CSc355: Section on AI Programming in Scheme

Geoff Coulson

Department of Computing, Lancaster University
geoff@comp.lancs.ac.uk
Administrative details

– Geoff Coulson
  • room C19, Infolab, phone 103064, geoff@comp.lancs.ac.uk
  • web: http://www.comp.lancs.ac.uk/computing/staff/geoff/

– This part of the course
  • 3 lectures per week for two weeks
Scheme is based on LISP…

- LISP
  - **LIS**t **P**rocessing language.
    - or, “lots of irrelevant silly parentheses” :-)  
  - long pedigree; contemporary with FORTRAN
    - invented by John McCarthy in 1956
  - based on a pure ‘functional’ approach
    - i.e. programs are sets of functions
      - functions return a value and don’t change their arguments
      - no global variables
    - leads to programs that are robust and easy to debug
  - usually interactive and interpreted
  - good for rapid prototyping as well as AI

- Scheme
  - LISP -> Scheme is roughly analogous to C -> Java
Learning to program in Scheme

- we take a pragmatic rather than an abstract approach
  - a “Scheme 48” interpreter is installed on the PCs in the lab
  - you can get this yourself from http://www.s48.org/index.html
  - there’s also an MIT Scheme interpreter on the central University unix server
    - you need to put the following in your .login file:
      ```
      set path=($path /usr/local/packages/mit-scheme-7.3)
      set path=($path /usr/local/packages/mit-scheme-7.3/bin)
      ```
- lots of programming and code examples involved
  - you will learn as much about a ‘different way of programming’ as about AI programming
  - you should try things out between the lectures!
    - this is the only way to really understand the material
  - exam questions will also be programming/ code oriented
Outline of what’s to come

– core material
  • unit 1: introduction to Scheme: objects and lists
  • unit 2: introduction to Scheme: lists and recursion
  • unit 3: more built-in Scheme procedures
– applying the core material
  • unit 4: search
  • unit 5: more on search
– (5 units in 6 lecture slots allows for some slippage…)

Geoff Coulson, Lancaster University 2/1/07
Books

- “The structure and Interpretation of Computer Programs”, Abelson and Sussman (in library)
- Lisp books (for background)
  - “Lisp”, Patrick Henry Winston and Berthold Klaus Paul Horn, Addison Wesley
    - there are many editions, any one is fine
  - “Metamagical Themas: Questing for the Essence of Mind and Pattern”, Douglas Hofstadter
    - contains an excellent short introduction to Lisp
Unit 1: Introduction to Scheme - objects and lists

• aims
  – to understand basic programming concepts in Scheme
  – to be able to go away and play with the Scheme interpreter
The Scheme interpreter

- start the interpreter (type “scheme”) and you get a prompt:

```
>
```

- Scheme evaluates expressions typed at the prompt in a so-called read, eval, print loop (REPL)

```
> (+ 2 2)
4
>
```

- can also read in files of expressions—i.e. full programs: use (load “filename.scm”)
Objects and lists

– Scheme consists of ‘objects’ and ‘lists’

– objects
  • examples: 4 9.6 fred? f-smith “fred” #\a max
  • i.e. ‘symbols’ (including procedure names) and ‘values’
  • special boolean objects: #t and #f
  • no relation to Java ‘objects’!

– lists
  • examples: (a b c) (((a))) ((26 a (b c)) b c 4.1)
    ()
  • i.e. sequences of objects or lists enclosed in brackets
Evaluating Scheme expressions

- example of an expression
  \[
  > (\text{max} \ (* \ (+ \ 6 \ 3) \ (- \ 6 \ 3)) \ 6 \ (* \ 2 \ 1))
  \]
  \[
  27
  \]
  
- an expression is a list of sub-exprs or objects
  - the first element must be a symbol that represents a \textit{procedure name}
    \(\) (or a sub-expr that evaluates to a procedure name)
  - the remaining elements are \textit{arguments} to the procedure
    \(\) (or sub-exprs that evaluate to arguments)

- evaluation starts with the arguments
  - symbols evaluate to their values
  - values (numbers, strings etc.) evaluate to themselves
  - evaluation is a recursive process

- then the results are passed to the named procedure
  - procedures can be either built-in or user defined
Procedures and special forms

– some expressions *look* like procedure calls but are not; they are built-in *special forms*

– these follow individual special rules for evaluation

– examples: quote, if, cond, define, let, begin and lambda...
  
