Agenda

1. Session Overview
2. Functional Programming
3. Conclusion
What is the course about?

- Course description and syllabus:
  - [http://www.nyu.edu/classes/jcf/CSCI-GA.2110-001](http://www.nyu.edu/classes/jcf/CSCI-GA.2110-001)

- Textbook:
  - *Programming Language Pragmatics (3rd Edition)*
    - Michael L. Scott
    - Morgan Kaufmann

- Additional References:
  - Osinski, Lecture notes, Summer 2010
  - Grimm, Lecture notes, Spring 2010
  - Barrett, Lecture notes, Fall 2008
  - Gottlieb, Lecture notes, Fall 2009
Session Agenda

- Session Overview
- Functional Programming
- Conclusion
Icons / Metaphors

- Information
- Common Realization
- Knowledge/Competency Pattern
- Governance
- Alignment
- Solution Approach
Session 4 Review

- Subprogram
- Environment of the Computation
- Review of Stack Layout
- Calling Sequences
- Calling Sequences (C on MIPS)
- Parameter Passing
- Generic Subroutines and Modules
- First Class Functions
- Higher-Order Functions
- Block Structure
- Exception Handling
- Coroutines
- Conclusions
Agenda

1. Session Overview
2. Functional Programming
3. Conclusion
Functional Programming

- Historical Origins
- Lambda Calculus
- Functional Programming Concepts
- A Review/Overview of Scheme
- Evaluation Order Revisited
- High-Order Functions
- Functional Programming in Perspective
Functional Programming refers to a programming style in which every procedure is functional, i.e. it computes a function of its inputs with no side effects.

Functional programming languages are based on this idea, but they also provide a number of interesting features that are often missing in imperative languages.

One of the most important and interesting features of functional programming languages is that functions are first-class values. This means that programs can create new functions at run-time.

This can be leveraged to build powerful higher-order functions: a higher-order function either takes a function as an argument or returns a function as a result (or both).

Functional languages draw heavily on the lambda-calculus for inspiration.
The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  » different formalizations of the notion of an algorithm, or effective procedure, based on automata, symbolic manipulation, recursive function definitions, and combinatorics

These results led Church to conjecture that any intuitively appealing model of computing would be equally powerful as well
  » this conjecture is known as Church’s thesis
Turing’s model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage “tape”

» the Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables
Church’s model of computing is called the **lambda calculus**

- based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter \( \lambda \)—hence the notation’s name.
- Lambda calculus was the inspiration for functional programming
  - basis for functional languages (e.g., Lisp, Scheme, ML, Haskell)
  - typed and untyped variants
  - has *syntax* and *reduction rules*
- one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions
Mathematicians established a distinction between

- *constructive* proof (one that shows how to obtain a mathematical object with some desired property)
- *nonconstructive* proof (one that merely shows that such an object must exist, e.g., by contradiction)

Logic programming is tied to the notion of constructive proofs, but at a more abstract level:

- the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs
We will discuss the pure, untyped variant of the λ-calculus.

The syntax is simple:

\[ M ::= \lambda x . M \quad \text{function} \]
\[ \quad | \quad M M \quad \text{function application} \]
\[ \quad | \quad x \quad \text{variable} \]

Shorthands:
- We can use parentheses to indicate grouping
- We can omit parentheses when intent is clear
- \( \lambda x y z . M \) is a shorthand for \( \lambda x . (\lambda y . (\lambda z . M)) \)
- \( M_1 M_2 M_3 \) is a shorthand for \( (M_1 M_2) M_3 \)
In a term $\lambda x . M$, the scope of $x$ is $M$.
We say that $x$ is bound in $M$.
Variables that are not bound are free.

Example:

$$(\lambda x . (\lambda y . (x (z y)))) y$$

- The $z$ is free.
- The last $y$ is free.
- The $x$ and remaining $y$ are bound.

We can perform $\alpha$-conversion at will:

$$\lambda x . (\ldots x \ldots) \rightarrow^\alpha \lambda y . (\ldots y \ldots)$$
The main reduction rule in the $\lambda$-calculus is function application:

$$(\lambda x \cdot M) N \rightarrow_\beta [x \mapsto N]M$$

The notation $[x \mapsto N]M$ means:

$M$, with all free occurrences of $x$ replaced by $N$.

