Overview

Lecture 2 provided a general overview of the main syntax and language constructs of MyPL. Before the next lecture, do the following.

1. *Environment Set Up and start HW-0.* Ensure your programming environment is set up and working (Java and Maven), obtain the HW-0 starter code, and work on HW-0. Note that the goal of HW-0 is for you to be ready to start on HW-1 as soon as it is posted (next Friday).
2. *Study the MyPL Syntax from Lecture 2.* Using the lecture 2 notes, study and ensure you know the MyPL syntax and constructs. You will be asked questions on quizzes, exams, etc., to create valid MyPL programs. Also, since this is the language we will be implementing, you must have a strong understanding of the syntax and constructs of MyPL!
3. *Ask one good question about the MyPL language on Piazza.* By the end of the day on Sunday, you must create a new Piazza post that asks one interesting / useful question concerning the MyPL language. Be sure your question is labeled with the "mypl" folder (you will see the available folders once you click the "New Post" button). Prior to asking your question, be sure you look over the other questions asked so that you don't ask the same question as another student.

MyPL Examples from Class

The following examples were used as part of the demo of MyPL on Friday. These are provided here for your reference and to help you better understand the syntax and constructs in MyPL. Make sure you understand the programs and you are able to write basic (and similar) MyPL programs from scratch (e.g., on a whiteboard or using pen and paper). An example of a simple program you should be able to write would be one that prompts a user for input until they enter a specific number, and then print the sum of the numbers entered (which is very similar to the first example program below). *A good way to learn and study the syntax would be to work with a partner and quiz each other on the syntax (e.g., at a whiteboard).*

Note that these programs are only meant to be basic examples of the MyPL syntax and supported language constructs. They are not meant to be examples well-designed programs that use good coding practices!

Example 1: Sum to Prime. This program prompts a user for integer values until a prime value is entered. The program outputs the sum of the non-prime numbers that were given by the user.

```
#=============================================================================
# Classic sum-to-prime program
#=============================================================================
```

```
#-------------------------------------------------------------------------
# Check that n is a prime number
#-------------------------------------------------------------------------
bool is_prime(n: int) {
  var m: int = n / 2
  var v: int= 2
  while v <= m {
    var r: int = n / v
    var p: int = r * v
    if p == n {
      return false
    }
    v = v + 1
  }
  return true
}

#-------------------------------------------------------------------------
# Driver program
#-------------------------------------------------------------------------
void main() {
  println("Please enter integer values to sum (prime number to quit)")
  var sum: int = 0
  while true {
    print("Enter an int: ")
    var val: int = int_val(readln())
    if is_prime(val) {
      println("The sum is: " + str_val(sum))
      println("Goodbye!")
      return null
    }
    sum = sum + val
  }
}
```

Example 2: Recursively Computing Fibonacci Numbers. Note that this is a really bad way to compute fibonacci numbers, however, it is a good example of recursion (since it involves two recursive calls at each recursive step).

```
#=========================================================================
# Compute and print some fibonacci numbers!
#=========================================================================


#-------------------------------------------------------------------------
# compute the nth fibonacci number recursively
# note: this is one of the worst ways to compute fibonacci numbers
# (with its O(2^n) time complexity)!
#-------------------------------------------------------------------------
int fib(n: int) {
  if n <= 1 {
    return n
```

```
  }
  else {
    return fib(n - 2) + fib(n - 1)
  }
}


#-------------------------------------------------------------------------
# helper function to print the nth fibonacci number
#-------------------------------------------------------------------------
void print_result(n: int, r: int) {
  print("fib(")
  print(n)
  print(") = ")
  print(r)
  print("\n")
}

#-------------------------------------------------------------------------
# print the first 32 fibonacci numbers:
#   0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
#   1597, 2584, ...
#-------------------------------------------------------------------------
void main() {
  var n: int = 0
  var m: int = 32
  while n < m {
    print_result(n, fib(n))
    n = n + 1
  }
}
```