  • discussion of all of these is coming up…
A built-in procedure: $reverse$

– if the symbol $x$ has previously been given the value of $(a\ b\ c)$, then:

> (reverse x)
(c b a)

• n.b. $reverse$ returns a $new$ list, not a changed version of the given list
• this is generally true; this is why $Scheme$ is a ‘functional’ language
  – but there are a few exceptions...
A built-in special form: *quote*

- *quote* suppresses evaluation
  ```lisp
  > (quote (a b c))
  (a b c)
  ```
- i.e. the second sub-expr in the quote expression is *not evaluated*
- usually a quote in expressed in the (entirely equivalent) ‘macro’ form: \`{(a b c)`
  - e.g.,
    ```lisp
    > (reverse `{(a b c))
    (c b a)
    ```
More built-in procedures: \textit{car}, \textit{cdr}

\textit{car}: returns a copy of the first element of a list

\textit{cdr}: returns a copy of a list with first element removed

\begin{itemize}
\item \textbf{examples}
\begin{itemize}
\item (car ‘(a b c d)) is a
\item (car ‘((a) b (c))) is (a)
\item (car ‘(+ 2 2)) is +
\item (car ‘()) is ()
\end{itemize}
\begin{itemize}
\item (cdr ‘(a b c d)) is (b c d)
\item (cdr ‘((a) b (c))) is (b (c))
\item (cdr ‘(+ 2 2)) is (2 2)
\item (cdr ‘()) is ()
\end{itemize}
\end{itemize}

\textbf{– car/cdr} applied to a non-list results in an error

> (car 6)

<error>
Extended *car* and *cdr*

– in a slightly bizarre way, *car* and *cdr* generalise to *cXXr*, *cXXXr*, *cXXXXr* etc.

…where each X is either

– an ‘a’, signifying *car*, or

– a ‘d’, signifying *cdr*

– evaluation works right to left

\[
\begin{align*}
(cadr \ (a \ b \ c)) &= (car \ (cdr \ (a \ b \ c))) = b \\
(caar \ ((a \ b) \ c)) &= (car \ (car \ ((a \ b) \ c))) = a \\
(caadr \ ((a \ ((b \ c) \ d)) &= b \\
(cadar \ ((a \ (b \ c)) \ d)) &= b
\end{align*}
\]

e tc.
Another built-in procedure: *cons*

- procedure to add a new element to the front of a list
  
  ```
  > (cons 'pie '(cake biscuit))
  (pie cake biscuit)
  > (cons 'waffle
   (cons 'pie
    (cons 'cake
     (cons 'biscuit '()))))
  (waffle pie cake biscuit)
  ```

- note again that a *new* list is returned; the arguments are not changed...
- ‘consing up a list’
More built-ins: *if* (special form), *equal?* (predicate procedure)

- value of following is *bus* if x=land, *ship* otherwise
  > (if (equal? x ‘land) ‘bus ‘ship)
  ship

- note that ‘if’ is a *special form*: only *one* of the two possible result expressions (consequent or alternative) is actually evaluated

- what is the value of the following?
  (if (equal? (car (reverse ‘(a b))) ‘a) (+ 1 1) 77)

- work it out now…
Another special form: \textit{cond}

- value of the following is \textit{bus} if \( x=\text{land} \), or \textit{ship} if \( x=\text{sea} \), or \textit{plane} otherwise
  \[
  \text{(cond (\text{equal? } x \text{ 'land}) \text{ 'bus})}
  \text{((equal? } x \text{ 'sea) \text{ 'ship})}
  \text{(else 'plane))}
  \]
- can take any number of (predicate result) ‘clauses’
- \text{cond} evaluates and returns the result of first clause whose predicate evaluates to \#t (or we hit the ‘else’)
  - or it returns \#f if no clauses evaluate to \#t and there’s no ‘else’ clause
- what is the value of the following?
  \[
  \text{(cond ((equal? (car (reverse '('a b))) 'a) 41)}
  \text{(else 77))}
  \]
The *define* special form

- symbols are given a value (we say: given a “binding”) via the *define* built-in special form
- example

```scheme
> (define bill (if (equal? (car (reverse '(a b))) 'a)
   (+ 1 1)
   77))

> bill
77
```

- note that the first argument (the symbol) is not evaluated
More built-ins: *begin* (special form), *display* (procedure)

- we can define a list of expressions using the *begin* special form
  - expressions are evaluated sequentially
  - result of the ‘begin’ is the result of the *last* expression; the rest are typically evaluated for side effects (e.g. I/O)
- example:
  
  ```lisp
  (begin (display "adding 2 and 2") (+ 2 2))
  ```
Another special form: *lambda*

- used to define procedures
  - notation derived from lambda calculus
- example:
  - define a procedure that takes one argument:
    \[
    > (\text{lambda} \ (x) \ (* \ x \ x))
    \]
  - now apply the procedure:
    \[
    > ((\text{lambda} \ (x) \ (* \ x \ x)) \ 2)
    \]
    4
- as above, procedure ‘body’ can be a list of expressions (value of last is returned)
- as seen above, procedures are *anonymous*
Naming procedures

– we simply bind the value of a procedure to a symbol

  • e.g. (define sqr (lambda (x) (* x x)))

– more examples

  (define rac (lambda (l)
    (car (reverse l))))
  (define rdc (lambda (l)
    (reverse (cdr (reverse l)))))
  (define how-to-go (lambda (x)
    (cond ((= x 1) 'land)
      ((= x 2) 'sea)
      (else 'air))))

  n.b. “=“ is like “equal?” but does numerical testing

– try defining “snoc”, a backward version of cons

  • answer in unit 2
Naming procedures 2

– a shorthand is available that avoids the use of lambda

(define rdc (lambda (l)
    (reverse (cdr (reverse l)))))