Restriction: $N$ should not have any free variables which are bound in $M$.

Example:

$$(\lambda x \cdot (\lambda y \cdot (x y))) (\lambda y \cdot y) \rightarrow_\beta \lambda y \cdot (\lambda y \cdot y) y$$

An expression that cannot be $\beta$-reduced any further is a normal form.
We have the $\beta$-rule, but if we have a complex expression, where should we apply it first?

$$(\lambda x.\lambda y. y x x)((\lambda x.x)(\lambda y.z))$$

Two popular strategies:

- **normal-order**: Reduce the outermost "redex" first.

  $$[x \mapsto (\lambda x.x)(\lambda y.z)](\lambda y.y x x) \xrightarrow{\beta} \lambda y.y ((\lambda x.x)(\lambda y.z))((\lambda x.x)(\lambda y.z))$$

- **applicative-order**: Arguments to a function evaluated first, from left to right.

  $$(\lambda x.\lambda y. y x x)([x \mapsto (\lambda y.z)]x) \xrightarrow{\beta} (\lambda x.\lambda y. y x x)((\lambda y.z))$$
Church-Rosser Theorem

- If a term $M$ can be reduced (in 0 or more steps) to terms $N$ and $P$, then there exists a term $Q$ such that both $N$ and $P$ can be reduced to $Q$. 

![Diagram showing the Church-Rosser Theorem](image-url)
The property expressed in the Church-Rosser theorem is called confluence. We say that \(-\)reduction is confluent.

- **Corollary**
  - Every term has at most one normal form.

- **Why at most one?**
  - There are some terms with no normal form!

- **Example:**

\[(\lambda x. xx)(\lambda x. xx)\]
The important notion of computability relies on a formal model of computation.

Many formal models have been proposed:

1. General recursive functions defined by means of an equation calculus (Gödel-Herbrand-Kleene)
2. \(\lambda\)-recursive functions and partial recursive functions (Gödel-Kleene)
3. Functions computable by finite machines known as Turing machines (Turing)
4. Functions defined from canonical deduction systems (Post)
5. Functions given by certain algorithms over a finite alphabet (Markov)
6. Universal Register Machine-computable functions (Shepherdson-Sturgis)
7. Any function you can program in your favorite programming language

Fundamental Result

All of these (and many other) models of computation are equivalent. That is, they give rise to the same class of computable functions.

Any such model of computation is said to be Turing complete.
Fact: The untyped -calculus is Turing complete. (Turing, 1937)

- But how can this be?
  - There are no built-in types other than “functions” (e.g., no booleans, integers, etc.)
  - There are no loops
  - There are no imperative features
  - There are no recursive definitions
- **number**: an abstract idea
- **numeral**: the representation of a number
  » Example: 15, fifteen, XV, 0F
    • These are different numerals that all represent the same *number*.

- **Alien numerals**:
  » frobnitz − frobnitz = wedgleb
  » wedgleb + taksar = ?
How can a value of “true” or “false” be represented in the -calculus?

Any way we like, as long as we define all the boolean operations correctly.

One reasonable definition:

- true takes two values and returns the first
- false takes two values and returns the second

| TRUE | \( \lambda a. \lambda b. a \) |
| FALSE | \( \lambda a. \lambda b. b \) |
| IF | \( \lambda c. \lambda t. \lambda e. (c \ t \ e) \) |
| AND | \( \lambda m. \lambda n. \lambda a. \lambda b. m \ (n \ a \ b) \ b \) |
| OR | \( \lambda m. \lambda n. \lambda a. \lambda b. m \ a \ (n \ a \ b) \) |
| NOT | \( \lambda m. \lambda a. \lambda b. m \ b \ a \) |
We can represent the number $n$ in the $\lambda$-calculus by a function which maps $f$ to $f$ composed with itself $n$ times: $f \circ f \circ \ldots \circ f$.

