Main topics of the week:
- Intro to Programming Language Concepts
- Programming Language Domains, Paradigms
- Compilation versus Interpretation
- Formal Definition of Grammar
- BNF Notation
- Languages and Grammars
- Parse Trees

Motivation for Studying Programming Languages

This course will introduce you to the fundamental ideas used in the design of programming languages. It is not a course about how to program in a particular language and we assume you have some experience programming in an imperative language like C, C++, or Java. This is not a course in how compilers work and how to design a compiler. Nor is it a course in the theory of computing. There are separate courses in Compilers and Automata Theory that cover those subjects in detail. We will see some things like grammars and memory management that certainly overlap with those other courses, but the focus here is on the general aspects of programming languages. We will introduce you to some new programming paradigms and languages, but you will not get a complete treatment of these languages. Rather, you will develop an appreciation for the differences between languages and increase your ability to teach yourself about new languages. And, you will understand the tradeoffs between characteristics of programming languages so that you can make better choices about what language is most appropriate for a given application, and what are the important issues in designing a new language.

One of the first questions to ask, is what drives the creation of a programming language in the first place? A computer is just a machine that is capable of executing instructions. A “program” is just the set of instructions. So we can take as a definition:

A **programming language** is a notational system for describing computation in machine readable and human readable form.

At the machine level, these instructions are just code – they constitute a “language” that the machine can understand, e.g., fetch a value from a memory location, add one value to another, store a value. However, if you have looked at any machine microcode or even assembly language, you know it is not easy for most people to understand. After all, it reflects the low level at which the machine operates, and when humans design solutions for problems, these solutions are typically thought of at a “higher” level. So long ago, machine code gave way to higher level languages. The first level up from the machine microcode is an assembly language. Assembly language maps directly to the machine code, but is more symbolic, rather than just the binary strings that the machine really understands. Programming assembly language is certainly an improvement over programming microcode, but still forces the programmer to deal with the machine’s model of computation. During the 1960’s, Fortran (Formula Translator) was created and became popular. The successful characteristic of Fortran was that it was a programming language that looked a lot like the symbolic algebraic formulas that were being used as the notation in describing the problems people were trying to solve with computers. At the time, there was great doubt about whether this was a good thing to do, since it was...
considered doubtful that a translator could produce reasonable code. And of course, as it turns out, the translators were able to produce reasonable code, enough so that the advantage of programming in a language closer to the solution domain far outweighed any inefficiency of the translated code over hand crafted microcode. Fortran became very successful for numerical applications, and Algol attempted to unify various similar languages for these applications. At the same time, Lisp gave rise to a strain of functional programming languages, a fundamentally different paradigm. With the advent of the minicomputer toward the end of the 1960’s, the C programming language, sharing much of the structure of Fortran but with more general purpose computing in mind, was designed along with the Unix operating system and C became one of the most widely used programming languages. ML merged ideas from both these strains. In the 1980’s, object oriented programming came into vogue and C++ and Java are the main successful languages (as measured by adoption) of that type. You can consult the genealogy in Louden for a more complete picture of the history of the programming languages.

However, what really drives the development of programming languages? The basic goal is just to perform some type of computation. A programming language is created to:

1) Make programming easier for people
2) Ensure more correct, bug free programs
3) Make efficient use of the computer resource

Sometimes these goals are at odds – a language that ensures that only “perfect” programs can be written may not produce very efficient code. Although one could argue that efficiency is the domain of interpreters and compilers, and should be totally transparent to the programmer, the reality is that the design of a language has a lot to do with whether we can implement compilers and interpreters to produce efficient code. Some folks would even argue that the implementation of the language is the driving force for the language design. Likewise, the design of the language has a lot to do with whether we can guarantee the “correctness” of the program. And of course, the design of a language has everything to do with whether the programming task is more convenient, i.e., easier. Finally, the programming language conveys a certain aesthetic about a program – mostly our concern about this is that the language makes a program easier to read and understand.

Since there is basically one fundamental computing model (as expressed at the most basic level in the Turing Machine), any programming language can theoretically be used to perform any programming task. After all, from an execution standpoint, the original language is irrelevant – a compiler, written in some language, just reads another language and produces some set of machine instructions. The machine doesn’t really “care” where those instructions came from – all of this programming language concern is really for us poor humans.

