(1 0 1 4)` corresponds to the polynomial $1 + x^2 + 4x^3$.

First, write a function `eval-poly` which takes a polynomial `p` and a value `v` and evaluates the polynomial `p` at the point $x = v$. Examples:

- `(eval-poly '(1 1 1) 3)` is 13...(1+3+9)
- `(eval-poly '(2 6) 2)` is 14...(2+12)
- `(eval-poly '(3 5 7) 1)` is 15...(3+5+7)

You can write `eval-poly` using the structure of the polynomial. But to do so, you need to come up with a mathematical expression relating the polynomial represented by `p` to the polynomial represented by `(cdr p)`.

Second, write a function `add-poly` which takes two polynomials `p1` and `p2` and computes their sum as a polynomial. For example:

- `(add-poly '(3 5 7) '(2 6))` is `(5 11 7)`
- `(add-poly '(2 6) '(3 5 7))` is `(5 11 7)`

You can also write `add-poly` using the structure of the polynomial. Here the “trick” is that the polynomials do not have to be the same length.

Third, write a function `diff-poly` which computes the derivative of a polynomial `p` with respect to its variable. For example, the derivative of $1 + x^2 + 4x^3$ with respect to its variable x is $2x + 12x^2$. So `(diff-poly '(1 0 1 4))` should give a polynomial equivalent to `(0 2`

12). Actually you may find that for this problem it is easier to return the answer `'(0 2 12 0)`, which is also correct.

You should again write `diff-poly` directly using the structure of the polynomial. You will have to rely on the same mathematical view of the polynomial you developed for `eval-poly`: how does the polynomial represented by `p` relate to the polynomial represented by `(cdr p)`? Your answer will then involve the product rule and `add-poly`.

Fourth, write another function `diff-poly2` which computes the derivative. Instead of using the product rule, `diff-poly2` should compute the derivative directly using a helper function `diff-aux`. The function `diff-aux` takes two arguments, a polynomial p and a whole number n . It computes the derivative of the polynomial $x^n * p$. The result is returned as a list r which omits a factor of x^{n-1} . In other words, the derivative of $x^n * p$ is $x^{n-1} * r$. Some examples should make this clearer.

- `(diff-aux '(0 1 4) 1)` is `'(0 2 12)`
- `(diff-aux '(1 4) 2)` is `'(2 12)`
- `(diff-aux '(4) 3)` is `'(12)`

Problem 2.

This problem is an exploration of **XML** and **Programs**.

Write an interpreter for the programming language **UFO**. The language **UFO** describes the actions of a spacecraft as it moves around the universe. Programs in **UFO** are written in **XML** within the tag `ufo`. The result of the program is the $(x\ y\ z)$ position in space that the spacecraft has moved to from the origin $(0\ 0\ 0)$ by following the specified actions. There are six basic commands in **UFO**

- `<east/>` - moves the spacecraft one parsec forward along the x axis.
- `<west/>` - moves the spacecraft one parsec back along the x axis.
- `<north/>` - moves the spacecraft one parsec forward along the y axis.
- `<south/>` - moves the spacecraft one parsec back along the y axis.
- `<up/>` - moves the spacecraft one parsec forward along the z axis.
- `<down/>` - moves the spacecraft one parsec back along the z axis.

In addition there are two special commands:

- `<home/>` - instantly returns the spacecraft to the origin, position $(0\ 0\ 0)$.
- `<twice>...</twice>` - runs the embedded **UFO** program two times in succession.

Here is a convenient breakdown of your **UFO** interpreter.

- `(eval-step step start)` - return the position that the `ufo` gets to by carrying out `ufo-action step` from the position `start`. Each step should be represented as an **SXML** node as would be read in from an **XML** statement of a **UFO** program.

A couple hints: if you use a `let` to define `x`, `y`, and `z`, your code will be much cleaner. The fact that the `step` may be `twice` means that `eval-step` and the next function, `eval-program`, will be *mutually recursive*...each will call the other.

- `(eval-program steps start)` - return the position that the `ufo` gets to by carrying out each of the steps in the list `steps` in order, starting from position `start`.
- `(eval-ufo struct)` - if `struct` is the **SXML** representation of a **UFO** file, this function returns the position that the embedded program unit leaves the `ufo` at, if it starts at the origin.

Some examples. This program:

```
<ufo><up/><up/><home/><down/></ufo>
```

evaluates to $(0\ 0\ -1)$. This program:

```
<ufo><north/><twice><up/><east/></twice></ufo>
```

evaluates to $(2\ 1\ 2)$.

Problem 3. This gives you a flavor of abstract programming and tests your ability to think about and talk about programs you already have.

Consider the code below:

```
(define (are-all? pred? here)
  (and (pred? here)
       (if (pair? here) (are-all-elts? pred? here) #t)))

(define (are-all-elts? pred? here)
  (if (pair? here)
      (and (are-all? pred? (car here))
           (are-all-elts? pred? (cdr here)))
      #t))
```

Write a brief statement of what these functions do, and how they do it. Your description should answer: what type of value does `pred?` have to be given? What type of value does `here` have to be given? What type of value does `are-all?` return? What different values does it return? And under what circumstances does it return those values? How does the computation proceed? Your answer should take the form of a comment around the `are-all?` and `are-all-elts?` function.

Consider a binary tree of numbers that is made up of two kinds of nodes. A leaf node is just a number—it satisfies the builtin Scheme function `number?`. An internal node is a 3-element list whose first element is a number and whose second and third elements are binary trees. Explain how you can combine `are-all?` and a simpler predicate `ok-in-tree?` to test whether an arbitrary Scheme structure is a binary tree. When should function `(ok-in-tree? n)` return true? Your answer should take the form of commented Scheme functions for `ok-in-tree?` and `is-binary?`.

Remember: you can test your hypotheses by interacting with Scheme. Here are some places to get started: for what `x` will `(are-all-elts? number? x)` be true? for what `x` will `(are-all? list? x)` be true?