Some numerals:

\[
\begin{align*}
0 & \equiv \lambda f x . x \\
1 & \equiv \lambda f x . f x \\
2 & \equiv \lambda f x . f(f x) \\
3 & \equiv \lambda f x . f(f(f x))
\end{align*}
\]

Some operations:

\[
\begin{align*}
\text{ISZERO} & \equiv \lambda n . n (\lambda x . \text{FALSE}) \text{TRUE} \\
\text{SUCC} & \equiv \lambda n f x . f(n f x) \\
\text{PLUS} & \equiv \lambda m n f x . m f(n f x) \\
\text{MULT} & \equiv \lambda m n f . m(n f) \\
\text{EXP} & \equiv \lambda m n . n m \\
\text{PRED} & \equiv \lambda n . n (\lambda g k . (g \, ^{1^\prime}) (\lambda u . \text{PLUS} (g k) \, ^{1^\prime}) k) (\lambda v . ^{0^\prime}) \, ^{0^\prime}
\end{align*}
\]
How can we express recursion in the λ-calculus?

**Example:** the factorial function

\[
\text{fact}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n - 1)
\]

In the λ-calculus, we can start to express this as:

\[
\text{fact} = \lambda n . (\text{ISZERO } n) \cdot 1^n (\text{MULT } n (\text{fact } (\text{PRED } n)))
\]

But we need a way to give the factorial function a name.

**Idea:** Pass in \text{fact} as an extra parameter somehow:

\[
\lambda \text{fact} . \lambda n . (\text{ISZERO } n) \cdot 1^n (\text{MULT } n (\text{fact } (\text{PRED } n)))
\]

We want the fix-point of this function:

\[
\text{FIX}(f) \equiv f(\text{FIX}(f))
\]
Definition of a fix-point operator:

\[ \text{FIX}(f) \equiv f(\text{FIX}(f)) \]

One step of \text{fact} is: \[ \lambda f . \lambda x . (\text{ISZERO } x) \downarrow 1 \downarrow (\text{MULT } x (f (\text{PRED } x))) \]

Call this \( F \). If we apply \text{FIX} to this, we get

\[ \begin{align*}
\text{FIX}(F)(n) &= F(\text{FIX}(F))(n) \\
\text{FIX}(F)(n) &= \lambda x . (\text{ISZERO } x) \downarrow 1 \downarrow (\text{MULT } x (\text{FIX}(F)(\text{PRED } x)))(n) \\
\text{FIX}(F)(n) &= (\text{ISZERO } n) \downarrow 1 \downarrow (\text{MULT } n (\text{FIX}(F)(\text{PRED } n)))
\end{align*} \]

If we rename \text{“FIX}(F)\text{” as \text{“fact”}, we have exactly what we want:

\[ \text{fact}(n) = (\text{ISZERO } n) \downarrow 1 \downarrow (\text{MULT } n (\text{fact}(\text{PRED } n))) \]

Conclusion: \text{fact} = \text{FIX}(F). (But we still need to define \text{FIX}.)
There are many fix-point combinators. Here is the simplest, due to Haskell Curry:

$$\text{FIX} = \lambda f \cdot (\lambda x. f (x x)) (\lambda x. f (x x))$$

Let’s prove that it actually works:

$$\text{FIX}(g) = (\lambda f \cdot (\lambda x. f (x x)) (\lambda x. f (x x))) g$$

$$\rightarrow_\beta ((\lambda x. g (x x)) (\lambda x. g (x x)))$$

$$\rightarrow_\beta g ((\lambda x. g (x x)) (\lambda x. g (x x)))$$

But this is exactly $g(\text{FIX}(g))$!
Functional Programming Concepts

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language.
- The key idea: do everything by composing functions.
  - no mutable state
  - no side effects
Necessary features, many of which are missing in some imperative languages

» 1st class and high-order functions
» serious polymorphism
» powerful list facilities
» structured function returns
» fully general aggregates
» garbage collection
So how do you get anything done in a functional language?

» Recursion (especially tail recursion) takes the place of iteration

» In general, you can get the effect of a series of assignments

\[
\begin{align*}
x & := 0 \quad \ldots \\
x & := \text{expr1} \quad \ldots \\
x & := \text{expr2} \quad \ldots \\
\end{align*}
\]

from \( f3(f2(f1(0))) \), where each \( f \) expects the value of \( x \) as an argument, \( f1 \) returns \( \text{expr1} \), and \( f2 \) returns \( \text{expr2} \)
Recursion even does a nifty job of replacing looping

\[
\begin{align*}
x & := 0; \quad i := 1; \quad j := 100; \\
\text{while } i < j \text{ do} & \\
& \quad x := x + i \times j; \quad i := i + 1; \\
& \quad j := j - 1 \\
\text{end while} & \\
\text{return } x
\end{align*}
\]

becomes \( f(0,1,100) \), where

\[
\begin{align*}
f(x, i, j) &= \text{if } i < j \text{ then } \\
& \quad f(x+i \times j, \ i+1, \ j-1) \text{ else } x
\end{align*}
\]
Thinking about recursion as a direct, mechanical replacement for iteration, however, is the wrong way to look at things