Toward the goal of making programming easier for people (i.e., human readable), programming languages deal with abstractions that keep the low level computing model at a greater distance. **Data abstractions** give us a higher level approach to thinking about the data so that we do not get caught up in the machine details. For example, when we write a program dealing with numbers, we don’t really want to get bogged down in how arithmetic is performed – e.g., two’s complement versus one’s complement, etc. The details are clearly important to the actual computation, and the machine designer must be very concerned with understanding this and getting it right, but we would like to stay...
above this level of detail. That is, we want to deal with the abstraction of a counting number. Moreover, we would like to refer to number constants with our usual mathematical notation, rather than specifying bit patterns, and we would like to refer to number entities symbolically – i.e., we want to use variables, which is another data abstraction. The ability to lay out structures and classes to collect our data together is another form of data abstraction. **Control abstractions** give us a higher level approach to thinking about the actions performed by the machine. Again, the machine designer must be concerned with things like how one value is copied to another location by the right combination of fetch and store instructions, but we would like to view this as the abstraction of assignment. Similarly, the program counter is used to know which instruction is to be executed next, but we want to deal at the abstract level of sequential execution with selection (if-else) and loops. Control can be further structured with the abstractions of procedures and functions or methods.

**Programming Domains**

The domain of the applications that will be written in the language heavily influences the design of a programming language. A general-purpose language is by definition not domain specific, and in principle does not favor any particular type of application. However, designing a language for specific types of applications will ideally result in a language that is more expressive, easier to program, less error prone, and more efficient for the applications for which it is intended. Some applications have special requirements (e.g., extremely high need for fault tolerance as in air traffic control or life support equipment) and the design of the language can facilitate these needs. We could roughly divide application domains into several areas:

- Scientific applications
- Business applications
- Artificial Intelligence
- Systems Programming
- Scripting Languages

Each of these domains has its own characteristics and requirements. For example, calculation expressiveness and efficiency may be more important, or parallel processing may be the highest priority. Scripting languages allow programs to be written very quickly, but with little attention to efficiency.

Other important characteristics to consider in a programming language are to find the right balance between complexity and simplicity, between ease of coding and correctness of the code. For example, we want to be able to express complex control flow, but not to the extent that the language has so many keywords and nuances that it becomes impossible to predict how a program works.

**Evaluating Languages**

We have said that the primary goals of a programming language are to make easier, safer, and more efficient use of the computer. What are the things we should look for to compare one language to another? One way to evaluate a language (relative to the programming domain, of course) is to consider the language’s **readability**, **writability**, and **reliability**. Readability of a language is a measurement of how easy it is to understand programs written in the language. This is important for maintenance and
enhancement of software. Experience in software engineering indicates that the majority of the software life cycle is spent on finding bugs and maintenance rather than original development. So having a language that is readable makes this part of the life cycle easier and less costly. Keeping a language simple tends to make it more readable, but the right balance must be struck between simplicity and expressiveness. Better languages tend to have a number of constructs that can be combined orthogonally. This allows expression of complex ideas without so many different constructs that it is difficult to remember them all. Notation that is too complex or has too many keywords gets in the way of readability. Likewise the organization of the code should be as natural as possible so to promote easy comprehension. Lots of features are nice, but too many, or unexpected hidden behaviors also impair readability. Language design requires a trade off between expressive power and the ability to create unintelligible programs. Designing the language so that the programmer cannot misuse the constructs (e.g., operator overloading) deprives the language of expressive power. The syntax and grammar of a language can facilitate or improve readability.

While readability contributes to easy maintenance of software, writability is a general measure of the ease of original software development. Many of the same factors that contribute to readability also contribute to writability, but in addition, language features that support data abstraction and constructs that give greater expressivity help to make a language writable. Some of the latter may be at odds with readability, e.g., the compact notation possible in expressions in the C language (with side effects, prefix/postfix operators, etc.) can allow a lot to be done with a small amount of code, but may make the code incomprehensible.

