Overview

Lecture 1 provided a general overview of CPSC 326. We also started discussing the basics of MyPL. We will continue to go over MyPL (focusing on syntax) on Friday. Before the next lecture, you should do the following.

1. *Log in to Piazza.* You should have received an email from piazza with instructions for logging in and viewing content. If you didn't receive the email (which should also contain a passcode for accessing the class discussion site), please email the instructor.
2. *Carefully read the syllabus.* If you have questions regarding the syllabus please reach out to the instructor (email, office hours, or on piazza).
3. *Scan the project requirements.* The project description is available from the course webpage. Be sure to read it over to get a general sense of what is being asked of you and to mark down due dates. You won't need to work on the project prior to Exam 1. However, it is good to start thinking about the project and what you might like to explore. The extra credit project is also described in the project requirements.
4. *Review terminology.* A few terms were used informally in the lecture. These are described in more detail below. We will cover many of these terms and concepts further as the semester progresses. A reference to a classic textbook on programming language concepts is also provided below.
5. *Set up your programming environment.* You should get started setting up your programming environment for the course. See below for details.

Terminology & Additional References

Note: Many terms used to describe programming languages are informally defined, sometimes a bit vague, and often used in different ways. The terminology can be confusing when looking online or in other resources since terms sometimes have multiple, hard to pin down meanings.

Model of Computation. A model of computation is a formal system that defines how computations are carried out. Many models of computation have been defined. Models of computation are generally independent of any particular implementation, either via a physical machine or by a programming language. Some examples of models of computation are: the Turing Machine model, the Lambda Calculus model (functions), the Logic Programming semantics, the RAM Machine model (registers), the Process Calculi models (for representing concurrency), and so on.

Programming Language Paradigms. A programming paradigm is an informal phrase that refers to the underlying patterns, concepts, and techniques that define or characterize programming languages. Sometimes a programming paradigm is used to also mean a model

of computation, a combination of models, or a "family" of languages. Some of the main paradigms are the imperative (typically procedural), functional, logic, and object-oriented programming paradigms. Each of these is also sometimes referred to as a distinct *family* of languages. Languages that combine multiple paradigms are considered *multiparadigm*. Most modern programming languages (including Python, C++, and Java) are multiparadigm in that they combine imperative/procedural, object-oriented, and functional approaches.

Parameter Passing. In languages with functions (procedures), a variety of mechanisms exist for passing items (values, objects, etc.) to a function as part of a function call. The input "positions" of a function are referred to as the *formal parameters* (or *formal arguments*). When the function is called (invoked), the items passed to the formal parameters are referred to as the *actual parameters* (or *actual arguments*). In the *pass-by-value* convention, the value of each actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the function. Other conventions include pass-by-name, pass-by-reference, and pass-by-copy. In class, we'll often refer to formal parameters as just parameters, and actual parameters as just arguments.

Object Identifiers. In many languages, the instantiation of a class (or struct) is called an *object*, and each such object is assigned an *object identifier* (*oid* for short) at creation time. The object identifier is used to refer to (i.e., to reference) the object. When passing an object as an argument to a function under pass-by-value, the object's identifier is passed as the corresponding object value. The use of the object identifier in this way is similar to pass-by-reference, however, it is technically just using the pass-by-value mechanism. This difference can be confusing because the object identifier serves as a reference to the object, but in this case we are passing the object identifier (reference) as the object's value (using pass-by-value). We will use this pass-by-value approach in our MyPL implementation.

Classifying Programming Languages. There are various terms that are used to help differentiate (i.e., classify) programming languages based on their properties and/or features. We'll discuss many of these throughout the semester. A few were briefly mentioned in Lecture 1, which are described further below.

Statically Typed. The term *static* here means before runtime (e.g., at compile time), whereas *dynamic* means during runtime (i.e., as the program is executing). In a language that is statically typed, errors relating to types in a program can be detected before the program is executed. (Note that this is why statically typed languages are beneficial—as programmers, we want to find bugs as quickly and painlessly as possible!) There are various ways to design languages so that they can be statically typed. Many languages achieve static typing by enforcing strict typing rules (e.g., in expressions) and requiring explicit type declarations (sometimes referred to as *type annotations*). Languages differ in terms of what typing errors can be found statically, and typically, not all typing errors are discoverable before runtime (i.e., some runtime checking is still needed). Examples of languages that are mostly (but not fully) statically typed include Java and C++. Many functional languages (e.g., Haskell and OCaml) are unique in that they are statically typed but do not require

explicit typing. Instead, these languages use sophisticated type inference techniques (i.e., analyzing the code to determine types) to find type errors statically. Java, C++, and newer languages are increasingly adding type inference to relax explicit type declarations (e.g., through Java's `var` and C++'s `auto` keywords). A difference between Haskell and these approaches, however, is that function signatures (both input and output types) are inferred without type annotations.

Dynamic Typing. In a language that employs dynamic typing, type errors are detected during runtime. This approach still retains *type safety*, i.e., errors due to type issues are still (eventually) detected. Many traditional "scripting" languages, like Python and JavaScript, largely only employ dynamic typing and do not require type annotations. (Note that Python is actually a relatively "older" language in that it was first introduced in 1991, whereas, Java and Javascript were first released in 1995.) While each value "knows its type", variables can hold different types of values as a program runs. Dynamic typing allows greater flexibility and reuse, however, it can lead to difficult to find bugs that only occur in specific cases (execution paths within the code). Another disadvantage of dynamically typed languages is the performance penalty that is introduced by having to maintain the type of each variable and to continually check compatibility of types in each expression that is executed at runtime.

Additional Information. The following textbook goes into considerably more detail regarding terminology and various aspects of programming language paradigms and techniques. This is an older edition of the book with copies available online. We will go over many of the ideas presented in the book.

Robert W. Sebesta. *Concepts of Programming Languages*, Pearson, 10th Edition, 2012.

Chapter 1 provides a good overview of why it is important to study programming languages and goes into more depth concerning various properties and classifications of languages.

Programming Environment

To implement MyPL this semester, we will be using Java as our programming language and Maven as our build system. To do the homework assignments on your own machine, you must download and install (if you haven't already) the following.

- Java JDK 21 (or higher). You can use either the OpenJDK distribution or the Oracle JDK SE distribution. For OpenJDK (preferred) see https://openjdk.org/. For the Oracle JDK (also fine) see https://www.oracle.com/java/technologies/downloads/.
- Apache Maven 3.8 (or higher). To download see https://maven.apache.org/.

Note that you will need to be able to run `java`, `javac`, and `mvn` from the terminal (command prompt) on your machine. This means you may need to set the path accordingly to access these programs. If you are unsure how to do this or have trouble installing software on your own machine, please get started early. Note that you can use any IDE you would like for the course, however, again, you will need to ensure it is set up appropriately prior to the first homework assignment.