- One has to get used to thinking in a recursive style

Even more important than recursion is the notion of *higher-order functions*

- Take a function as argument, or return a function as a result
- Great for building things
Lisp also has (these are not necessary present in other functional languages)
  » homo-iconography
  » self-definition
  » read-evaluate-print

Variants of LISP
  » Pure (original) Lisp
  » Interlisp, MacLisp, Emacs Lisp
  » Common Lisp
  » Scheme
Pure Lisp is purely functional; all other Lisps have imperative features

All early Lisps dynamically scoped
  » Not clear whether this was deliberate or if it happened by accident

Scheme and Common Lisp statically scoped
  » Common Lisp provides dynamic scope as an option for explicitly-declared *special* functions
  » Common Lisp now THE standard Lisp
    • Very big; complicated (The Ada of functional programming)
- Scheme is a particularly elegant Lisp
- Other functional languages
  » ML
  » Miranda
  » Haskell
  » FP
- Haskell is the leading language for research in functional programming
As mentioned earlier, Scheme is a particularly elegant Lisp

- Interpreter runs a read-eval-print loop
- Things typed into the interpreter are evaluated (recursively) once
- Anything in parentheses is a function call (unless quoted)
- Parentheses are NOT just grouping, as they are in Algol-family languages
  - Adding a level of parentheses changes meaning
    - \((+ 3 4) \Rightarrow 7\)
    - \(((+ 3 4)) \Rightarrow \text{error}\)
    - (the ' \Rightarrow ' arrow means 'evaluates to')
A closure is a function whose execution has access to an external environment.

LISP was the earliest language to do closures, and it did them the other way (dynamic).

Static generally considered better; Scheme is basically LISP with closures done “right”.
- related to Lisp, first description in 1975
- designed to have clear and simple semantics (unlike Lisp)
- statically scoped (unlike Lisp)
- dynamically typed
  - types are associated with values, not variables
- functional: first-class functions
- garbage collection
- simple syntax; lots of parentheses
  - homogeneity of programs and data
- continuations
- hygienic macros
A Review/Overview of Scheme – Sample Scheme Session

\[
\begin{align*}
(+ & \ 1 \ 2) \\
\Rightarrow & \ 3 \\
( & 1 \ 2 \ 3) \\
\Rightarrow & \ \text{procedure application: expected procedure; given: 1} \\
& \ a \\
\Rightarrow & \ \text{reference to undefined identifier: } a \\
(\text{quote } (+ & \ 1 \ 2)) \ ; \ a \ \text{shorthand is } '(+ \ 1 \ 2) \\
\Rightarrow & \ (+ \ 1 \ 2) \\
(\text{car } '( & 1 \ 2 \ 3)) \\
\Rightarrow & \ 1 \\
(\text{cdr } '( & 1 \ 2 \ 3)) \\
\Rightarrow & \ (2 \ 3) \\
(\text{cons } 1 & \ '(2 \ 3)) \\
\Rightarrow & \ (1 \ 2 \ 3)
\end{align*}
\]
A Review/Overview of Scheme

- Scheme:
  - Boolean values #t and #f
    - Scheme has true and false values:
      - #t – true
      - #f – false
    - However, when evaluating a condition (e.g., in an if), any value not equal to #f is considered to be true.
  - Numbers
  - Lambda expressions
If variables do not have associated types, we need a way to find out what a variable is holding:
  » symbol?
  » number?
  » pair?
  » list?
  » null?
  » zero?