Finally, the reliability of a language generally refers to how much analysis can be performed by the compiler or interpreter to ensure that the design of the software is using the language constructs correctly. Type checking is a static analysis that can be performed to ensure types are used correctly, and as we will see later, languages can have various degrees of the strength of type checking, ranging from typeless languages to very strongly typed languages. The language may also include features to promote runtime reliability, such as exception handling that allows cleaner software error recovery, or memory management and garbage collection that avoids misuse of program storage.

**Design Considerations**

In addition to the general criteria above for evaluating languages, there are some other criteria that factor into language design. In the early days of languages, execution efficiency was the most important consideration. With the spectacular increases in processing speed, execution efficiency is not as critical, but it is still important since we keep designing more complex programs that push the computational limits. Efficiency is a more general concern – one aspect is the execution speed, but there are also efficiency aspects to memory use, programming design speed/ease, maintenance ease, compilation speed, etc. For example, static language constructs permit static analysis, which may not only speed up compilation, but may also speed up execution by avoiding the need for certain runtime checks. Lack of automatic garbage collection may slow down the execution of a program, but may speed up its development. So often times we have
conflicts between various efficiency aspects and other language criteria like readability and writability.

**Regularity** is a measure of how well a language integrates its features, so that there are no unusual restrictions, interactions, or behavior. The constructs should be general enough to accommodate most programming needs, but not so general as to make the language unwieldy. Constructs should typically be orthogonal – there should not be unexpected interactions between different constructs. This makes it easier to learn and use each construct correctly. The language should be uniform in that things that look the similar should behave similarly, and different constructs should really be different and look different. In C, functions are not general since there are no local (nested) functions, yet there are global and local variables. Declarations are not uniform in that class declarations must be followed by a semi-colon, but function declarations should not be. Parameters are not orthogonal to type in Java – primitive variables have copy semantics, but object values have reference semantics (just like arrays versus other types in C/C++).

**Simplicity** as a design goal means to make things as simple as possible, but not simpler. Ideally, this improves readability, writability, and probably efficiency and reliability, but while C is a simple language, one could argue about how well it meets these other measures. Likewise, expressiveness makes it possible to express conceptual abstractions directly and simply and furthers these design goals. We can also design a language to be extensible and allow the programmer to extend the language in various ways (C++ operator overloading). **Security** as a language characteristic means that programs cannot do unexpected damage (Java has this as a prime design feature). A good language should have a precise definition that can answer programmers and implementers’ questions. (Most languages today have precise definitions, but only ML has a mathematical definition.) Being able to run the same on any machine is **machine-independence** and Java is the prime example of a language designed for this purpose (write once, run anywhere). **Restrictability** is the property that a programmer can program effectively in a subset of the full language. This was Stroustrup’s primary design imperative for C++: you shouldn’t have to pay runtime penalties for features you don’t use.

**Programming Paradigms**

In this course, we will look at four basic paradigms for programming languages. These are not the only ways of looking at programming languages, but are a convenient way of roughly categorizing languages by their characteristics. Some languages share characteristics from several paradigms. The four paradigms are: imperative, object oriented, functional, and logic.

The main characteristic of an imperative language is that it is closest to the architecture of the (von Neumann) machine, and is execution oriented and allows assignment. An imperative language basically specifies a sequence of state changing actions. Variables are used to refer to the abstract machine’s memory locations, and the contents (state) changes as the sequence of actions unfolds. The key operation in an imperative language is **assignment**. Examples are Fortran, Algol, and C.

In a functional language, there are no named memory locations – everything is a function call, and values are passed to functions and returned by functions. In particular,
there is no assignment. The key operation in a functional language is function application, i.e., calling a function. Recursion figures prominently in the use of a functional language. Examples are ML and Scheme. The functional paradigm comes from mathematics, and allows programs to be defined more precisely and reasoned about more easily.

Logic languages use a formal logical specification of a problem. These specifications will indicate how to recognize a solution, but not how to find it. The solution will be arrived at through a reasoning process. The key operation is unification. Prolog is a logic programming language. The logic paradigm also has its roots in mathematics – symbolic logic.