Example 3: Non-Balanced Binary Search Tree. This is a simple example of using struct objects within MyPL for a classic binary-search tree implementation. The program just creates a simple tree and displays it.

```
#=========================================================================
# Basic (non-balanced) binary search tree implementation
#=========================================================================


#-------------------------------------------------------------------------
# tree node that holds an int value
#-------------------------------------------------------------------------
struct node {
  value: int,
  right: node,
  left: node
}


#-------------------------------------------------------------------------
# create a new tree
#-------------------------------------------------------------------------
```

```
node make_tree(val: int) {
  return new node(val, null, null)
}


#-------------------------------------------------------------------------
# insert a value into the given tree
# assumes root is not null
#-------------------------------------------------------------------------
void insert(root: node, val: int) {
  if root == null {
    return null
  }
  if val <= root.value {
    if root.left == null {
      root.left = make_tree(val)
    }
    else {
      insert(root.left, val)
    }
  }
  else {
    if root.right == null {
      root.right = make_tree(val)
    }
    else {
      insert(root.right, val)
    }
  }
}


#-------------------------------------------------------------------------
# print out the tree in sorted order (in-order traversal)
#-------------------------------------------------------------------------
void print_tree(root: node) {
  if root != null {
    print_tree(root.left)
    print(root.value)
    print(" ")
    print_tree(root.right)
  }
}


#-------------------------------------------------------------------------
# get the height of the tree
#-------------------------------------------------------------------------
int height(root: node) {
  if root == null {
    return 0
  }
  else {
    var left_height: int = height(root.left)
    var right_height: int = height(root.right)
```

```
    if left_height >= right_height {
      return 1 + left_height
    }
    else {
      return 1 + right_height
    }
  }
}


#-------------------------------------------------------------------------
# test the implementation by building and printing a simple tree
#-------------------------------------------------------------------------
void main() {
  # create a tree and print it
  # should print ...
  #   Tree Values: 1 2 5 7 10 12 13 14 15
  #   Tree Height: 5

  var tree: node = make_tree(10)

  insert(tree, 5)
  insert(tree, 15)
  insert(tree, 2)
  insert(tree, 12)
  insert(tree, 7)
  insert(tree, 1)
  insert(tree, 13)
  insert(tree, 14)
  print("Tree Values: ")
  print_tree(tree)
  print("\n")
  print("Tree Height: ")
  print(height(tree))
  print("\n")
}
```

Example 4: Tic-Tac-Toe Game. Another classic example program. This implementation (which again, is not built for efficiency or to demonstrate good coding practie) is an example of using structs and arrays together. In this case, a struct that contains an array is used to represent the state of the tic-tac-toe board. If you like games, here are three challenges for you: (1) revise the game for misére play (i.e., misére tic-tac-toe); (2) extend the game to support an $n \times n$ board, where n is given by the user; and (3) write an extended program for playing ultimate tic-tac-toe.

```
#=========================================================================
# Basic Tic-Tac-Toe implementation
#=========================================================================


#-------------------------------------------------------------------------
# Representation of a tic-tac-toe board and helper functions
#-------------------------------------------------------------------------
```

```
struct board {
  cells: [string]
}

string cell(b: board, c: int) {
  return b.cells[c]
}

bool has_val(b: board, c: int, p: string) {
  return cell(b, c) == p
}

string set(b: board, c: int, p: string) {
  b.cells[c] = p
}

bool valid(b: board, c: int) {
  return (c >= 0) and (c < size(b.cells))
}

bool empty(b: board, c: int) {
  return b.cells[c] == " "
}


#-------------------------------------------------------------------------
# Win, lose, and draw
#-------------------------------------------------------------------------

bool all_three(b: board, c1: int, c2: int, c3: int, p: string) {
  return has_val(b, c1, p) and has_val(b, c2, p) and has_val(b, c3, p)
}

bool win(b: board, p: string) {
  # check rows and columns
  for i from 0 to 2 {
    if all_three(b, i * 3, (i * 3) + 1, (i * 3) + 2, p) {
      return true
    }
    else if all_three(b, i, i + 3, i + 6, p)  {
      return true
    }
  }
  # check diagonals
  if all_three(b, 0, 4, 8, p) {
    return true
  }
  else if all_three(b, 2, 4, 6, p) {
    return true
  }
  return false
}

bool draw(b: board) {
```

```
  for i from 0 to 8 {
    if empty(b, i) {
      return false
    }
  }
  return true
}


#-------------------------------------------------------------------------
# Pretty print a given board
#-------------------------------------------------------------------------

bool display(b: board) {
  var d = new board(new string[9])
  for i from 0 to 8 {
    set(d, i, cell(b, i))
    if empty(b, i) {
      set(d, i, str_val(i))
    }
  }
  println("   " + cell(d, 0) + " | " + cell(d, 1) + " | " + cell(d, 2) + "
      ")
  println("  ---+---+---")
  println("   " + cell(d, 3) + " | " + cell(d, 4) + " | " + cell(d, 5) + "
      ")
  println("  ---+---+---")
  println("   " + cell(d, 6) + " | " + cell(d, 7) + " | " + cell(d, 8) + "
      ")
}


#-------------------------------------------------------------------------
# The driver program
#-------------------------------------------------------------------------

void main() {
  # create an initialize a board
  var b: board = new board(new string[9])
  for i from 0 to 8 {
    set(b, i, " ")
  }

  # player X starts
  var p: string = "X"

  # keep playing until win, lose, or draw
  while (not win(b, "X")) and (not win(b, "Y")) and (not draw(b)) {
    # display current game and prompt for move
    display(b)
    print("Please select a cell for player " + p + ": ")
    var c = int_val(readln())

    # check if invalid valid move
    if (not valid(b, c)) or (not empty(b, c)) {
```

```
      println("Invalid choice, please try again")
    }

    # valid move
    else {
      set(b, c, p)
      # switch players
      if p == "X" {
        p = "Y"
      }
      else {
        p = "X"
      }
    }
  }

  # display the final board state and game status
  display(b)
  if win(b, "X") {
    println("X wins!")
  }
  else if win(b, "Y") {
    println("Y wins!")
  }
  else {
    println("Draw!")
  }
}
```