Different dialects may have different naming conventions, e.g., symbolp, numberp, etc.
A Review/Overview of Scheme

- Scheme (continued):
  - Quoting
    
    \((+ 3 4) \Rightarrow 7\)
    
    \((\text{quote } (+ 3 4)) \Rightarrow (+ 3 4)\)
    
    \'(+ 3 4) \Rightarrow (+ 3 4)\)

  - Mechanisms for creating new scopes
    
    \(\text{let } ((\text{square } (\text{lambda } (x) (* x x))) \text{ (plus +)})\)
    
    \(\text{let*}\)
    
    \(\text{letrec}\)
A Review/Overview of Scheme

- **Scheme (continued):**
  - Conditional expressions
    - (if (< 2 3) 4 5) ⇒ 4
    - (cond
      - ((< 3 2) 1)
      - ((< 4 3) 2)
      - (else 3)) ⇒ 3
  - Imperative stuff
    - assignments
    - sequencing (begin)
    - iteration
    - I/O (read, display)
A Review/Overview of Scheme – Simple Control Structures

- Conditional
  
  (if condition expr1 expr2)

- Generalized form
  
  (cond
   (pred1 expr1)
   (pred2 expr2)
   ...
   (else exprn))

Evaluate the pred's in order, until one evaluates to true. Then evaluate the corresponding expr. That is the value of the cond expression.

if and cond are not regular functions
define is also special:

\[
\text{(define (sqr n) (* n n))}
\]

The body is not evaluated; a binding is produced: \text{sqr} is bound to the body of the computation:

\[
\text{(lambda (n) (* n n))}
\]

We can define non-functions too:

\[
\text{(define x 15)} \\
\text{(sqr x)} \\
\Rightarrow 225
\]

\text{define} can only occur at the top level, and creates global variables.
Scheme standard functions (this is not a complete list):

- arithmetic
- boolean operators
- equivalence
- list operators
- symbol?
- number?
- complex?
- real?
- rational?
- integer?
A Review/Overview of Scheme – Lists and List Manipulation

- Expressions are either atoms or lists
- Atoms are either constants (e.g., numeric, boolean, string) or symbols
- Lists nest, to form full trees
- Syntax is simple because programmer supplies what would otherwise be the internal representation of a program:
  \[ (+ (* 10 12) (* 7 11)) \; ; \text{means} \; (10 \times 12 + 7 \times 11) \]

- A program is a list:
  ```scheme
  (define (factorial n)
      (if (eq n 0)
          1
          (* n (factorial (- n 1))))
  )
  ```

Three primitives and one constant:

- **car**: Get head of list
- **cdr**: Get rest of list
- **cons**: Prepend an element to a list
- **nil** or (): Null list

Add equality (\(=\) or eq) and recursion, and you’ve got yourself a universal model of computation
A Review/Overview of Scheme – Rules of Evaluation

- A number evaluates to itself
- An atom evaluates to its current binding
- A list is a computation:
  - must be a form (e.g., if, lambda), or
  - first element must evaluate to an operation
  - remaining elements are actual parameters
  - result is the application of the operation to the evaluated actuals
(car '(this is a list of symbols))
⇒ this

(cdr '(this is a list of symbols))
⇒ (is a list of symbols)

(cdr '(this that))
⇒ (that); a list

(cdr '(singleton))
⇒ (); the empty list

(car '())
⇒ car: expects argument of type <pair>; given ()
Operations like:

\[(\text{car} \ (\text{cdr} \ \text{x}s))\]
\[(\text{cdr} \ (\text{cdr} \ (\text{cdr} \ \text{y}s)))\]

are common. Scheme provides shortcuts:

\[(\text{cadr} \ \text{x}s) \ \text{is} \ (\text{car} \ (\text{cdr} \ \text{x}s))\]
\[(\text{cdddr} \ \text{x}s) \ \text{is} \ (\text{cdr} \ (\text{cdr} \ (\text{cdr} \ \text{y}s)))\]

Up to 4 a’s and/or d’s can be used.
(cons 'this '(that and the other))
⇒ (this that and the other)
(cons 'a '())
⇒ (a)

useful shortcut:

(list 'a 'b 'c 'd 'e)
⇒ (a b c d e)

equivalent to:

(cons 'a
   (cons 'b
     (cons 'c
       (cons 'd
         (cons 'e '())))))))
A Review/Overview of Scheme – What Lists are Made of

\[(\text{cons } \text{`a } \text{`(b)) } \Rightarrow (\text{a} \text{ b}) \text{ a list}\]

\[(\text{car } \text{`(a b)) } \Rightarrow \text{a}\]

\[(\text{cdr } \text{`(a b)) } \Rightarrow (\text{b})\]

\[(\text{cons } \text{`a } \text{`b) } \Rightarrow (\text{a} \cdot \text{ b}) \text{ a dotted pair}\]

\[(\text{car } \text{`(a \cdot b)) } \Rightarrow \text{a}\]

\[(\text{cdr } \text{`(a \cdot b)) } \Rightarrow \text{b}\]

A list is a special form of dotted pair, and can be written using a shorthand:

\'[\text{a b c} )\text{s shorthand for }'(\text{a} \cdot (\text{b} \cdot (\text{c} \cdot ()')))\]

We can mix the notations:

\'(\text{a b \cdot c) }\text{s shorthand for }'(\text{a} \cdot (\text{b} \cdot \text{c}))
A Review/Overview of Scheme – Recursion on Lists

(define (member elem lis)
  (cond
    ((null? lis) #f)
    ((eq elem (car lis)) lis)
    (else (member elem (cdr lis))))

Note: every non-false value is true in a boolean context.
Convention: return rest of the list, starting from `elem`, rather than `#t`. 
(define (map fun lis)
  (cond
    ((null? lis) '())
    (else (cons (fun (car lis))
                (map fun (cdr lis)))))

(map sqr (map sqr '(1 2 3 4)))
⇒ (1 16 81 256)
Basic let skeleton:

```
( let
  ((v1 init1) (v2 init2) ... (vn initn))
  body)
```

To declare locals, use one of the let variants:

- **let**: Evaluate all the inits in the current environment; the vs are bound to fresh locations holding the results.
- **let***: Bindings are performed sequentially from left to right, and each binding is done in an environment in which the previous bindings are visible.
- **letrec**: The vs are bound to fresh locations holding undefined values, the inits are evaluated in the resulting environment (in some unspecified order), each v is assigned to the result of the corresponding init. This is what we need for mutually recursive functions.
“A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls.”

```scheme
(define (factorial n)
  (if (zero? n) 1
      (* n (factorial (- n 1)))) ; not tail recursive
  ; stack grows to size n

(define (fact-iter prod count var)
  (if (> count var) prod
      (fact-iter (* count prod) ; tail recursive
                (+ count 1) ; implemented as loop
                var)))

(define (factorial n) (fact-iter 1 1 n)) ; OK
```
A Review/Overview of Scheme - Continuations

- A procedure with parameters to tell you what to do with an answer after computing it

- In Continuation Passing Style (CPS), instead of returning a result, you pass the result to the continuation that was passed to you

```
(define (mysqrt x) (sqrt x)) ; normal definition of mysqrt
(display (mysqrt 4)) ; normal call to display mysqrt of 4
(+ (mysqrt 4) 2) ; normal call to compute sqrt(4) + 2

(define (mysqrt x k) (k (sqrt x))) ; CPS definition of mysqrt
(mysqrt 4 display) ; CPS call to display mysqrt of 4
(mysqrt 4 (lambda (x) (+ x 2))) ; CPS call to compute sqrt(4) + 2

(define (factorial n) (if (<= n 1) 1 (* n (factorial (- n 1)))))  ; normal definition of factorial
(+ (factorial 4) 2) ; normal call to compute 4! + 2

(define (factorial n k) (if (<= n 1) (k 1) (factorial (- n 1) (lambda (ret) (k (* n ret)))))
(factorial 4 (lambda (x) (+ x 2))) ; CPS call to compute 4! + 2
```
To return multiple values: x, y, z, instead of packing it into a single object like \((x.(y.z))\), and later unpacking, simply call the continuation passing \(x, y, z\).

