CSCC24 — Functional Programming — Scheme

Carolyn MacLeod

winter 2012

\footnote{Based on slides from Anya Tafliovich, and with many thanks to Gerald Penn and Prabhakar Ragde.}
One of the most important and fascinating of all computer languages is Lisp (standing for "List Processing"), which was invented by John McCarthy around the time Algol was invented. – Douglas Hofstadter, Godel, Escher, Bach
What is Scheme

- A Functional Programming Language
  - actually...
What is Scheme

- A Functional Programming Language
  - actually...
    - an imperative programming language

  - an imperative programming language
What is Scheme

• A Functional Programming Language
  • actually...
    an imperative programming language
    with a functional core
What is Scheme

• A Functional Programming Language
  • actually...
    an imperative programming language
    with a functional core

• static scoping
• dynamic typing
• clear and simple syntax
• clear and simple semantics (for the core): \(\lambda\)-calculus
• uniform treatment of program and data
• implementations properly tail-recursive
• functions are values: created dynamically, stored, passed as parameters, returned as results, etc.
• pass-by-value
Why Scheme?

OK, so why should we care about Scheme? Scheme is in the Lisp family of languages. (There are some significant differences, but Scheme is a Lisp dialect) This family of languages has many descendents you might have heard of, without realizing the relationship:

- F# now ships with Visual Studio
- Clojure targets the Java Virtual Machine
- Hadoop and Google’s MapReduce model for parallel programming is based on the functional combinators “map” and “reduce” found in Lisp and Scheme
- It’s still a good fit for creating a language and parsing it, for example, LilyPond, an open-source music typesetting program.
- A Racket-based language is used to implement content for the ”Uncharted” video game series.
Why Scheme?

Even Java owes something to this family of languages:

_We were not out to win over the Lisp programmers; we were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp._ – Guy Steele, Java spec co-author
A brief history of Scheme

- It starts with Lisp (from List Processing) (1958)
  - Origin of the if-then-else and conditional construct (!)
  - First implementation recursion in a programming language
  - Function type (allowing for higher order functions)
  - Programs are a tree of expressions
  - Lists are fundamental
  - Garbage collection - easy with this model of computation

- Simula and Smalltalk influenced a model of computation around "actors".

  You don’t add 3 to 2 and get 5; instead, you send 3 a message asking it to add 2 to itself
A brief history of Scheme

- AI languages at MIT (following in the footsteps of Lisp) lead eventually to Scheme
  - Planner (1969)
  - Conniver (1972)
  - Schemer (1975)...which with a 6-character limit becomes Scheme

- Scheme started with a small Lisp interpreter
- Added exactly two constructs
  - A way to make actors
  - A way to pass messages
- Add static scoping and proper recursion, and we end up with Scheme
- The simplicity was not intentional - and hopefully like Scheme itself, your programs will get simpler as you work with them, and better understand the problem.
Why Scheme?

But if you want my opinion...
pure functional languages

(We will revisit this in greater detail after we learn a bit of Scheme.) Programs are viewed as collections of functions.

Execution of programs is viewed as evaluation.
pure functional languages

- **Referential transparency:**

  The value of a function application is independent of the context in which it occurs.

  A language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program.

  - Value of \( f(a, b, c) \) depends only on the values of \( f \), \( a \), \( b \) and \( c \).
  - It does not depend on the global state of computation.
  - Main advantage: facilitates reasoning about programs and applying program transformations.
pure functional languages

- **No assignment statement:**

  A variable may not have a value, or its value might be a function that has not been applied to its arguments yet, but variables in pure FP are said to be *logical* in that, having acquired a value in the course of an evaluation, they retain that value until the end of evaluation.

  This is similar to the manner in which variables are used in mathematics. Remember when you had to unlearn your mathematics training to make sense of \( x := x+1 \)? In FP, this is anathema!

