Lecture 13: 
Introduction to the big project
Lists
## Schedule

<table>
<thead>
<tr>
<th></th>
<th>Date</th>
<th>Lecture: Lists, and introduce the big project</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Nov 20-24</td>
<td>Lab: Lists; start on the project</td>
</tr>
<tr>
<td>14</td>
<td>Nov 27–Dec 1</td>
<td>Lecture: Lists; Computer Science at UCB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Work on the project</td>
</tr>
<tr>
<td>15</td>
<td>Dec 4 – Dec 8</td>
<td>Lecture: Course summary; practice problems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Finish project; due at end of week</td>
</tr>
<tr>
<td></td>
<td>Dec 16th</td>
<td>Final Exam (Saturday, 8am-11am)</td>
</tr>
</tbody>
</table>
Any questions?
The Big Project

• Three possible projects:
  - Database
  - Yukon
  - Blocks World

• You can, and should, work in partnerships
• You will have three weeks to work on this (it is due on the last lab)
• Worth 15% of your final grade
Project Check-offs

• There are 3 checkoffs

  You need to do them on time in order to get credit for the project

3. Tell your TA which project you will do and who you will do it with

4. Show your TA that you have accomplished something. S/he will comment.

5. Show that you have most of the work done: your TA will run your code.
Due dates on the final project

<table>
<thead>
<tr>
<th>Tues/Wed</th>
<th>Thur/Fri</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Nov 21/22) Introduction</td>
<td>(Nov 23-24) Thanksgiving</td>
</tr>
<tr>
<td>(Nov 28-29) Checkoff 1</td>
<td>(Nov 30 – Dec 1) Checkoff 2</td>
</tr>
<tr>
<td>(Dec 5-6) Checkoff 3</td>
<td>(Dec 8, Friday) Due (at midnight)</td>
</tr>
</tbody>
</table>
Only two more lectures (after this one)...

What would you like to do?

- Hear about the CS major, and other courses…
- Do exam-type problems…
- Review…
Let's see the projects in action
What issues matter

• Does it work?
  - This is a primary grading standard…

• Programming style
• Reading specifications carefully
• Error checking inputs (especially in the database project)
• Adequate testing
• Code reuse (again, with the database)
Working in partnerships

• Highly recommended!
  - For those of you continuing with CS, you'll be doing this for many future projects

• Won't be faster, necessarily
  - While you are less likely to get stuck, there will be a lot of communication necessary

• A big benefit will be with testing

• Remember, only one grade is given…
  - this grade will be the same, whether the project is a solo or a partnership
Functional Programming

• In CS3, we have focused on programming without *side-effects*.
  - All that can matter with a procedure is what it returns
  - In other languages, you typically:
    - Perform several actions in a sequence
    - Set the value of a variable – and it stays that way
  - All of this is possible in Scheme.
• With *Blocks World* and *Yukon* you will need to display information.
  - *Simply Scheme* chapter 20 is nice summary.
  - And, all the projects have file input/output routines that you don't need to "understand", as well as user input routines.
Data structures

• The format of data used in these projects in a central feature
  - A "data structure" (abstract data type) is a specification of that format. Here, generally, lists of lists (of lists).
  - Accessors and constructor allow for modularity: letting parts of a program work independently from other parts.
Strings versus words

• One useful data structure is a string
  - Strings are surrounded by double quotes when printed.
  - Strings are a native type in Scheme.

• In CS3, you used words (sometimes sentences) to present names and other output to the user.
  - In the "real world", strings are used.
Lists
Lists

- Lists are containers, like sentences, where each element can be anything

  - Including, another list

    (((beatles 4) (beck 1) ((everly brothers) 2) ... )

    (((california 55) (florida 23) ((new york) 45) )

    (#f #t #t #f #f ... )
# Sentences(words) vs lists: constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| **cons**    | Takes an element and a list  
             | Returns a list with the element at the front, and the list contents trailing |
| **append**  | Takes two lists  
             | Returns a list with the element of each list put together |
| **list**    | Takes any number of elements  
             | Returns the list with those elements |
| **sentence**| Takes a bunch of words and sentences and puts "them" in order in a new sentence. |
What is the point of cons?