Instead of alternate exits (e.g. succeed and return \(x, y, z\) vs. fail and return nothing), use multiple continuations.
(define get-expr-prefix (lambda (lst ; a list of unparsed tokens
    success ; continuation(parsedexpr unparsed) parsedexpr : expr-tree of prefix of lst, unparsed : rest
    failure ; continuation(unparsed)
  )
  (letrec (make-closure (lambda (parsedexpr ; parsedexpr : expr parsed so far
      cont ; a continuation(handle-has-suffix, handle-has-no-suffix)
      ; handle-has-suffix: lambda(operator unparsed)
      ; handle-has-no-suffix : lambda(unparsed)
    )
      (letrec ((expr parsedexpr)
        (handle-has-suffix-method (lambda (operator unparsed …)
          (handle-has-no-suffix-method(lambda (unparsed) (success parsedexpr unparsed))
        )
      )
    )
    (define get-expr (lambda lst)
      (get-expr-prefix lst (lambda (parsed, unparsed) (if (empty? unparsed) (display parsed) (display 'error))
      (lambda(unparsed) (display 'error))
      )
    )
  )
  (letrec (got-term (lambda (parsedterm rest)
    (make-closure (cons 'expr (list parsedexpr operator parsedterm))
    (lambda (has-suffix has-no-suffix)
      (get-additive-operator rest has-suffix has-no-suffix)
    )
  )
    (no-term (lambda rest) (failure rest))
  )
  (get-term-prefix rest got-term no-term)

; Find the term following the +/- operator
; If there is one, extend the expression, save it in the closure,
; and once again try to find a suffix
; If there isn't one, this is a bad expression, e.g. A * B +

letrec (got-term (lambda (parsedterm rest)
    (make-closure (cons 'expr (list parsedexpr operator parsedterm))
    (lambda (has-suffix has-no-suffix)
      (get-additive-operator rest has-suffix has-no-suffix)
    )
  )
    (no-term (lambda rest) (failure rest))
  )
  (get-term-prefix rest got-term no-term)
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x) (if (negative? x) (exit x))) '(54 0 37 -3 245 19)) #t)) => -3

(define list-length (lambda (obj)
  (call-with-current-continuation
    (lambda (return)
      (letrec ((r (lambda (obj)
                      (cond ((null? obj) 0) ((pair? obj) (+ (r (cdr obj)) 1))
                                (else (return #f))))))
       (r obj)))))))

(list-length '(1 2 3 4)) => 4
(list-length '(a b . c)) => #f
These are ways to extend Scheme
They are invoked on static expressions rather than on values
Scheme introduced “hygienic macros”

» **Hygiene**: If a macro transformer inserts a binding for an identifier, the new binding will not capture other identifiers of the same name introduced elsewhere.

» **Referential Transparency**: If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.
This allows you to create an arbitrary expression representing a Scheme program as a data object, and then execute it.

There is an interpreter that one can write in Scheme that emulates eval: what is surprising is how short the program is – a Scheme interpreter in Scheme is a shorter program than your homework program!
(define isort (  
  lambda (l)  
  (letrec ( ; defines a list of bindings (here just 1)  
    (insert ( ; inserts item x in sorted order to list l  
      lambda (x l)  
      (if (null? l)  (list x)  
          (if (<= x (car l)) (cons x l)  
              (cons (car l) (insert x (cdr l)))) ; means this insert  
          )))  
    ; the below is executed in the context of the bindings  
    (if (null? l) nil (insert (car l) (isort (cdr l)))))  
  )  
))  
(isort (3 20 13 2))
We'll invoke the program by calling a function called 'simulate', passing it a DFA description and an input string.

- The automaton description is a list of three items:
  - start state
  - the transition function
  - the set of final states

- The transition function is a list of pairs:
  - the first element of each pair is a pair, whose first element is a state and whose second element is an input symbol
  - if the current state and next input symbol match the first element of a pair, then the finite automaton enters the state given by the second element of the pair
A Review/Overview of Scheme
Example program - Simulation of DFA

(define simulate
  (lambda (dfa input)
    (cons (current-state dfa) ; start state
      (if (null? input)
        (if (imfinal? dfa) '(accept) '(reject))
        (simulate (move dfa (car input)) (cdr input))))))

;; access functions for machine description:
(define current-state car)
(define transition-function cadr)
(define final-states caddr)
(define imfinal?
  (lambda (dfa)
    (memq (current-state dfa) (final-states dfa)))))

(define move
  (lambda (dfa symbol)
    (let ((cs (current-state dfa)) (trans (transition-function dfa))
      (list
        (if (eq? cs 'error)
          'error
          (let ((pair (assoc (list cs symbol) trans)))
            (if pair (cadr pair) 'error))))) ; new start state
      trans ; same transition function
      (final-states dfa))))) ; same final states

