Today

• Course Overview
• MyPL Intro

Assignments

• Exercise 1 (due next Tues)
Course Overview

Deep dive into programming language (PL) design & implementation

- implement a “made up” typed, procedural programming language (MyPL)
- explore functional programming (using Haskell)

General course goals

- More programming experience (using ideas/techniques you’ve already learned)
- Better understanding of how compilers/interpreters work
- Better understanding of language design (syntax, types, constructs, trade-offs)
- Exposure to different programming “paradigms” (procedural vs functional)

Why study language implementation (“compilers”) ...

1. Essential part of computer science (and most computer-science curriculums)
2. Complicated engineering problems (example of how to build larger systems)
3. Techniques useful for a wide range of software development problems
4. Better understanding of how languages work (can improve your programming)

Why study functional programming?

1. Functional constructs are gaining popularity in most (non-FP) languages
2. New ways to think about programming, new tools for problem solving
Syllabus Overview

Communication:

- Website: cs.gonzaga.edu/bowers/courses/cpsc326
- Piazza (via invite)
- GitHub classroom (requires GitHub account)
- Blackboard (for posting grades)

Getting Help:

- Piazza: Preferred for (basic) questions, clarifications, etc.
- Office hours: Tu/Th 2–3; 11-12 & 1–2 Wed; or by appt

Grading: out of a total of 1,000 points

- Programming Assignments: 500 points (10 at 50 points each)
- Quizzes: 100 points (10 at 10 points each)
- Midterm Exams: 100 points (2 at 50 points each)
- Final Exam: 100 points
- Final Project: 125 points
- Participation: 75 points (30 class meetings at 2.5 points per meeting)

Software: More later ...

- Dev tools: Java (jdk-13 or higher), bazel, ghc (v8 or higher)
- Either: Remote development on ada server (e.g, vs code) ... “supported"
- Or: Setup tools on your own machine ... not “supported"
Expectations:

- Stay engaged and participate in class
- Do your own work
- Ask when you have questions
- Start assignments early and give yourself enough time to succeed
- Assume you have everything you need already (and ask when in doubt)

(Some) collaboration is encouraged ...

- can discuss and talk about assignments
- can work together (as long as you are doing your own work)
- plagiarism not allowed: no splitting up work, no code sharing, etc.

Things to avoid:

- Falling behind (assignments build on each other)
- Not studying the material (e.g., for quizzes)
- Not reading and following instructions
- Using YouTube, Google, StackOverflow, and other sites to "learn" the material
- Not coming to office hours and not using (or checking) piazza
Homework late and resubmission policy

- 10 point late penalty (20%) within one week after due date
- 15 point late penalty (30%) after one week past due date
- if score < 40 points (80%), can fix and resubmit (for max score of 40)
- late work cannot be resubmitted
- must notify me when work submitted / resubmitted (via form)

Online access to lectures (see syllabus for details)

- Only for those that have a COVID-19 illness or related issue
- Must notify me prior to lecture (I’ll need some lead time)
- Must send me your notes from lecture for participation credit

Homework warning:

- many of the assignments are longer than in most classes
- most can’t be finished in a single sitting

Note on “exercises”:

- a few take home exercises to help (mostly) with non-programming content
- will count towards “participation” grade
Brief intro to MyPL v5

Basics:

- Simple PL for exploring design and implementation ideas
- Mixes explicit and implicit typing (for type checking and type inference)
- Includes typical primitive types as well as “record” types (like simple structs)
- Functions use pass-by-value (i.e., copies), objects passed as “reference” values

Comments:

```
# this is a single line comment
# only single-line comments are supported
```

Primitive Data Types:

```plaintext
int       # 4 bytes
double    # 8 bytes double precision
bool      # either true or false (not 0, 1)
char      # single character
string    # sequence of characters
void      # for procedures
```

Values:

```
0, 1, 7, 10, 20, 876132  # int values
1.0, 1.01, 10.3, 0.5, etc  # double values
true, false               # bool values
'a', 'b', '1'             # char values
"foo", "bar", ""          # string values
nil                        # similar to null
```
Variable declarations

• all declarations must have initializers

• variable types are optional in most cases

```solidity
# type inferred from initializer
var x1 = 5
var x2 = 5 * 3 + 2
var z = 3.14159
var flag = true
var u = x1 + x2
var s = "Hello"
```

```solidity
# type required when var initialized to nil (all types)
var string str = nil
var int x = nil
```

```solidity
# type can be optionally provided (still type checked)
var int x = 42
var int y = x
var bool c1 = false
```

Variable assignments

```solidity
# assumes all variables declared and types match
x = 10
z = false
u = 3.1 * 4.2 + v
w = nil
```

Notes on nil:

• for initialization, assignment, and comparison (!=, ==)