While append and list make more sense, cons is more closely tied to use in recursion:

```scheme
(define (sent-square-all sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
           (sent-square-all (bf sent)))))

(ssa '(1 2 3)) \rightarrow (se 1 (se 4 (se 9 '())))

(define (list-square-all lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (list-square-all (cdr lst)))))

(lsa '(1 2 3)) \rightarrow (cons 1 (cons 4 (cons 9 '())))
```
## Sentences(words) vs lists: selectors

<table>
<thead>
<tr>
<th>Car</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the first element of the list</td>
<td>Returns the first word (although, works on non-words)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cdr</th>
<th>Butfirst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a list of everything but the first element of the list</td>
<td>Returns a sentence of everything but the first word (but, works on lists)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Last</th>
<th>Butlast</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Sentences(words) vs lists: HOF</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>map</strong></td>
<td><strong>every</strong></td>
</tr>
<tr>
<td>Returns a list where a func is applied to every element of the input list.</td>
<td>Returns a sentence where a func is applied to every element of an input sentence or word.</td>
</tr>
<tr>
<td>Can take multiple input lists.</td>
<td></td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td><strong>keep</strong></td>
</tr>
<tr>
<td>Returns a list where every element satisfies a predicate.</td>
<td>Returns a sentence or word where every element satisfies a predicate</td>
</tr>
<tr>
<td>Takes a single list as input</td>
<td></td>
</tr>
<tr>
<td><strong>reduce</strong></td>
<td><strong>Accumulate</strong></td>
</tr>
<tr>
<td>Returns the value of applying a function to successive pairs of the (single) input list</td>
<td>Returns the value of applying a function to successive pairs of the input sentence or word</td>
</tr>
</tbody>
</table>
CS3: Introduction to Symbolic Programming

Lecture 13: Introduction to the big project
Lists

Fall 2006 Nate Titterton
nate@berkeley.edu
# Schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Date Range</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Nov 20-24</td>
<td>Lecture: Lists, and introduce the big project</td>
<td>Lab: Lists; start on the project</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Nov 27–Dec 1</td>
<td>Lecture: Lists; Computer Science at UCB</td>
<td>Lab: Work on the project</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Dec 4 – Dec 8</td>
<td>Lecture: Course summary; practice problems</td>
<td>Lab: Finish project; due at end of week</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dec 16&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Final Exam (Saturday, 8am-11am)</td>
<td></td>
</tr>
</tbody>
</table>
Midterm #2

Any questions?
The Big Project

• Three possible projects:
  - Database
  - Yukon
  - Blocks World

• You can, and should, work in partnerships

• You will have three weeks to work on this (it is due on the last lab)

• Worth 15% of your final grade
Project Check-offs

• There are 3 checkoffs
  You need to do them on time in order to get credit for the project

3. Tell your TA which project you will do and who you will do it with
4. Show your TA that you have accomplished something. S/he will comment.
5. Show that you have most of the work done: your TA will run your code.
# Due dates on the final project

<table>
<thead>
<tr>
<th>Tues/Wed</th>
<th>Thur/Fri</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(Nov 21/22)</em></td>
<td><em>(Nov 23-24)</em></td>
</tr>
<tr>
<td>Introduction</td>
<td>Thanksgiving</td>
</tr>
<tr>
<td><em>(Nov 28-29)</em></td>
<td><em>(Nov 30 – Dec 1)</em></td>
</tr>
<tr>
<td>Checkoff 1</td>
<td>Checkoff 2</td>
</tr>
<tr>
<td><em>(Dec 5-6)</em></td>
<td><em>(Dec 8, Friday)</em></td>
</tr>
<tr>
<td>Checkoff 3</td>
<td>Due (at midnight)</td>
</tr>
</tbody>
</table>

Spring 2006 CS3: 6
Only two more lectures (after this one)…

What would you like to do?

- Hear about the CS major, and other courses…
- Do exam-type problems…
- Review…
Lets see the projects in action
What issues matter

• Does it work?
  - This is a primary grading standard…

• Programming style
• Reading specifications carefully
• Error checking inputs (especially in the database project)
• Adequate testing
• Code reuse (again, with the database)
Working in partnerships

• Highly recommended!
  - For those of you continuing with CS, you'll be doing this for many future projects

• Won't be faster, necessarily
  - While you are less likely to get stuck, there will be a lot of communication necessary

• A big benefit will be with testing

• Remember, only one grade is given…
  - this grade will be the same, whether the project is a solo or a partnership
Functional Programming

• In CS3, we have focused on programming **without side-effects**.
  - All that can matter with a procedure is what it returns
  - In other languages, you typically:
    - Perform several actions in a sequence
    - Set the value of a variable – and it stays that way
  - All of this is possible in Scheme.
Printing, and sequencing

• With Blocks World and Yukon you will need to display information.
  - Simply Scheme chapter 20 is nice summary.
  - And, all the projects have file input/output routines that you don't need to "understand", as well as user input routines.
Data structures