Figure 10.1 Scheme program to simulate the actions of a DFA. Given a machine description and an input symbol i, function move searches for a transition labeled i from the start state to some new state s. It then returns a new machine with the same transition function and final states but with s as its "start" state. The main function, simulate, tests to see if it is in a final state. If not, it passes the current machine description and the first symbol of input to move, and then calls itself recursively on the new machine and the remainder of the input. The functions cadr and caddr are defined as (lambda (x) (car (cdr x))) and (lambda (x) (car (cdr (cdr x))), respectively. Scheme provides a large collection of such abbreviations.
A Review/Overview of Scheme

Example program - Simulation of DFA

\begin{figure}
\centering
\begin{tikzpicture}
\node[state] (q0) at (0,0) {$q_0$};
\node[state] (q1) at (1,1) {$q_1$};
\node[state] (q2) at (1,-1) {$q_2$};
\node[state] (q3) at (0,-2) {$q_3$};
\draw[->] (q0) edge [loop above] node {$1$} (q0);
\draw[->] (q0) edge [bend right] node {$0$} (q2);
\draw[->] (q0) edge [bend left] node {$0$} (q3);
\draw[->] (q1) edge [loop below] node {$1$} (q1);
\draw[->] (q1) edge [bend right] node {$1$} (q3);
\draw[->] (q1) edge [bend left] node {$0$} (q2);
\draw[->] (q2) edge [loop left] node {$0$} (q2);
\draw[->] (q2) edge [bend right] node {$1$} (q3);
\draw[->] (q2) edge [bend left] node {$0$} (q0);
\draw[->] (q3) edge [loop right] node {$1$} (q3);
\draw[->] (q3) edge [bend right] node {$0$} (q2);
\draw[->] (q3) edge [bend left] node {$0$} (q1);
\end{tikzpicture}
\caption{DFA to accept all strings of zeros and ones containing an even number of each.}
\end{figure}

At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 10.1.
Evaluation Order Revisited

- **Applicative order**
  - what you're used to in imperative languages
  - usually faster

- **Normal order**
  - like call-by-name: don't evaluate arg until you need it
  - sometimes faster
  - terminates if anything will (Church-Rosser theorem)
In Scheme
  » functions use applicative order defined with lambda
  » special forms (aka macros) use normal order defined with syntax-rules

A *strict* language requires all arguments to be well-defined, so applicative order can be used

A *non-strict* language does not require all arguments to be well-defined; it requires normal-order evaluation
Lazy evaluation gives the best of both worlds

But not good in the presence of side effects.

» delay and force in Scheme

» delay creates a "promise"
Higher-order functions

» Take a function as argument, or return a function as a result

» Great for building things

» Currying (after Haskell Curry, the same guy Haskell is named after)
  • For details see Lambda calculus on CD
  • ML, Miranda, and Haskell have especially nice syntax for curried functions
Advantages of functional languages

- lack of side effects makes programs easier to understand
- lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
- lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
- programs are often surprisingly short
- language can be extremely small and yet powerful
Problems

- difficult (but not impossible!) to implement efficiently on von Neumann machines
  - lots of copying of data through parameters
  - (apparent) need to create a whole new array in order to change one element
  - heavy use of pointers (space/time and locality problem)
  - frequent procedure calls
  - heavy space use for recursion
  - requires garbage collection
  - requires a different mode of thinking by the programmer
  - difficult to integrate I/O into purely functional model
1st class procedures
Dynamically created procedures
Based on lambda calculus over atoms and pairs; by convention, lists are pairs \(<\text{head}, \text{rest}>\)
Continuations
Automatic garbage collection
Applicative style: binding, no update, no side effects (but there are exceptions to this, the “set!” operator)
Static scoping, but no static typing!
Simple syntax
- \((\text{afunction} \ \text{arg1} \ \text{arg2} \ \ldots)\) ; function application
- Special forms, e.g. \((\text{if} \ \ldots) \ (\text{quote} \ (x \ y)) \ (\text{lambda} \ (x) \ (* \ x \ x))\)
Readings

» Chapter Section 10 including 10.6.1 on the CD

Programming Assignment #2

» See Programming Assignment #2 posted under “handouts” on the course Web site

» Due on July 24, 2014