==

(define (rdc l)
    (reverse (cdr (reverse l))))
Block structure

– the following are equivalent except that the scope of ‘bill’ is global in the former and local in the latter

> (define bill 1)
> (define (fred x) (+ x bill))

> (define (fred x)
    (define bill 1)
    (+ x bill))

– better to avoid global scope where possible…
Summary

– Scheme is an interpreted language and the interpreter evaluates Scheme expressions
– expressions are of the form
  \((\text{procedure}\text{name}\ \text{arg}_1\ \text{arg}_2\ \ldots\ \text{arg}_n)\)
– procedures follow standard evaluation rules; but special forms don’t
– we saw some built-in procedures and special forms
  • max, +, −, *, reverse, quote, car, cdr, cXXr, cons, if, cond, define, begin, lambda, equal?, display
Unit 2: Introduction to Scheme - lists and recursion

• aims
  – to understand some more basic Scheme programming concepts - particularly recursion
  – to be able to go away and play more meaningfully with the Scheme interpreter
Definition of “snoc”

> (define snoc (lambda (a l)
  (reverse (cons a (reverse l))))))

> (snoc 'z `(w x y))
(w x y z)
Repetition

- Jacopini identified *six* programming language facilities required to perform general computation
  - basic operations
    - +, -, *, /, equal?, car, cdr, cons, ...
  - input/output
    - read, write, display, ...
  - assignment or binding
    - define, lambda, let…
  - sequencing
    - begin, let and define bodies, ...
  - selection
    - if, cond, …
  - *repetition*: ??
  - repetition in Scheme is usually (not always) achieved using *recursion*
Example: a “power2” procedure

– the problem...

• define a procedure power2 that takes an integer argument $x$ and returns $2^x$

• so...
  – (power2 0) should return $2^0 = 1$
  – (power2 1) should return $2^1 = 2$ (n.b. 2 times (power2 0))
  – (power2 2) should return $2^2 = 4$ (n.b. 2 times (power2 1))
  – (power2 3) should return $2^3 = 8$ (n.b. 2 times (power2 2))
  – etc.
Power2: the Scheme solution

– step 1: envisage a (potentially infinite) “tower” of calls with a non-recursive one at the bottom

(power2 3) = (* 2 (power2 2))
(power2 2) = (* 2 (power2 1))
(power2 1) = (* 2 (power2 0))
(power2 0) = 1

– step 2: express the procedure in terms of

• i) the simple, non-recursive, case(s); and
• ii) the recursive case(s)

(define (power2 x)
  (if (= x 0) 1
      (* 2 (power2 (- x 1)))))

* simple, non-recursive, case
* recursive case
Let’s have that in English!

```
(define (power2 x)
  (if (= x 0)
      1
      (* 2 (power2 (- x 1)))))
```

\[=\]

\[\text{power2}(x)\]

\[\begin{align*}
\text{non-recursive case} \\
\text{if } x \text{ is } 0 \text{ then return } 1 \\
\text{else} \\
\text{return } 2 \times (\text{power2 with argument } x-1) \\
\end{align*}\]

\[\text{recursive case}\]
Isn’t this circular?

– we can use Scheme’s *trace* procedure…

  > (trace power2)
  > (power2 2)
  1. Trace: (POWER2 '2)
  2. Trace: (POWER2 '1)
      3. Trace: (POWER2 '0)
      3. Trace: POWER2 ==> 1
  2. Trace: POWER2 ==> 2
  1. Trace: POWER2 ==> 4

  4

– so, power2 is evidently not a circular definition
  • actually its a *linear* definition
  • its linear because it is a tower with a ‘solid foundation’
    (i.e. a simple, non-recursive, case)
Recursive foundations

– a firm foundation
  • man holds photo of himself 10 years younger in which he is holding a photo of himself 10 years younger in which ... of himself as a new born baby; **END**

– some ‘shaky’ foundations
  • box of chocs on which a girl holds a box of chocs on which a girl holds a box of chocs...
  • look in a doubled mirror and see yourself in the mirror...
  • “a horse is a four legged animal that is produced by two other horses…”
  • the numbskulls (little men inside little men…)
  • dictionary defn of recursion: *recursion: see ‘recursion’*...
Example: a pile of stones

- define a procedure that returns a list of \( n \) instances of the symbol `stone` (e.g. `(stone stone stone stone)` with \( n=3 \))

- step 1: envisage the tower
  
  `(pile 3) = (cons ‘stone (pile 2))`
  `(pile 2) = (cons ‘stone (pile 1))`
  `(pile 1) = (cons ‘stone (pile 0))`
  `(pile 0) = ‘()`

- step 2: express as i) simple and ii) recursive cases
  
  `(define (pile n)`
  `(if (= n 0)`
  ` ‘()`  
  ` (cons ‘stone (pile (- n 1))))))`
Example: list length