• other uses result in a runtime error (e.g., 5 + nil or x < nil)
For loops

• limited version for iterating over integers

```plaintext
var x = 0

for i from 1 upto 5 {       # i is declared locally
  x = x + i
}
var y = x                   # y is 15, i undefined

for j from 5 downto 1 {
  y = y - j
}
var z = y                   # z is 0, j undefined
```

Relational comparators and Boolean operations

• normal comparators: >, <, <=, >=, !, == ... for most primitive types

• normal Boolean operators: and, or, not

• expressions can be parenthesized, e.g., (x <= y) or ((x == 0) and (y > 1))

While loops

• normal version of while

```plaintext
while x > 1 and x < 10 {       # note parens optional
  ...
}
```
Conditional Statements

- normal version of if-elseif-else statements

```python
if x == 1 or x == 2 {
    ...
}
elif y > 20 and y <= 30 {
    ...
}
elif y > 30 {
    ...
}
else {
    ...
}
```

Arithmetic operations

- typical operations for ints and doubles
- overloaded version of + for strings, chars
- can be parenthesized and chained together (e.g., `x + (y * z)`)

```plaintext
x + y # int/double addition as well as string concat, append, prepend
x - y # int/double subtraction
x * y # int/double multiplication
x / y # int/double division
x % y # int mod
neg x # int/double sign flip (- reserved for sub)
```
Functions

- all functions have explicit return and parameter types
- recursion is allowed

```plaintext
# function that takes two ints and returns an int
fun int f(int x, int y) {
    var z = x + y
    if x < y {
        z = neg z
    }
    return z
}

# function that takes an int, no return value
# print is special since it takes a value of any primitive type
fun void g(int x) {
    print(x)
    print("\n")
}

# nth fibonacci number
fun int fib(int n) {
    if n < 0 {
        return nil
    }
    if n == 0 or n == 1 {
        return n
    }
    return fib(n-1) + fib(n-2)
}

# examples of function calls
var x = f(1, 2)  # call f
g(x)  # call g
fib(10)  # call fib
```
Structured (aka record) types

- similar to `struct` in C++, or classes without functions
- essentially just a set of variables
- dynamically allocated and deallocated (to/from the heap)

```javascript
# linked list node with int value and next reference
type Node {
    var val = 0          # implicitly an int
    var Node next = nil  # initially nil
}

var n1 = new Node      # create a new Node object on the heap
n1.val = 10             # set n1's value
var n2 = new Node       # create a new Node object on the heap
n2.val = 20             # set n2's value
n1.next = n2            # link n2 to n1
n2.next = new Node      # create a new node linked to n2
n2.next.val = 30        # set the new node's value
n2.next.next = nil      # terminate the list

var eq1 = n1 == n2      # false, n1 & n2 different references
var n4 = n2.next.next   # n4 is nil
var eq2 = n4 == nil     # true

# runtime error:
# var v = n4.val

delete n2.next
delete n2
delete n1
delete nil          # not an error
```
MyPL does not perform automatic type conversion

- for example, the following result in type errors:

```plaintext
var int x = "4"      # type error (expected int, found string)
var bool y = 1       # type error (expected bool, found int)
var double z = 3     # type error (expected double, found int)
```

MyPL provides built-in functions for type conversion

```plaintext
# string to int and double (can lead to runtime error)
var int x1 = stoi("4")  # string to int
var double x2 = stod("3.14")  # string to double

# int to string and double
var string y1 = itos(4)  # int to string
var double y2 = itod(3)  # int to double

# double to string and int
var string z1 = dtos(3.14)  # double to string
var int z2 = dtoi(3.14)   # double to int (truncates)
```

Additional MyPL built-in functions

```plaintext
var string s = read()  # read string from standard input
var int x = length("foo")  # number of string chars
var char c = get(0, "foo")  # get i-th string char
```

String append and concatenate

```plaintext
var s1 = "Zags" + '!'    # s1 is assigned "Zags"
var s2 = 'Z' + "ags!"   # s2 is assigned "Zags!"
var s3 = "Za" + "gs!"   # s3 is assigned "Zags!"
```
MyPL Programs

- only consider single-file programs
- normal scoping rules (more later)
- must have a **main** procedure

```plaintext
# user defined type
type Car {
  var string make = nil
  var string model = nil
  var int year = nil
}

# create a car object (not required, but like a constructor)
fun Car make_car(string make, string model, int year) {
  var c = new Car # creates a new car object (on the heap)
  c.make = make # set the car's make
  c.model = model # set the car's model
  c.year = year # set the car's year
  return c # return a reference to the new car
}

# pretty print a car object
fun void print_car(Car c) {
  print(c.make + " " + c.model + " " + itos(c.year) + "\n")
}

# destroy a car object (not required, but like a destructor)
fun void destroy_car(Car c) {
  delete c
}

fun void main() {
  var c1 = make_car("Toyota", "Corolla", 2021)
  var c2 = make_car("Honda", "Civic", 2020)
  print_car(c1)
  print_car(c2)
  destroy_car(c1)
  destroy_car(c2)
}
```