In object oriented languages, the focus is on the data abstraction rather than the execution. These data objects communicate with each other and the key operation is this message passing. Object oriented languages may be imperative or functional. Examples are C++, Java.

In addition to considering the different programming paradigms, we also consider the differences between interpreted languages and compiled languages. The compiler statically analyzes program “source” and produces code that is then run on the computer. The compiled program can be executed many times. An interpreted language is dynamically interpreted as it is read and executed by the interpreter. This is for a single execution, and the interpretation of the program source must be done again for all subsequent executions. The latter approach is usually not as efficient, but can more accurately understand the intent of the programmer, as well as allow the program to change dynamically during execution.

Before we look at specific examples of languages, we will first spend some time considering the general structure of languages – that is, how the syntax of a language is formally specified by a grammar.

Syntax and Parsing

A program consists of a string of characters. The compiler and interpreter’s jobs are to determine which of these strings constitute a legal program and arrange for the execution. The process will begin with a lexical analysis. Basically, the lexical analysis will break the string of characters into meaningful substrings. These substrings are technically called the lexemes of the language and categories of them are called the tokens of the language. They can be things like variable names, keywords, operator symbols, etc. Lexemes cannot be broken into smaller pieces, so they are the atomic units of the language. The job of the lexical analyzer is to convert the program into a sequence of lexemes. This is generally done with pattern matching, and in fact the lexical syntax of the language is a regular grammar. The program that does the lexical analysis is usually called a scanner, and is actually an instance of a deterministic finite automaton. The scanner typically ignores and discards white space and comments in the program source, using white space only to the extent it is necessary to distinguish lexemes.

The next phase in the process is syntactical analysis of the stream of tokens. The job here is to determine if the program is syntactically correct, that is, is it a legal program. This stage is not concerned with any meaning of the program, rather just the form of the
program – is it using the legal words of the language, and are they arranged correctly (i.e., is the grammar correct). The result of doing a syntactical analysis is to produce an abstract syntax tree (or parse tree).

Once the abstract syntax tree is built, it can be semantically analyzed and annotated with the semantic actions associated with the language constructs. This gives meaning to the program and provides the information needed to generate code (for immediate interpretive execution or later static execution).

We won’t focus on lexical analysis – that is a pretty straightforward task done by tools like Unix lex. It typically involves regular expressions to specify what character sequences constitute tokens in the language. We are more concerned with how to specify the syntax of a language. Tools are also available for this parsing, notably Unix yacc. This tool gets tokens from lex and according to a grammar we specify and code we provide, builds an abstract syntax tree (or whatever else we want to do). We won’t go over the code provided (that would be part of a compiler course), but we do want to understand how to specify a grammar. A parser like yacc uses recursion and is basically a push down automaton.

Grammars

We need to have some way of precisely describing a language so that we can tell if a given program is a legal program in the language. English descriptions could be used to define the language, but are easily subject to misinterpretation. A context free grammar (from now on, just grammar) is a formalism for specifying this structure of a language precisely. It specifies how the tokens can be combined to produce legal programs and generally reveals something about the structure of programs in the language. A grammar consists of

1) A set T of terminal symbols
2) A set N of non-terminal symbols (sometimes called variables)
3) A set P of production rules
4) A special start symbol S

Although this is a precise definition of a grammar, it is not to be construed as a programmer’s guide to the language. The grammar is the “legal” definition of the syntax of the language but is not likely to be the place you would look if you were trying to learn the language. However, if you were writing a compiler for the language, you would certainly be interested in seeing this specification of the grammar.

Backus-Naur Form (BNF) is a notation used to write down a grammar. The terminals are usually just written as themselves (or sometimes in quotes to emphasize they are literal). Non-terminals are often written between angle brackets and may have suggestive names like <expr> or <statement>. The production rules consist of a non-terminal on the left, the symbol ::= (or an arrow: → ) and the right hand side as a sequence of terminals and non-terminals. Single production rules for the same non-terminal may be combined into a single rule by joining the right hand sides of the rules into a single right hand side with the “or” operation denoted by |. Unless otherwise specified, the non-terminal on the left of the first rule is the start symbol. Another style is to use upper case letters for the non-terminals and lower case letters for the terminals.
A grammar gives the rules for producing all legal programs. The way the rules work is this: beginning with the start symbol, we replace it using any of the rules for the start symbol. In the resulting string, any non-terminals are replaced according to a rule for them. In this way, we keep eliminating non-terminals until we have a string consisting of just terminals. This result is called a **production** of the grammar, and is a legal program according to the grammar. The sequence of substitutions using the rules of the grammar is called a **derivation**.

