Lecture #14: Programming Languages and Programming on the Web

CS106E Spring 2018, Young

In this lecture, we explore why there are so many programming languages and how programming languages differ. As we discover there are a number of different programming paradigms that languages are based on. We take a look at some of these including imperative, procedural, object-oriented, logic, functional, and pure-functional programming.

There are other dimensions in which languages differ. One important one is type systems. We contrast languages that support static type checking with those who use untyped variables, parameters, and function return types. We review the difference between compiled languages and interpreted languages (originally covered in the L04-L05 CPU lectures) and managed (e.g. garbage collected) vs. unmanaged languages (covered in the L06 Memory lecture). We end our discussion by looking at the characteristics of scripting languages and see how this matches up with JavaScript, a scripting language built into all web browsers.

We end the lecture with a discussion of how programming works on the web. We see that there are two distinct programming models – client-side programming and server-side programming.

Why are there so many Programming Languages?

- The [Wikipedia Page on Programming Languages] lists over 700.
- There are a variety of reasons why we create new languages, here are some of the more important:
    - Different languages serve different purposes. Compare a scripting language to a language designed to write device drivers. Our scripting language is designed to allow an end-user[1] to write scripts inside of an application (such as someone adding new Visual Basic Scripts to automate email tasks in Microsoft Outlook). Our hardware driver language needs to give direct access to the system hardware. These languages serve very different needs and naturally have very different characteristics.

    - As computer scientists continue to explore computing, we come up with new programming concepts. You may be surprised to find that concepts such as functions and while loops did not exist in the earliest programming languages. While new concepts are often grafted on to existing older languages, in some cases, the revised language just isn't that clean. Coming up with a new language, designed from the ground up to incorporate these concepts, may be a better approach.

[1] The term *end-user* refers to the actual person who uses the product. So for example, the end-user for Microsoft Word is the person using it to write papers.

- Where do these new concepts come from? Languages are sometimes created to explore specific concepts. Here are a few examples:

    - The Self language was designed to explore a different concept of objects based on the concepts of prototypes. While Self isn't widely used, JavaScript used Self's prototyping concepts as the basis for its object model.

    - The APL language was designed to explore what a language designed to manipulate matrices might look like.

Programming Paradigms
We can distinguish programming languages in a variety of ways. One of the most important is which programming paradigm (or programming paradigms, as many languages support more than one paradigm) is the language based on. Let's take a look at some of the most important paradigms:

Imperative Programming – In imperative programming, we provide a step-by-step sequence of statements for the computer to carry out, and provide the specific order of operations.

Procedural Programming – This adds to Imperative Programming by allowing the programmer to define procedures that can be called (as mentioned above, the earliest programming languages did not have the concept of functions or procedures). The vast majority of programming done today fits into both the Imperative and Procedural categories.

Object-Oriented Programming – In object oriented programming we allow organization of data into classes of objects and then define methods that operate on those objects. Some language scholars distinguish between *Object-Based Languages* that allow the creation of classes and objects and *Object-Oriented Languages* that allow us to create a hierarchy of classes, with subclasses inheriting properties and methods from their parent classes.

Logic Programming – While this paradigm is not currently in widespread use, it provides a nice contrast with the Imperative and Procedural paradigms. In logic programming, instead of telling the computer what to do step-by-step, we provide a set of rules or relationships. For example:

- x is the grandparent of z if x is the parent of y, and y is the parent of z.
- x is an ancestor of y if x is the parent of y.
- x is the ancestor of z if y is the parent of z and x is an ancestor of y.

We then provide a set of facts, for example:

- Mary is the parent of Alice.
- John is the parent of Mary.
- Elizabeth is the parent of John.

We can now ask our computer questions on the basis of our logic programming such as:

- Is Mary the grandparent of John?
- Is Elizabeth an ancestor of Alice?

While not in widespread usage today, at one point, Logic Programming was thought the most likely candidate for achieving advanced artificial intelligence. In 1982, the Japanese Government made logic programming the centerpiece of a $400 million dollar effort called "Fifth Generation Computer Systems (FGCS)," which was designed to leapfrog Japan ahead of its competitors.[2] The project was largely a failure, something that I point out to students asking about China's new government-led Artificial Intelligence effort. China may well succeed, however, as Japan discovered, throwing money at the problem does not guarantee success.

Functional Programming – A language that programmers refer to as a "Functional Language" may actually refer to one of several different issues.

- The ability to create and refer to functions just as one can refer to other objects is quite useful. When a language has this capability, we say ***functions are first-class objects*** in the language (meaning that functions have every bit the same abilities as other objects, for example, they can be created and stored in variables and passed to other functions as parameters).

  Let's see why this ability is useful. Suppose I write code that sorts a list of objects. One problem with sorting is that in order to sort, I need a way of comparing objects – for example, if I have a list of students, am I sorting them based on their year in school, their last name, or their first name? If my language supports functions as first class objects, my sorting procedure can be written to take a sorting function as a parameter. If I want to sort by first name, I pass in a function that compares two objects on the basis of their first name. If I want to sort by year in school, the function I pass in compares students on how long they've been in school.