– return the length of a list

– the design
  • simple case: length of empty list is 0
  • recursive case: length of a list is length of (cdr the-list) + 1

– the solution

  (define (length l)
    (cond ((null? l) 0)
          (else (+ 1 (length (cdr l))))))

  • (n.b. null? is a built-in boolean procedure that returns #t iff its argument is an empty list)
Example: list membership

- the problem: is a symbol $s$ present in a list $l$?
- the design:
  - simple case: $s$ is not present if $l$ is empty; or $s$ is present if it is equal to the car of $l$
  - recursive case: $s$ is present if it is a member of the cdr of $l$
- the solution

\[
\begin{align*}
&\text{(define (member? s l)} \\
&\quad \text{(cond ((null? l) #f) )}
\end{align*}
\]
\[
\begin{align*}
&\quad ((\text{equal? s (car l)}) \ #t) \\
&\quad (\text{else (member? s (cdr l))))})
\end{align*}
\]

- (n.b. there is a built-in ‘member’ that is slightly different)
Example: summing a list

- the problem: return the sum of a list of ints
- the design:
  - simple case: the sum of an empty list is 0
  - recursive case: the answer is the first number in the list (car) added to the sum of the cdr
- the solution???
  - over to you…
Example: *symbol-count*

- count all the symbols in a potentially nested list
  - note difference from *length*
  - `>` `( ((a b c) a (a b c)) a ((a b c d) ((a)) a))`
    14

- the design
  - simple case: the argument is an empty list (answer: 0); *or* the argument is not a list at all but a single symbol (answer: 1)
    - (n.b. *symbol?* returns #t iff its argument is an symbol)
  - recursive case: the symbol-count of the car of the list + the symbol-count of the cdr of the list
Symbol-count (cont.)

- the solution

  (define (symbol-count l)
    (cond ((symbol? l) 1)
          ((null? l) 0)
          (else (+ (symbol-count (car l))
                   (symbol-count (cdr l))))))

- n.b.
  
  • (the first recursive call on a one-symbol list leads to the ‘symbol?’ case)
  • the second recursive call on a one-symbol list leads to the ‘null?’ case)
Append: another example with multiple simple cases

- the problem: concatenate two lists

  \( \text{(append '(a b c) '(d e)) returns (a b c d e)} \)
  \( \text{(append '(a b c) '()) returns (a b c)} \)
  \( \text{(append '(a (b c)) '(d e)) returns (a (b c) d e)} \)

- the solution

  \( \text{(define (append l1 l2)} \)
  \( \text{(cond ((null? l1) l2)} \)
  \( \text{((symbol? l1) (cons l1 l2))} \)
  \( \text{(else (append \text{rdc l1)}} \)
  \( \text{(cons (rac l1) l2)))}) \)
Append example

(append '(a b c) '(d e)) →
(append '(a b) '(c d e)) →
(append '(a) '(b c d e)) →
(append '() '(a b c d e)) →
(a b c d e)
Aside: an alternative formulation of `append` using anonymous functions (\textit{lambda})

\begin{verbatim}
(define (append l1 l2)
  (cond ((null? l1) l2)
        ((symbol? l1) (cons l1 l2))
        (else
         (append
          ((lambda (l)(reverse (cdr (reverse l)))) l1))
          (cons ((lambda (l)(car (reverse l)))) l1) l2))))
\end{verbatim}
Append and list

- `append` is also available as a built-in procedure
  - takes an arbitrary number of arguments rather than just two
- the built-in procedure `list` is similar to `append` except it makes a list out of its arguments—it doesn’t ‘run its args together’

```
> (append '(a b) '(c d))
(a b c d)
> (append '(((a) (b)) '((c) (d)))
(((a) (b)) (c) (d))

> (list '(a b) '(c d))
(((a b) (c d))
> (list '(((a) (b)) '((c) (d)))
(((a) (b)) ((c) (d)))
```
Another one for you to try at home…

– define a procedure `flatten` that returns a ‘flat’ list that contains in order all the symbols in its single list argument

– for example

> (flatten `(((a b c) d (e f g)) h ((i j k l) ((n)) a)))
(a b c d e f g h i j k l m n a)

– hints

• there are two simple cases
• you probably need to use both `list` and `append`
• not a million miles away from symbol-count! (i.e. work separately on the `car` and `cdr` of the argument)
Summary

– we’ve seen how Scheme handles repetition - through recursion

– we’ve studied how to break problems down into recursive solutions
  • simple case(s) and recursive case(s)
  • converging to simple case by (e.g.)
    – decrementing a counter (e.g. pile of stones)
    – ‘cdr’ing’ down lists (e.g. list length, member)
    – ‘car and cdr based recursion’ (e.g. flatten, append)

– along the way we’ve met some more built-ins
  • symbol?, null?, append, list, length, member, trace
Unit 3: More Scheme procedures

• aims
  – to introduce more examples of Scheme code
  – to introduce some new Scheme facilities that will be used in the application-oriented units
    • properties
    • second-order procedures (sort, map and apply)
    • remove-if
    • lambda (again)
First, the homework answer

