Course Contents

- Introduction
- Syntax and semantics
  - Regular expression
  - BNF
  - CFG
- Functional programming (in Scheme)
- O-O programming (in Python)
- Logic programming (in Prolog)

Today

- Scheme
  - Introduction
  - Basic data types
  - Properties of Pure Functional Languages

Functional Programming: Introduction

- LISP
  - "LISr Processor"
  - A functional language developed by John McCarthy in the late 1950s for use in AI.
  - Semantics (meaning) based on \( \lambda \)-calculus.
    - The \( \lambda \)-calculus is a theoretical model of computation
  - Functions operate on lists or atomic symbols.
    - Programs consist of "S-expressions"
    - Five basic functions: cons, car, cdr, equal, atom
    - One conditional construct: cond
  - Programs and data are both S-expressions.

Introduction (Cont.)

- Common LISP
  - LISP implementations began to stray
  - Semantics were redefined in Common LISP.
  - Common LISP is now the standard version of LISP.
    - Designed by committee in 1980s.
    - Simple syntax, but a large language.
- Scheme
  - Created in 1975 by Gerald Sussman and Guy Steele.
  - A dialect of LISP
  - Simple syntax, small language
  - Close to the initial semantics of LISP
  - Allows functions to be first-class object
The Scheme We'll Use

• There are many flavours of Scheme, with quite a bit of variation among them.
  – But not as much variation as among the LISPs.
• We'll use MzScheme.
  – A subset of PLT Scheme
  – Use the command "mzscheme" to launch the MzScheme interpreter on ECF.
  – Version 4.2 is installed on ECF.
• DrScheme is an IDE that works with PLT Scheme.
  – Use the command "drscheme" on ECF.
  – Make sure the selected language is Module.
  – PLT Scheme (including DrScheme) can be downloaded from http://download.plt-scheme.org/
  – Your assignments must run using DrScheme on ECF.

A Scheme Program

• A program consists of
  – a set of function definitions
  – an expression to be evaluated
• A simple example that defines function abs-val and then calls this function (note that ">": the interpreter's prompt):

> (define (abs-val x)
  (if (>= x 0) x
  (- x)))
> (abs-val (- 3 5))

2
The Basic Structure of a Scheme (or LISP) Program

- A program consists of S-expressions ("symbolic expressions") written in parenthesized prefix form.
  - "Prefix form": The name of the function appears before the arguments to the function, even for mathematical functions like +, −, *, and /.
  - The general form of an S-expression in prefix-form is:
    \[(<\text{function}> <\text{arg1}> <\text{arg2}> \ldots <\text{argn}> )\]

- Examples of S-expressions in prefix-form:
  \[(+ 4 5)\] ; A simple example.

Note that ";" denotes a comment in Scheme.

\[(+ (* 3 4 5) (- 5 3))\]

S-expressions

- To evaluate an expression:
  - Evaluate the \(<\text{function}>\) to a value that is a function.
  - Evaluate each \(<\text{arg1}>...<\text{argn}>\) to obtain its value.
  - Apply the function value to these argument values.

- Evaluating our examples:
  \[(+ 4 5)\]
  4 and 5 evaluate to themselves.
  Applying "+" to 4 and 5 results (not surprisingly) in the value 9.

\[(+ (* 3 4 5) (- 5 3))\]
Evaluate \((* 3 4 5)\) to obtain the value 60.
Evaluate \((- 5 3)\) to obtain value 2.
Evaluate \((+ 60 2)\) to obtain value 62.
19

Basic Data Types

- Number: 5, -7, 3.14, 1+2i
  Including integers, rationals, reals, and complex numbers: these types are not disjoint
- Symbol: a, ABC, foo
- Pair: \((1 . 2)\)
- Constructors -- cons
  \[(\text{cons } 1 2)\]
  \[(\text{cons } (\text{cons } 1 2) 3)\]
  \[(\text{cons } 1 (\text{cons } 2 3))\]
- Selectors -- car cdr
  \[(\text{car } '(1 . 2))\]
  \[(\text{cdr } '(1 . 2 . 3))\]
Basic Data Types

- List: (), (1), (a b), (1 2 3)
  - the only list that is not a pair is ()
  - a sequence of pairs, such that each pair's cdr is next pair, and
    cdr of last pair is ()

- Examples
  (cons 1 ())
  (cons '(1) '(2 3))
  (car '(1))
  (cdr '(1))
  (cdr '(1 2 3))

Values and Evaluation

Objects and their values:
- number: it's own value
- symbol: may be undefined
  (define x 3)
- (quote expr) : expr
  'expr : shortcut for (quote expr)
- Lists:
  '() has value ()
  (f arg0 ... argn) has value f(arg0, ..., argn)

READ-EVAL-PRINT Loop
(REPL)

READ: Read input from user
EVAL: Evaluate input
List elements are evaluated from left to right:
(f arg0 arg1 ... argn)
1. evaluate f: obtain a procedure
2. evaluate each arg0 ... argn
3. apply procedure to argument values
PRINT: Print resulting value: the result of the procedure application

This is called applicative-order evaluation.