  Many languages fit into this category of functional programming.

- Some functional languages go further and allow the program to define new functions or modify existing functions on the fly (meaning while the program is actually already running). This quality is one aspect of meta-programming, which we'll address further in another section of today's class notes.

- One quality that a functional language may have is to have its functions act as mathematical functions, disallowing side effects. Such a language is sometimes called a ***pure functional language***.

  Just as mathematical functions such as sin or cosine always returns the same results, no matter how many times they are called, functions in a pure functional language always return the same results. Most computer languages don't actually work this way. Instead, most programming languages allow a function to have side effects, such as changing global variables, or modifying the state of the world (e.g., adding data to a database). When called a second time, functions that allow side effects are liable to return different results because they modified the internal state of the program the previous time they were called and this affects the results they return.

[2] I haven't been able to determine if that $400 million number is in 1992 dollars (when the project ended) or in 1982 dollars (when the project started). The total cost of the project was somewhere between $700 million to $1 billion dollars in 2018 dollars.

Pure functional language do have some interesting aspects. Because a function always returns the same results, the order in which functions are called often doesn't matter.[3] This facilitates allowing us to run different functions in parallel.

Procedural vs. Declarative Languages
One interesting distinction that can be made between languages is dividing them into languages that are procedural and those that are declarative. In a procedural language, we specify the steps that are taken to carry out the task. Most programming languages are procedural (this category includes imperative programming, procedural programming, object-oriented programming, and most functional programming). In contrast, some languages don't specify how a task should be carried out. The logic programming described in the previous section is an example of a declarative language.

Depending on where one wants to draw the line on what exactly counts as a programming language, SQL could be considered a declarative programming language. One might even make a case for languages such as HTML to be considered as declarative languages.

Static vs. Dynamic Typing

- Some languages support ***Static Type Checking***. In these languages, the programmer is required to explicitly provide typing information in their programs. When I declare a variable or parameter, I specify what type of data will be used and when I write a function, I say what type of information will be returned. For example, I might create a variable to store someone's age, and I would declare that that variable would store the information as an 8-bit unsigned integer (allowing me to store ages from 0-255).

  The compiler can take the information on variables, parameters, and function return types that has been provided by the programmer and use it to make sure that the program will work correctly. If a function is called with parameters which do not match those expected, the compiler will notify us before the program can be run.

  Languages with Static Type Checking are particularly useful on larger projects. The explicit type information provided on variables, parameters, and function return types acts as a form of documentation, which is both required and also guaranteed to be kept up to date. The ability of the compiler to perform type checks is particularly useful when my code calls code someone else has written, as I may not be entirely familiar with what types that code expects me to provide.

- If a language does not require the programmer to explicitly provide type information on variables, parameters, or function return types, we say that these are ***untyped***. For example, in a language in which variables are untyped, my variable named "age" might be assigned an integer value like 18, it might have a decimal value like 18.3 assigned to it, or it might even have a string value like "eighteen" assigned to it. My program does not specify what type of information should be stored in the variable.

  In a language where variables, parameters, and return values are untyped, type checking occurs at runtime. These languages are said to use ***Dynamic Typing***. You may also hear the term ***Duck Typing***[4] used in conjunction with these languages.

[3] Order won't matter unless the output of one function is actually used directly as the input to another function.
[4] The term duck typing comes from the phrase "if it looks like a duck and quacks like a duck, it's a duck." The idea here being if I am storing some kind of an object in a variable, as long as it does what my code asks it to do, that's good enough. So if I try to add with it, and it supports adding (perhaps because it's an integer, a decimal number, or even an imaginary number) then it's fine.

In general, languages that do not require typing of variables, parameters, or function return values are less verbose and potentially more flexible. However, as previously mentioned static type checking is particularly beneficial for programming done with teams (as opposed to small programs written by just an individual programmer), and because of this, there are some languages which are essentially variants of an untyped language which have had types inserted – for example, TypeScript is JavaScript with types added.

Compiled vs. Interpreted (vs. Hybrid)
We previously discussed Compiled vs. Interpreted languages in the lecture on CPUs (as well as hybrid approaches such as Java's compilation to Bytecode). This is a major distinction that occurs with languages. Please refer to the "L04-05 Hardware (CPU)" lecture notes for more on this issue.

One thing worth mentioning here is that statically typed languages are generally compiled. The analysis during the compilation process is when the static types are used to correct programming errors. Languages with untyped variables/parameters/return values tend to be interpreted, although one can certainly come up with a compiled version of these languages.

Managed vs. Unmanaged
In the lecture on computer memory, we discussed the difference between languages that require the programmer to manage their own memory vs. languages that manage memory for the programmer using a garbage collector. This is also a major distinction between different languages. See the handout "L06 Hardware (Memory)" to review this issue.

Meta-Programming
Some languages allow us to access and modify features of a program that are typically thought of as static. This ability of a program to modify its own behavior is referred to as Meta-Programming.

- For example, using meta-programming features of a language, we might be able to access information about an object's class. In fact, in some cases, the class might itself be considered an object that we could manipulate and modify on the fly.