– problem: define a procedure `flatten` that returns a flat list of all the symbols in its argument

```
> (flatten `(((a b c) d (e f g)) h ((i j k l) ((n)) a)))
(a b c d e f g h i j k l m n a)
```

– solution:

```
(define (flatten s)
  (cond  ((null? s) '(())
    ((symbol? s) (list s))
    (else (append (flatten (car s))
                  (flatten (cdr s))))))
```
Properties

- Scheme symbols may have associated *properties*
  - each symbol can have any number of properties
  - a property is a \((name\ val)\) pair; the choice of both names and values is entirely up to the programmer

- properties are implemented as global ‘2-d table’;
  each symbol’s properties are kept in an list like this:
  \(((name_1\ val_1)(name_2\ val_2)...(name_i\ val_i))\)

- 2d-put! and 2d-get are used to set and get properties
  > (2d-put! ‘pyramid-a ‘colour ‘red)
  > (2d-put! ‘pyramid-a ‘is-a ‘pyramid)
  > (2d-get ‘pyramid-a ‘colour)
  red
Map and apply

- map and apply are second-order procedures
- map ‘interates’ a procedure over a list of arguments...
  > (map odd? '(1 2 3))
  (#t #f #t)
  - (n.b. odd? returns #t if its argument is an odd number)
  (map = '(1 2 3) '(3 2 1)); 2 lists for a 2 arg proc
  (#f #t #f)
- note that (map + '(1 2 3)) produces an error as + needs at least 2 arguments
  - to apply a function to a list of arguments we can use apply
    > (apply + '(1 2 3))
    6
Map and apply (cont.)

– we can use map and apply to define a faster symbol counting procedure (i.e. one with less recursion)

\[
\text{(define (symbol-count1 s); original version)}
\]
\[
\text{(cond ((null? s) 0)}
\]
\[
\text{((symbol? s) 1)}
\]
\[
\text{(else (+ (symbol-count1 (car s))}
\]
\[
\text{(symbol-count1 (cdr s)))))}
\]

\[
\text{(define (symbol-count2 s); new version)}
\]
\[
\text{(cond ((null? s) 0)}
\]
\[
\text{((symbol? s) 1)}
\]
\[
\text{(else (apply + (map symbol-count2 s)))))}
\]

– let’s check it out with (symbol-count2 ‘((a b) c)) ...
Old symbol-count

> (symbol-count1 '((a b) c))
1. Trace: (SYMBOL-COUNT '((A B) C))
2. Trace: (SYMBOL-COUNT '(A B))
   3. Trace: (SYMBOL-COUNT 'A)
   3. Trace: SYMBOL-COUNT ==> 1
   4. Trace: (SYMBOL-COUNT '(B))
   4. Trace: SYMBOL-COUNT ==> 1
   4. Trace: (SYMBOL-COUNT ())
   4. Trace: SYMBOL-COUNT ==> 0
3. Trace: SYMBOL-COUNT ==> 1
2. Trace: SYMBOL-COUNT ==> 2
2. Trace: (SYMBOL-COUNT '(C))
3. Trace: (SYMBOL-COUNT 'C)
3. Trace: SYMBOL-COUNT ==> 1
3. Trace: (SYMBOL-COUNT ())
3. Trace: SYMBOL-COUNT ==> 0
2. Trace: SYMBOL-COUNT ==> 1
1. Trace: SYMBOL-COUNT1 ==> 3
New version with map/apply

> (symbol-count2 '((a b) c))
  1. Trace: (SYMBOL-COUNT2 '((A B) C))
  2. Trace: (SYMBOL-COUNT2 '(A B))
     3. Trace: (SYMBOL-COUNT2 'A)
     3. Trace: SYMBOL-COUNT2 ==> 1
     3. Trace: (SYMBOL-COUNT2 'B)
     3. Trace: SYMBOL-COUNT2 ==> 1
     2. Trace: SYMBOL-COUNT2 ==> 2
  2. Trace: (SYMBOL-COUNT2 'C)
     2. Trace: SYMBOL-COUNT2 ==> 1
  1. Trace: SYMBOL-COUNT2 ==> 3
3
  • note that there is significantly less recursion => faster
Sort

- another simple second-order procedure
  \( \text{sort } \text{<sequence>} \text{ <2-place-predicate>} \)

- we can use any 2-place predicate that defines an ordering (i.e. if \( x \ pred \ y \) return \#t; if \( x \ !pred \ y \) return \#f)

- examples
  
  > (sort `(8 6 3 5 9 2) <)
  (2 3 5 6 8 9)

  > (sort `(8 6 3 5 9 2) >)
  (9 8 6 5 3 2)
Remove-if

– a useful second-order procedure that takes a procedure (predicate) and a list, and returns a ‘filtered’ list that is identical except that the elements for which the predicate is true are omitted

> (define (remove-if pred l)
  (cond ((null? l) l)
    ((pred (car l)) (remove-if pred (cdr l)))
    (else (cons (car l)
                (remove-if pred (cdr l))))))

> (remove-if fruit? (broccoli milk apple bread butter pear))
(broccoli milk bread butter)
Anonymous procedures revisited

– suppose we want to know which items in a list of groceries are fruit

  • we already know how to do the following:

    > (define (fruit? x)
        (equal (2d-get x 'kind-of) 'fruit))

    > (map fruit?
          '(broccoli milk apple bread butter pear))
    (#f #f #t #f #f #t)
Anonymous procedures (cont.)