  This is one example of how a pure functional language uses denotation semantics over operational semantics.
pure functional languages

• **No side effects:**

Function calls have no side effects.
  • No need to consider global state.
  • Programs are easier to reason about.
  • Programs are easier to prove correct.
  • Historically, imperative programming had to refer to Turing machines to talk about state, however there are very good techniques now.
pure functional languages

- Functions are first-class values:
  - Can be returned as the value of an expression.
  - Can be passed as an argument.
  - Can be put in a data structure as a value.
  - Unnamed functions exist as values.
pure functional languages

- **recursion vs. iteration:**

  In pure FP, recursion is generally preferred as a matter of style to iterative constructs like while-loops or for-loops. This makes programs
  - easier to prove correct,
  - easier to conceptualize as functions:

    Recursions, like functions, identify the structure of one or more arguments as the only values upon which the computation (and termination) depends.
pure functional languages

• a higher-level language:

• All storage management is implicit:
  we don’t have to think about how program state maps to a computer's memory (or at least, not until we want to write super-efficient code). That means no assignment, no new/free calls to manage our own memory, and no pointers.

• The state of a computation is much easier to think about.
Scheme expressions

- A number is an expression. More generally, any Scheme value is an expression.

\[
\text{\texttt{<expr> ::= <val>}}
\]
Scheme expressions

- A number is an expression. More generally, any Scheme value is an expression.

\[ \text{expr} ::= \text{val} \]

- That value can be computed by writing a list. The first element of the list is is a function, all other elements are arguments to this function.

\[ \text{expr} ::= ( \text{id} \ \text{expr} \ ... ) \]

  - \((+ 1 2)\)
  - \((+ (* 3 4) (- 6 5))\)
Scheme expressions

- A number is an expression. More generally, any Scheme value is an expression.

\[
\text{<expr>} ::= \text{<val>}
\]

- That value can be computed by writing a list. The first element of the list is a function, all other elements are arguments to this function.

\[
\text{<expr>} ::= (\ \text{id} \ \text{<expr>} \ldots)
\]

  - (+ 1 2)
  - (+ (* 3 4) (- 6 5))

Scheme provides some primitive (built-in) numeric functions (e.g. \(\slash\), quotient, remainder, expt).
Scheme data types

- **number**: \{ 5, -7, 3.14, ... \}
  Includes integers, rationals, reals, and complex numbers. These types are not disjoint, they form a hierarchy.
Scheme data types

- **number**: \{ 5, \(-7\), 3.14, \... \}
  Includes integers, rationals, reals, and complex numbers. These types are not disjoint, they form a hierarchy.

- **symbol**: \{a, ABC, foo, \... \}
Scheme data types

- **number**: \{ 5, -7, 3.14, ... \}
  Includes integers, rationals, reals, and complex numbers. These types are not disjoint, they form a hierarchy.

- **symbol**: \{a, ABC, foo, ...\}

- **pair**: \{(1 . 2), (1.12 . -4), (1 . a), ...\}
Scheme Basics

• A **pair** (also known as a cons pair) is the building block of lists in Scheme. I’ll draw you a diagram.
  
  • Constructor: cons.
    
    • (cons 1 2)
    • (cons (cons 1 2) 3)

  • Selectors: car, cdr.
    
    car - ”contents of address register”
    cdr - ”contents of decrement register”

    These reflect instructions on the IBM 704 machine, the address and decrement registers being 15-bit fields of a memory word.
    
    • (car ’(1 . 2))
    • (cdr ’((1 . 2) . 3))
    • (car 1)
    • (cdr ’())
Where we are

- A brief overview of the features of Scheme and of pure functional programming languages in general. All of these ideas will be revisited in more detail.
- History/context of Scheme.
- Scheme: Values, lists, and pairs. car/first cdr/rest and cons, cons pairs.
- Today - More Scheme. We will visit conditionals, definitions, functions in more detail, and we will go through some simple examples.
- Today or next class - Recursion and tail recursion.
- Coming up: Higher order functions.
- Coming up: Lambda calculus.
Scheme data types

- **list**: 
  
  \{(\), (1), (a b), (1 2 foo 3.14), \ldots\}

  - The only list that is not a pair is \((\)\).
  - If \(L\) is a list and \(V\) is a Scheme value, then \((\text{cons } V \ L)\) is a list.