- As I mentioned earlier, in some languages, we can define new functions on the fly, and programs written in the language can themselves be treated and manipulated as data. In LISP for example, functions are simply lists that happen to have nodes that correspond to control constructs in the language. I can assemble new lists, and if I construct them of the correct components, I can then execute those lists, exactly the same as I can execute the original program.

Case Study: A Scripting Language for the Web
As I mentioned at the start of this handout, scripting languages are languages that are designed to run inside of another application, allowing an advanced user to automate a task. A scripting language might allow the end-user to write a short program to automate movement of mail messages to different folders or a program that combines a spreadsheet of names and addresses with a word processing document to generate custom variants of a letter.

Let's consider what sort of properties we might want in general for a scripting language.

- In general, as we've seen most programming languages follow the imperative/procedural paradigm, so our scripting language is likely to use these paradigms. Although providing an end-user with a declarative language might actually simplify their programming task considerably (writing SQL queries or HTML webpages is a much simpler task then writing procedures in Visual Basic Script), declarative languages are likely to be less flexible.

- If we do choose to follow the imperative/procedural paradigm, we will probably include object-oriented abilities. We might also want to include some functional capabilities, such as treating functions as first class objects. Both of these are very common in modern languages (in contrast, pure functional languages tend to occur much less frequently).

- We might decide that scripts will tend to be small and we aren't particularly concerned with efficiency because of this. Therefore, our language need not provide a wide range of types – we won't need the ability to distinguish between 8-bit integers, 32-bit integers, 64-bit integers, different sizes of floats, and signed vs. unsigned numbers. We'll simply provide a generic "number" data type.

- We want our users to be able to write compact code quickly so we choose to not require type information on our variables, parameters, and return types.

- Finally, we want our user's code to run regardless of whether the application is on an Intel processor, PowerPC processor, or some other type of CPU, so we choose to make our language interpreted. This also provides our programmers with a simpler development cycle, since they don't need to go through the extra step of compiling their code before running it.

We've just described the JavaScript language, which is the scripting language built into all web browsers.[5]

Programming on the Web

There are two distinct models for how programming occurs in conjunction with websites. The difference between the two models is which computer the program actually runs on.

In ***Client-Side Programming*** the program files are sent to a visitor's web browser, just as the HTML and CSS files are sent to the browser. When the program executes, it executes on the website visitor's computer. As you may recall, we refer to the computer making requests of the web server as the client computer and may also refer to the web browser itself as a web client. Thus, this type of programming is called client-side programming.

In ***Server-Side Programming*** the program remains on the web server, where it executes. The results from the program are sent through the Internet, typically as HTML files, although server-side programming can be used to generate any type of files including creating new JPEG or PNG files.

Advantages and Disadvantages

Client-Side Programming

- Gives near instantaneous response — In contrast with Server-Side Programming, which requires inputs to be sent through the Internet to the servers and the results to be sent back through the Internet from the server, with Client-Side Programming once the webpage has been received by the web browser, no further communication with the server is needed.

[5] In fact, some of the assumptions made above turned out to not be correct. In particular, JavaScript was designed for writing small programs and was not originally intended for large programs written by many programmers working together. As I mentioned earlier, the untyped aspects have caused problems and as a result, several different companies have developed typed versions of JavaScript. JavaScript also lacked important features needed for larger programs such as some sort of module or package mechanism. These have finally been added ~20 years after JavaScript was first introduced.

- Offloads work to clients' computers — Because the actual program is running on the client's computer, client-side programming does not add to our web server load. This keeps costs down.
- Allows direct user interactions with webpages — Server-side programming is based on submitting form information to a web server. In contrast, client-side processing is more flexible, and has full access to everything displayed on the webpage. Client-Side processing can be used for things like allowing the user to drag images around the webpage or to display popup items as the user moves the mouse around the webpage.

Server-Side Programming

- Gives server access to information — With server-side programming the user sends information to the webserver, where it can be processed and stored. Any application that requires information to be stored on the server must include server-side programming. For example, a web storefront needs order information to be sent to the server, where we can store it in a database. Similar a bulletin board system requires messages to be sent and saved on the server.
- Improved security — Client-side processing requires us to send all information involved with the program to the client computer. If we want to maintain security, we may want to keep data on the server.
- Better for large databases —If we have a lot of data involved, we won't want to send it all to the client. Instead, we'll maintain a large database on the server, process the information there and only send results to the client.

- In general, if an application can be done on the client, it should be done on the client. Your user will have a much smoother experience, and your application will need fewer computing resources (and thus be cheaper to run) than the same application done server-side.
- However, there are many, many applications that require server-side programming. As previously noted, a web storefront is an example of an application that simply cannot be done strictly client-side.

- It is possible to combine client-side and server-side programming. For example, if a user is submitting an order, before the order is submitted, I can use client-side programming to check and make sure that the user has filled in all their information and that items such as credit card numbers and email addresses are in the correct format. Only if the client-side program has verified that the information inputted looks correct, do we allow the webpage to submit information to the server.