...but it’s a pain to have to think up a name and explicitly define *fruit*? when it may only be used once

- *lambda* enables us to conveniently define an ‘anonymous’ procedure on the fly:
  
  ```lisp
  > (map (lambda (x) (equal (2d-get x 'kind-of) 'fruit))
        '(broccoli milk apple bread butter pear))
  (#f #f #t #f #f #t #f)
  ```

- a lambda expression can go anywhere a procedure name can go (think: lambda == ‘define anonymous’)

- *lambda* is especially useful with second order procedures
Summary

– we have covered the following:
  • 2d-get, 2d-put!, map, apply, sort, odd?
– (note a common convention for naming procedures: a ‘?’ suffix denotes a predicate; a ‘!’ suffix denotes a procedure with side effects)
– we now know what a second-order procedure is
  • it’s a procedure that takes a procedure as an argument (or returns a procedure as a result)
Unit 4: Search

• aims
  – to apply our working knowledge of Scheme
  – to understand the ‘search problem’ (ubiquitous in AI)
  – to develop a generic search program that can be specialised for a range of search strategies
  – and then to specialise our generic search program to do depth first search
    • we’ll specialise it further in unit 5
The search problem

• given a problem domain with lots of decision points or potential solutions…
• we *search* through the decision tree or the space of solutions
• examples
  – given a roadmap, find a road route from Lancaster to London
  – find your way out of a maze
  – find the best move in a game such as chess, draughts or noughts-and-crosses
  – find the correct way to parse a sentence
    • “time flies like an arrow”
  – make a goal-directed plan from a set of available sub-plans
    • e.g. make a plan for “*put block A on block B*” from:
      *lift block, move block, clear block, put-down block*
Representing search

- a search problem is characterised as a search space (graph) of states (nodes)
  - begin at a start state from which there are links to a number of possible successor or child states (from which there are in turn further successors, etc., etc.)
  - walk the links until we encounter the goal: a finish state
- a search space is naturally represented as a graph or a tree
- many search strategies are available
  - e.g. depth first search or breadth first search
Search strategy 1: depth first search

“Examine one child of the start node; if this is not the finish node, examine one of its successors (children), again ignoring the rest.

Do the above repeatedly. If there are no links left to explore at a given level, back-up to the last place there was a choice and continue from there.”
Search strategy 2: breadth first search

- movement is ‘level by level’: “at each state examine all the children of that state; if we fail to find a finish state among these, move down to one of the children and proceed from there”
- i.e. the links one level down are fully examined before we go down to the next level
Representing a search space

- a tree can easily be represented as a nested list

\[
(s \ (j \ (k \ (r \ (w)))) \ x) \ (t \ (u \ (y)) \ (v \ (z))))
\]

= 

\[
(s \ (j \ (k \ (r \ (w)))) \ x) \\
\ (t \ (u \ (y)) \\
\ (v \ (z))))
\]

Diagram:

- **S**
  - **j**
    - **k**
      - **r**
        - **w**
  - **x**
  - **t**
    - **u**
      - **y**
    - **v**
      - **z**
Depth first search in a nested list (tree)

- basic *dfs* is in a nested list is trivial!

```
(define (search finish-state tree)
  (cond ((equal? finish-state tree) #t)
       ((symbol? tree) #f)
       (else (or (search finish-state (car tree))
                  (search finish-state (cdr tree))))))
```

- **but** this approach has severe limitations
  - it only works on trees, not graphs
  - it assumes the tree is already available (i.e., pre-computed)
  - it only returns #t or #f; it doesn’t remember the route it took to get to the finish
  - we can’t easily extend it to breadth first search and other strategies

- so, we will take a more general approach...
A better search space representation

– use a ‘children’ property to represent a graph

(2d-put! 's 'children '(l o))
(2d-put! 'l 'children '(m f))
(2d-put! 'm 'children '(n))
(2d-put! 'n 'children '(f))
(2d-put! 'o 'children '(p q))
(2d-put! 'p 'children '(f))
(2d-put! 'q 'children '(f))

– all trees are also graphs, so we get trees for free
A general search program (skeleton)

- \textit{search} just calls \textit{search1}, having put the given start state in a list...

\begin{verbatim}
(define (search s f)
  (define (search1 queue finish)
    (cond ((null? queue) #f) ; not found
          ((equal? (car queue) finish) #t); found
          (else (search1
                  \(<\textit{appropriate merge of}\>
                  (expand (car queue)) and queue>
                  finish))))
    (search1 (list s) f)); initialise

- ‘queue’ is an ordered list of nodes for us to examine next
- \textit{expand} returns the children of given node
\end{verbatim}
Specialisation to depth first search