• The format of data used in these projects in a central feature
  - A "data structure" (abstract data type) is a specification of that format. Here, generally, lists of lists (of lists).
  - Accessors and constructor allow for modularity: letting parts of a program work independently from other parts.
Strings versus words

• One useful data structure is a string
  - Strings are surrounded by double quotes when printed.
  - Strings are a native type in Scheme.
• In CS3, you used words (sometimes sentences) to present names and other output to the user.
  - In the "real world", strings are used.
Lists
Lists

- Lists are containers, like sentences, where each element can be anything

  - Including, another list

    ((beatles 4) (beck 1) ((everly brothers) 2) ... )

    ((california 55) (florida 23) ((new york) 45) )

    (#f #t #t #f #f ... )
## Sentences(words) vs lists: constructors

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cons</strong></td>
<td>Takes an element and a list</td>
</tr>
<tr>
<td></td>
<td>Returns a list with the element at the front, and the list contents trailing</td>
</tr>
<tr>
<td><strong>append</strong></td>
<td>Takes two lists</td>
</tr>
<tr>
<td></td>
<td>Returns a list with the element of each list put together</td>
</tr>
<tr>
<td><strong>list</strong></td>
<td>Takes any number of elements</td>
</tr>
<tr>
<td></td>
<td>Returns the list with those elements</td>
</tr>
<tr>
<td><strong>sentence</strong></td>
<td>Takes a bunch of words and sentences and puts &quot;them&quot; in order in a new sentence.</td>
</tr>
</tbody>
</table>

Some "tips":

With append, you erase the middle parentheses

```
(append '( a   b    c    ) (   d  (e)  f )      )
 ;;                        | |
 ;;                        X X
 ->    ( a   b    c          d   (e) f )
```

With list, you add parentheses around the arguments

```
(list      '( a b c )   (d  (e)  f)   (g h i)        )
 ;;           |                                    |
 ;;           V                                    V
 ->     ( ( a b c )   (d  (e)  f)   (g h i)  )
```

With cons, the last argument is a list (almost always in the real world, and always in this class).  cons stretches the opening paren for that second argument to include the first argument:

```
(cons    'a    '( b )      )
 ;;                    |
 ;;           ----------
 ;;           V
 ->       (  a       b )
```
What is the point of cons?

While append and list make more sense, cons is more closely tied to use in recursion:

(define (sent-square-all sent)
  (if (empty? sent)
      ()
      (se (square (first sent))
       (sent-square-all (bf sent))))

(ssa '(1 2 3)) ➔ (se 1 (se 4 (se 9 '())))

(define (list-square-all lst)
  (if (null? lst)
      ()
      (cons (square (car lst))
            (list-square-all (cdr lst))))

(lsa '(1 2 3)) ➔ (cons 1 (cons 4 (cons 9 '())))

Spring 2006 CS3: 18
### Sentences(words) vs lists: selectors

<table>
<thead>
<tr>
<th>Car</th>
<th>Returns the first element of the list</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Returns the first word (although, works on non-words)</td>
</tr>
<tr>
<td>Cdr</td>
<td>Returns a list of everything but the first element of the list</td>
</tr>
<tr>
<td>Butfirst</td>
<td>Returns a sentence of everything but the first word (but, works on lists)</td>
</tr>
<tr>
<td>Last</td>
<td>...</td>
</tr>
<tr>
<td>Butlast</td>
<td>...</td>
</tr>
</tbody>
</table>
### Sentences(words) vs lists: HOF

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>map</strong></td>
<td>Returns a list where a func is applied to every element of the input list.</td>
</tr>
<tr>
<td></td>
<td>Can take multiple input lists.</td>
</tr>
<tr>
<td><strong>every</strong></td>
<td>Returns a sentence where a func is applied to every element of an input</td>
</tr>
<tr>
<td></td>
<td>sentence or word.</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>Returns a list where every element satisfies a predicate.</td>
</tr>
<tr>
<td></td>
<td>Takes a single list as input</td>
</tr>
<tr>
<td><strong>keep</strong></td>
<td>Returns a sentence or word where every element satisfies a predicate</td>
</tr>
<tr>
<td></td>
<td>Takes a single list as input</td>
</tr>
<tr>
<td><strong>reduce</strong></td>
<td>Returns the value of applying a function to successive pairs of the (single)</td>
</tr>
<tr>
<td></td>
<td>input list</td>
</tr>
<tr>
<td><strong>Accumulate</strong></td>
<td>Returns the value of applying a function to successive pairs of the input</td>
</tr>
<tr>
<td></td>
<td>sentence or word.</td>
</tr>
</tbody>
</table>