Example

> (cons 'a (cons 'b '(c d)))
;Value 1: (a b c d)

Read the input (cons 'a (cons 'b '(c d)))
- Evaluate cons : obtain a procedure
- Evaluate 'a : obtain a
- Evaluate (cons 'b '(c d)):
  - Evaluate 'b: obtain b
  - Evaluate '(c d): obtain (c d)
  - Apply cons to b and '(c d): obtain (b c d)
- Apply the cons procedure to a and (b c d) : obtain (a b c d)
- Print the result of the application: (a b c d)
Basic Data Types (Cont.)

- Number: 5, -7, 3.14, 3+4i
  - Some useful procedures:
    - Type predicates: number?, integer?, real?, complex?
    - Other predicates: positive?, negative?, zero?, even?, odd? ...
    - Arithmatic operations: +, -, *, /, floor, remainder, ...
    - Comparison predicates: =, <, >, <=, >=

- Symbol: a, ABC, foo
  - Type predicate: symbol?

- Boolean: #t, #f
  - Useful predicates: boolean?, boolean=?, not, and, or

Basic Data Types (Cont.)

- Procedure: (lambda (arg0 ... argN) expr)
  > (lambda (x) (+ x 1))
  ;Value 11: #[compound-procedure 11]

> ((lambda (x) (+ x 1)) 42)
;Value: 43

> ((lambda (x y) (if (> x y) x y)) 42 0)
;Value: 42

Basic Data Types (Cont.)

- Pair: (1 . 2), (#t . #f), (1 . #t)
- List: (), (1), (a b), (1 2 foo 3.14)
  - predicates: list?, pair?, null?
  - selection: liscar, cdr, cadr, caadr, caddr, ...
  - construction: cons, cons*, list, ...
  - more: length, append, member, ...
  - higher-order: map, reduce, fold-right, fold-left, ...

The Spirit of Functional PLs

- Programs are viewed as collections of functions.
- Execution of programs is viewed as evaluation.
Pure Functional Languages

- **Manifest Interface Principle** (of Programming Languages): All interfaces should be apparent (manifest) in the syntax.
  - A pure function is one that simply uses its input arguments to compute a return value, without performing any side-effects.
  - Side-effects are changes to the system’s computational state that could affect future calls to itself or other functions.

Pure Functional Languages

- **Referential Transparency**: syntactically identical expressions mean the same thing, (i.e. return the same result when evaluated) regardless of WHERE they appear in a program.
  - In a language that only uses pure functions, any function call with instantiated arguments, e.g. (f 5 10), **ALWAYS** returns the same value and hence *means the same thing* in the context of a particular program.

Pure Functional Languages

- When all functions are pure, referential transparency and the manifest interface principle are upheld, and thus:
  - No side-effects, programs are much easier to formally analyze
    - a) Once we know the local behaviors of functions, we can reason about the system in terms of independent function calls, without having to worry about the (side) effects of these calls on future calls.
    - b) We can identify identical functional calls as equivalent objects and simplify the code accordingly.
    
      E.g. (+ (f 5 10) (f 5 10)) simplifies to (* 2 (f 5 10)) for any numeric function f
  - Programs are much easier to DEBUG!!!

    You can understand programs by just looking at the static source code. You need not think about the underlying computational states and how they are affected by program dynamics.

    **What you see is what you get!**

    (Mitchell, page78 for more)

Pure Functional Languages

- **to assign is bad... (no assignment)**

  - When an assignment statement is applied to variables (i.e. memory locations) that
    - a) will be maintained AFTER the function call is completed,
    - b) will be used for their values during later function calls (to the same or other functions),

    *It violates referential transparency and destroys one's ability to statically analyze source code (both formally or intuitively).*

  - Example:
    
    ```scheme
    (define g 10) ; define a global variable, g.
    (define (f a) ; define function f, with one argument, a.
      (+ a g)) ; Scheme's assignment operator, meaning g = g*g
    (set! g (* g g))
    ]=> (f 7) 107
    ]=> (f 7) 10007 ; BADD 😞
    ```
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

• **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 reason about and 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

Why Lisp is successful?
Motivation, simple, unambiguous execution model, theoretical

What's the contributions of Lisp?
  Recursive functions, lists, programs as data, garbage collection (Mitchell page 39-40)

Scheme: programming style

• **Use special suffix:**
  – "?" for predicates (i.e. functions returning #t or #f, e.g. `member?`)
  – "!*" for any procedure with "side effects" (i.e. changes of bindings for non-local variables, e.g. `set!`)

• **Procedure definitions should be brief**
  – Oriented towards a single, well-defined task
  – Should be split into a number of subtasks if > 1 page

• **Comments:**
  – ; for comments on the same line with code
  – ;; for comments that run from beginning of line
  – ;;; for comments that describe the contents of the file (usually first in file)

• **Indentation:**
  – Indent procedure definitions like this, with the body starting a new line, and indented a few characters
    (define (foo) 15)
• Deeply nested cars and cdrs are often difficult to understand, and should therefore be avoided.

• Since Scheme is a dynamically typed language, the names of parameters should reflect their value.

• Most general guidelines on programming style also apply to Scheme programs.