Here are some examples of production rules that you might expect to find in a language parser:

\[
\begin{align*}
<\text{stmt}> & ::= <\text{var}> = <\text{expr}> ; \\
<\text{var}> & ::= A \mid B \mid C \\
<\text{expr}> & ::= <\text{var}> + <\text{var}> \mid <\text{var}> - <\text{var}> \mid <\text{var}>
\end{align*}
\]

This is just a simple production with start symbol \(<\text{stmt}>\) that would generate assignment statements consisting of an identifier name (A, B, or C), the assignment operator ‘=’, the variable name B, a plus sign, the variable C, and a terminating semicolon. From this little grammar, we could generate ‘A = B + C;’, that is, this statement would be in the language of the grammar. We can see that this statement is in the grammar by the derivation:

\[
\begin{align*}
<\text{stmt}> & \rightarrow <\text{var}> = <\text{expr}> ; \\
& \rightarrow A = <\text{expr}> ; \\
& \rightarrow A = <\text{var}> + <\text{var}> ; \\
& \rightarrow A = B + <\text{var}> ; \\
& \rightarrow A = B + C ; \\
\end{align*}
\]

The derivation can also be represented by a **parse tree**:

```
\[
\begin{align*}
<\text{stmt}> & \rightarrow <\text{var}> = <\text{expr}> ; \\
& \rightarrow A = <\text{var}> + <\text{var}> ; \\
& \rightarrow A = B + C ;
\end{align*}
\]
```

where we read the resulting generated string in the language along the bottom. Notice that in a parse tree, the start variable is at the top, and its children are a production rule. All leaves in the parse tree must be terminals and internal nodes are the non-terminals.

Grammars do not have to be unique. The above grammar could add the rules:

\[
\begin{align*}
<\text{sum}> & ::= <\text{var}> + <\text{var}> \\
<\text{diff}> & ::= <\text{var}> - <\text{var}>
\end{align*}
\]

and change the rule:

\[
<\text{expr}> ::= <\text{sum}> \mid <\text{diff}> \mid <\text{var}>
\]

This clearly generates the same language, but the parse trees would be different since there would be more internal nodes in the example given.
A good grammar captures the logical structure of the language, and like “good” programs, uses meaningful names, and is easy to read and as unambiguous as possible.

Here’s a fragment of the top level of a programming language (terminals are indicated by all caps):

```
program ::= declarations_and_process_list
declarations_and_process_list ::= declarations
| process
| declarations_and_process_list declarations
| declarations_and_process_list process
process ::= SESSION ID body
| SESSION ID LPAREN arg_list RPAREN body
| SUBSESSION dtype ID LPAREN param_list RPAREN body
statement_list ::= statement
| declarations
| statement_list statement
| statement_list declarations
body ::= LBRACE statement_list RBRACE
statement ::= expr SEMI
| SEMI
| compound_statement
| PRINT expr SEMI
| IF LPAREN expr RPAREN statement
| IF LPAREN expr RPAREN statement ELSE statement
| SWITCH LPAREN expr RPAREN LBRACE case_list RBRACE
| FOR LPAREN expr SEMI expr SEMI expr RPAREN statement
| WHILE LPAREN expr RPAREN statement
| CONTINUE SEMI
| BREAK SEMI
| EXIT SEMI
| EXIT LPAREN RPAREN SEMI
| EXIT expr SEMI
| REGION ID cstatement
| SESSION_RETURN expr SEMI
| SESSION_RETURN SEMI
| TRY cstatement catch_list FINALLY cstatement
| TRY cstatement catch_list
| THROW ID SEMI
| THROW ID LPAREN RPAREN SEMI
| THROW ID LPAREN expr RPAREN SEMI
```