    - \((\text{cons } 1 \ '(())\)
    - \((\text{cons } '(1) \ '(2 3))\)
    - \((\text{car } '(1))\)
    - \((\text{cdr } '(1))\)
    - \((\text{cdr } '(1 2 3))\)

- Scheme has better selector names that are more intuitive: first, rest. Even Lisp/Scheme’s creators like them better, so I only mention car/cdr for historical interest, and because some tutorials and resources use them.

- The type predicate is list?
values and evaluation

- number: it’s own value
values and evaluation

- number: it’s own value

- (quote expr): expr
  
  ’expr : shortcut for (quote expr)
values and evaluation

• number: it’s own value

• \((\text{quote expr})\): \(\text{expr}\)
  \(\text{'expr}\): shortcut for \((\text{quote expr})\)

• lists:
  • \((f . (\text{arg0} \ldots \text{argN}))\)
    has value
    \(f(\text{arg0}, \ldots, \text{argN})\)
  This is called \textit{prefix notation}. 
values and evaluation

- lists:
  - \((f . (\text{arg}0 \ldots \text{arg} N))\)
    - has value
    - \(f(\text{arg}0, \ldots, \text{arg}N)\)

Not all lists have values:
\((1 \ 2 \ 3) \Rightarrow ?\)
values and evaluation

- lists:
  - \((f . (\text{arg}0 \ldots \text{arg} N))\) has value \(f(\text{arg}0, \ldots, \text{arg}N)\)

Not all lists have values:
\((1 \ 2 \ 3) \Rightarrow ?\)

1 evaluates to 1, which is not a procedure and cannot be applied to 2 and 3.
Read-Eval-Pint Loop

Read: Read input from user.

Eval: Evaluate input.

Lists are evaluated as follows:
(f arg₀ arg₁ ... argₙ)

1. Evaluate f (a procedure) and each argᵢ (in an unspecified order).
2. Apply procedure to argument values.

Print: Print resulting value:
the result of the procedure application.
Read-Eval-Print Loop example

(cons 'a (cons 'b '(c d))) => (a b c d)

1. Read the input
   (cons 'a (cons 'b '(c d))).
2. Evaluate 'a: obtain a.
3. Evaluate (cons 'b '(c d)):
   3.1 Evaluate '(c d): obtain (c d).
   3.2 Evaluate 'b: obtain b.
   3.3 Evaluate cons: obtain a procedure.
   3.4 Apply the cons procedure to b and (c d): obtain (b c d).
4. Evaluate cons: obtain a procedure.
5. Apply the cons procedure to a and (b c d):
   obtain (a b c d).
6. Print the result of the application:
   (a b c d).
Read-Eval-Pint Loop — Racket (PLT Scheme)

Read: Read input from user.

Eval: Evaluate input.
Lists are evaluated as follows:
(f arg_0 arg_1 ... arg_n)
1. Evaluate f (a procedure).
2. Evaluate each arg_i (in left-to-right order).
3. Apply procedure to argument values.

Print: Print resulting value:
the result of the procedure application.
Read-Eval-Print Loop example — Racket (PLT Scheme)

\[(\text{cons 'a (cons 'b '(c d))) } \Rightarrow (\text{a b c d})\]