– fill in the blanks as follows

- ‘expand’ becomes
  
  (define (expand node) (2d-get node ‘children))

- ‘appropriate merge’ becomes
  
  (append (expand (car queue)) (cdr queue))

(define (dfs s f)
  (define (dfs1 queue finish)
    (cond ((null? queue) #f)
          ((equal? (car queue) finish) #t)
          (else (dfs1 (append (expand (car queue))
                           (cdr queue))
                   finish)))))

  (dfs1 (list s) f))
A trace for a ‘town to town’ search

> (trace dfs dfs1)
(dfs 's 'f)
1. Trace: (DFS 'S 'F)
   2. Trace: (DFS1 '(S) 'F)
      3. Trace: (DFS1 '(L O) 'F)
         4. Trace: (DFS1 '(M F O) 'F)
            5. Trace: (DFS1 '(N F O) 'F)
               6. Trace: (DFS1 '(F F O) 'F)
               6. Trace: DFS1 ==> #t
               5. Trace: DFS1 ==> #t
                  4. Trace: DFS1 ==> #t
                     3. Trace: DFS1 ==> #t
                        2. Trace: DFS1 ==> #t
                           1. Trace: DFS ==> #t
                              #t
Another trace

> (dfs 's 'f)
1. Trace: (DFS 'S 'F)
   2. Trace: (DFS1 '(S) 'F)
      3. Trace: (DFS1 '(L O) 'F)
         4. Trace: (DFS1 '(M R O) 'F)
            5. Trace: (DFS1 '(N R O) 'F)
               6. Trace: (DFS1 '(R R O) 'F)
                  7. Trace: (DFS1 '(R O) 'F)
                     8. Trace: (DFS1 '(O) 'F)
                        9. Trace: (DFS1 '(P Q) 'F)
                           10. Trace: (DFS1 '(T Q) 'F)
                              11. Trace: (DFS1 '(Q) 'F)
                                 12. Trace: (DFS1 '(T) 'F)
                                    13. Trace: (DFS1 '()' 'F)

   13. Trace: DFS1 ==> #f

(etc.)
Making *dfs* return a route

- pack the required information into the queue elements
- the queue *formerly* developed like this (in our first example graph):

\[(s) \rightarrow (l \ o) \rightarrow (m \ f \ o) \rightarrow (n \ f \ o) \rightarrow (f \ f \ o)\]

- we will change it so it develops like this:

\[
\begin{align*}
((s)) \\
((l \ s) \ (o \ s)) \\
((m \ l \ s) \ (f \ l \ s) \ (o \ s)) \\
((n \ m \ l \ s) \ (f \ l \ s) \ (o \ s)) \\
((f \ n \ m \ l \ s) \ (f \ l \ s) \ (o \ s))
\end{align*}
\]

success achieved when (equal? (caar queue) finish)

*this is the search route*
Making *dfs* return a route (cont.)

- new code in bold font

```
(define (dfs s f)
  (define (dfs1 queue finish)
    (cond ((null? queue) #f)
          ((equal? finish (caar queue))
           (reverse (car queue)))
          (else (dfs1 (append (expand (car queue)) (cdr queue))
                     finish))))
  (dfs1 (list (list s)) f))
```

```
(define (expand route) ; return list of new routes
  (map (lambda (child) (cons child route))
       (2d-get (car route) 'children)))
```
Detecting closed loops in *dfs*

– so far, we have assumed *loop-free* graphs
  
  • our program will fail to terminate if there are loops!

– we can fix this by augmenting ‘expand’, using *remove-if* to flush already-visited nodes from the route

```scheme
(define (expand route)
  (remove-if (lambda (pth) (member (car pth) (cdr pth)))
    (map (lambda (child) (cons child route))
      (2d-get (car route) ‘children))))
```

*our remove-if predicate*
A *dfs* example with closed loops

– here’s the data structure

\[
\begin{align*}
(2d-put! \ 's \ 'children \ '(a \ b)) \\
(2d-put! \ 'a \ 'children \ '(s \ b \ f)) \\
(2d-put! \ 'b \ 'children \ '(s \ a \ c)) \\
(2d-put! \ 'c \ 'children \ '(b \ f)) \\
(2d-put! \ 'f \ 'children \ '(a \ c))
\end{align*}
\]

– given the queue develops as follows:

\[
\begin{align*}
(((s)) & \quad \text{call: (dfs ‘s ‘f)} \\
((a \ s) \ (b \ s)) \\
((s \ a \ s) \ (b \ a \ s) \ (f \ a \ s) \ (b \ s)) \\
((s \ b \ a \ s) \ (a \ b \ a \ s) \ (c \ b \ a \ s) \ (f \ a \ s) \ (b \ s)) \\
((b \ c \ b \ a \ s) \ (f \ c \ b \ a \ s) \ (f \ a \ s) \ (b \ s))
\end{align*}
\]
Summary