1. Read the input
   \[(\text{cons 'a (cons 'b '(c d)))}\].
2. Evaluate \text{cons} : obtain a procedure.
3. Evaluate \text{'a} : obtain a.
4. Evaluate \text{(cons 'b '(c d))}:
   4.1 Evaluate \text{cons} : obtain a procedure.
   4.2 Evaluate \text{'b} : obtain b.
   4.3 Evaluate \text{'(c d)} : obtain \text{(c d)}.
   4.4 Apply the \text{cons} procedure to b and \text{(c d)} : obtain \text{(b c d)}.
5. Apply the \text{cons} procedure to \text{a} and \text{(b c d)} : obtain \text{(a b c d)}.
6. Print the result of the application:
   \text{(a b c d)}. 
evaluation by substitution

(+ (* 3 4) (− 6 5))
=> (+ 12 (− 6 5))
=> (+ 12 1)
=> 13

*Substitution model:* An expression is reduced to a value by repeatedly finding the leftmost expression ready for substitution (all arguments are values) and replacing it with its value.

The substitution model provides good intuition for the pure subset of Scheme.
A definition \emph{binds} a name to a value.

\begin{itemize}
\item (define x 7); binds x to 7
\item (define y (* x x)); binds y to 49
\end{itemize}

A subsequent use of the name results in a substitution of the associated value.
These are not variables as you know them.
special forms

special forms: ( syntactic-keyword expr )

Some syntactic keywords: and, or, if, cond.

<expr> ::= (and { <expr> } )
<expr> ::= (or { <expr> } )
special forms — if

Syntax:
<expr> ::= (if <expr> <expr> <expr>)

Semantics:

(if condition expr0 expr1)
  1. Evaluate condition.
  2. If result is not #f, evaluate expr0.
  3. Otherwise, evaluate expr1.

> (if false (car '()) 42)
42
special forms — cond

Syntax:
<expr> ::= (cond {(<expr> <expr>)} [(else <expr>)]

Semantics:

(cond (cond0 expr0)
     (cond1 expr1)
     ....
     (else exprN))

- Evaluate cond0.
- If the result is not #f, evaluate expr0.
- Otherwise, evaluate cond1.
- If the result is not #f, ...

More syntactic keywords: define, quote, lambda.
Scheme Functions

The last piece you need to really get going in Scheme are the functions:

- procedure
- type predicate: procedure?

Syntax:
<expr> ::= <function>
<function> ::= (lambda (\{<id>\}) <expr>)

For example:
(l lambda (x) (+ x 1))
You may be familiar to anonymous (lambda) functions in Python, and they’re of immense use in Scheme. However, we want to give our functions names so we can use them repeatedly.

```scheme
> (define add (lambda (x y) (+ x y)))
> add
#<procedure:add>
> (add 2 3)
> 5
```
Scheme data types

- number: 5, -7, 3.14, 3+4i includes integers, rationals, reals, and complex numbers: these types are not disjoint.

Some useful procedures:

- type predicates: number?, integer?, real?, complex?
- other predicates: positive?, negative?, zero?, even?, odd?, ...
- arithmetic operations: +, -, *, /, floor, remainder, ...
- comparison predicates: =, <, <=, >, >=, eq?
Scheme data types

- **boolean**: \{#t, #f\}.
  - Useful synonyms: true, false.
  - Type predicate: boolean?.
  - Useful procedures: not, boolean=?.
indentation

Write code so that it is readable by other human beings.

> (define max
    (lambda (x y)
        (if (> x y) x y)))
> (max 3 4)

Use a good editor: it will take care of indentation for you.
basic data types

- pair: (1 . 2), (#t . #f), (1 . #t)
- list: (), (1), (a b), (1 2 foo 3.14)
- predicates: list?, pair?, null? (empty?)
- selection: car, cdr, cadr, caadr, cadddr, ...
- list selection: first, rest, second, third, ...
- construction: cons, cons*, list, ...
- more: length, append, member, ...
- higher-order: map, reduce, foldr, foldl, filter...
Racket (PLT-Scheme) on mathlab

- Type drracket. This starts up DrRacket, the Racket programming environment.
- Choose “Use the language declared in the source” in the Language menu.
- Start your file with #lang scheme.
- Browse the documentation in HelpDesk.