- search is a problem with states and a state space
- basic tree search is trivial but restrictive; a general search procedure is more useful
  - can be easily adapted to do *dfs*, *bfs*, ...
  - uses an explicit graph state space representation that is very flexible and extensible (graphs + trees)
  - returns the route traversed
  - detects and avoids loops
Unit 5: More searching

• aims
  – to continue the exploration of search started in the previous unit
    • breadth first search
    • best first search
    • hill climbing
    • branch and bound
    • beam search
Breadth first search

- modifying our \textit{dfs} program to do \textit{bfs} is trivial!

  just add new routes to the \textit{back} of queue rather than the front

\begin{verbatim}
(define (bfs s f)
  (define (bfs1 queue finish)
    (cond ((null? queue) #f)
      ((equal? finish (caar queue))
        (reverse (car queue)))
      (else (bfs1 (append (cdr queue)
                   (expand (car queue)))
                     finish))))
  (bfs1 (list (list s)) f))
\end{verbatim}
Evolution of the queue of routes in bfs

- ((s)) ->
- ((l s)(o s)) ->
- ((o s)(m l s)(f l s)) ->
- ((m l s)(f l s)(p o s)(q o s)) ->
- ((f l s)(p o s)(q o s)(n m l s))

... and on the next expansion we detect success (equal? (caar queue) ‘f)
Best first search

- *dfs* expands the *first* route in the queue
- *bfs* expands the *last* route in the queue...
- *best first search* expands the *‘best’* route
  - ‘best’ is that which is estimated to be nearest to a finish node
  - determining ‘best’ requires a *heuristic* (informed guess)
  - the heuristic sorts the queue of routes according to a predicate *closer*? 
Best first search

(define (best s f)
  (define (best1 queue finish)
    (cond ((null? queue) #f)
          ((equal? finish (caar queue))
           (reverse (car queue)))
          (else (best1 (sort (append
                          (expand (car queue))
                          (cdr queue))
                          (lambda (x y)
                           (closer? x y finish)))
                           finish)))))
  (best1 (list (list s)) f))
An implementation of \textit{closer}?

- the following works if we consider ‘as-the-crow-flies’ geographical distance to be a good heuristic
  - we assume properties called X and Y to represent coordinates
  - we assume the existence of a \textit{sqrt} square-root procedure

\begin{verbatim}
(define (closer? a b target)
  (define (distance n1 n2)
    (sqrt (+ (square (- (2d-get n1 'X)
                        (2d-get n2 'X)))
            (square (- (2d-get n1 'Y)
                        (2d-get n2 'Y)))))
    (< (distance (car a) target)
        (distance (car b) target)))
\end{verbatim}
Hill climbing

- like best first search, but, rather than sorting the whole queue, we sort only the children of the first queue item and place them at the head of the queue
- hill climbing is thus a compromise between best first search and dfs
  - it minimises best first search’s overhead of queue sorting on each expansion
  - while still giving some direction to the blindness of DFS
- homework: define a hill climbing variant of our search program
Branch and bound search

– *guaranteed* to find the ‘shortest’ route from start to finish
– works by sorting the queue of routes in terms of distance travelled so far - always expand the shortest route next
– just use best first search with *shorter*? as sort predicate:

```
(define (shorter? route1 route2)
  (define (route-length p)
    (cond ((null? (cdr p)) 0)
          (else (+ (distance (car p) (cadr p))
                    (route-length (cdr p))))))
  (< (route-length route1) (route-length route2)))
```
Beam search

– think of someone searching in the dark with a torch with a fixed beam width...

– only keep a fixed number, $w$, of routes in the queue
  - if there are more than $w$ routes in the queue discard all but the first $w$
  - then expand all the remaining routes in the queue and sort according to closer?
  - beam search is not guaranteed to find a finish node!
Beam search (cont.)

(define (beam s f w)
  (define (first-w s w)
    (cond ((zero? w) `())
      (else (cons (car s)
          (first-w (cdr s) (- w 1)))))
  (define (beam1 queue finish w)
    (cond ((null? queue) #f)
      ((equal? (caar queue) finish)
        (reverse (car queue)))
      (else (beam1 (sort (apply append
          (map 'expand
            (first-w queue w)))
        (lambda (x y)
          (closer? x y finish)))
        finish
        w)))
  (beam1 (list (list s)) f w))
Summary

– we have applied our generalised search program to implement the following:
  • breadth first search
  • best first search - heuristic search
  • hill climbing - gives heuristic direction to DFS
  • branch and bound - guaranteed to find ‘shortest’ route
  • beam search - not guaranteed to find an existing finish

– we have looked at a simple heuristic - distance - which is used by closer? and shorter?
Overall conclusion on the Scheme material

• we have learned some Scheme concepts and vocabulary
• we have learned to program using recursion
  – simple and recursive cases
  – converging to simple case by
    • decrementing integer (e.g. pile of stones)
    • cdr’ing down lists (e.g. list length)
    • car and cdr based recursion (e.g. flatten)
• we have looked at search
  – designed a generic search space representation
  – designed a generic search procedure
  – specialised the generic search procedure with various strategies...