Programming Languages

D.E. Stevenson
School of Computing, Clemson University, Clemson, South Carolina, U.S.A.

Abstract
Programming languages are first and foremost languages and hence they can be understood in terms of our knowledge of natural and other formal languages. The concept of algorithm (although not called such) begins with constructive Euclidean geometry. We start there and consider the development of programming languages not as a subject in itself but a natural outgrowth of the requirements to concisely describe computations through language. We start with a short history of constructive and algebraic mathematics. We then introduce the structural theory of de Sassure and Chomsky to describe the various parts of a contemporary programming language. Our goal is not to be technically deep but to explain the thinking behind the various parts of the language development process.

Note to the Reader. This entry assumes that the reader has significant programming experience with at least one higher-level language. We assume the reader has experience with writing and interpreting recursive functions and implementing graph structures.

Goal. After reading this entry, the reader should be prepared to read more technical presentations of computing languages, including graduate textbooks.

Purpose. Our purpose is to provide information technology professionals with insights needed to understand, evaluate, and use contemporary language systems in information technology. Our intent is to use the least technical language consistent with the subject.

Our story. Languages are used to convey information among individuals of a community. No language springs completely formed from thin air; computing and programming have a long history that we explore. To understand the technical details we explore the structure of language using the structuralist theory of de Saussure and Chomsky. With this background we can study the detailed structure of programming languages.

PART I: PRINCIPLES OF LANGUAGE AS APPLIED TO COMPUTING

Introduction

We cannot hope to survey all of the programming languages nor all of the various concepts that have been developed since 1954, the year FORTRAN I appeared. What we can do is explain to programmers how such languages are designed and the broad overview of how they are implemented.

Before going on, let us agree on the use of certain words. Like any technical subject, the study of programming language has a large, complex technical vocabulary. Different authors use words differently, so we need to agree on the informal (nontechnical) meaning of several terms (see Fig. 1).

Context

Algorithms are developed by humans to compute a solution to a problem. Algorithms are comprised of representations and transformations. A programming language encodes these representations and transformations. A programming language is any notation that can instruct a computer how to compute after translation. (Because machine language has no translator, only an interpreter, we do not consider it a programming language.) Computers compute, by which we mean that computers follow a set of rules in the computer’s instruction set to produce a representation of a solution. For this entry, we have in mind the von Neumann model, that is, we do not include multicore, multime- mory, quantum, or analog computers. Such computers have a memory and a central processing unit that follows the program produced from the programming language compiler. The instruction set is a set of operations that the computer’s central processing unit can perform. The compiler transforms algorithms written in the programming language into memory layout and sets of instructions from the instruction set. A computer may have external devices attached to it; these devices have their own programming requirements.

The purpose of language is to transmit information from one entity to another. We can use a modified Shannon–Weaver (Fig. 2) diagram to visualize the process. There is a source message—call it a program—constructed in a source language. It must be encoded, interpreted, and then decoded into a target language.
message—output. We replace the “channel” in the original Shannon–Weaver diagram (Fig. 3) by an “interpreter.”

We are interested in programming languages used to encode algorithms into programs, although the same concepts hold for any formal language-based system. Algorithms have four components: data representation, data storage, transformations, and the ordering of executions to transform data. Notice the recursion! Our approach is to describe programming languages in terms of our understanding of natural and formal languages.
and using the information model as the fundamental mechanism.

Linguistics is the study of natural language, but the more general study of language is semiotics,\cite{1} Semiotics is the study of signs and symbols and their interpretation (use). There are three major themes in semiotics: syntax, which includes orthography (spelling) and grammar; semantics, which deals with meaning; and pragmatics, which deals with how a language is used. Programming languages are formal languages with formal syntax. The use of the term semantics is somewhat an abuse of language: In general use, the term is strictly applied to single words. Semantics of programming languages is discussed at an appropriate time. Informally, in programming languages we mean how constructs are translated. Pragmatics represents cognitive and metacognitive processes of the group and is not necessarily formal; in effect, pragmatics equals “current best practice.”

A SHORT HISTORY

In order to make sense of the subject, it is worthwhile to consider a short history of the development of formal languages. The reader is urged to read Gleick’s *The Information.*\cite{3} The first historical find of an arithmetical nature is a fragment of a table: The broken clay tablet Plimpton 322 (Larsa, Mesopotamia, ca. 1800 BCE) contains a list of “Pythagorean triples” that was laid out like a spreadsheet. The appropriate place, then, to discuss programming languages is the origins of algorithmic thinking and notations.

Constructive Mathematics

India and Mesopotamia are the birthplace of arithmetic. While the Greeks were primarily interested in logical relationships, Eratosthenes and Menaechmus were interested in constructing solutions. The Islamic mathematicians were interested in the computation of values. The best known of these mathematicians is Abū ‘Abdallh Muḥammad ibn Mūs al-Khwārizmī (c. 780–850) who introduced algebraic techniques for solving linear and quadratic equations. Al-Khwārizmī is also credited with introducing the term *al-jabar,* which is transliterated to *algebra.* Modern algebraic notation evolved through the seventeenth century and extended with the development of the Calculus in the eighteenth century. Theorem 5 of Newton’s *Principia* is the algorithm we know today as “Newton’s Method for Finding Roots.” Newton and his contemporaries develop many tabular methods for various operations in order to compute values of derivatives and integrals. Modern numerical analysis is built around Taylor’s theorem. Concomitantly, other forms of computation were developed such as logical notations and constructive geometry to name two. The modern computer allows us to compute a broad range of elements.

L. E. J. Brouwer, in his 1907 dissertation, adopted the notion that mathematical objects exist if and only if they can be constructed by algorithms. There are several branches of such constructive mathematics. We take Church’s, Turing’s, and Markov’s view, which is that constructive mathematics is essentially recursive functions. The modern version of this idea is due to Per Martin-Löf. His thinking is most obvious in strongly-typed functional languages. Every object in a computation has a representation, and the class it belongs to is called a type. So, for example, 123 is not just three characters but it represents something of “integer type”; but then again DXXIII is the same object with a different name. Following Martin-Löf\cite{7} “A type is defined by describing what we have to do in order to construct an object of that type.” A full development of Martin-Löf theory leads naturally to functional notations.

The Lambda Calculus and Turing Machines

Two important theoretical ideas were proposed in the 1930s: the lambda calculus by Alonzo Church and the Turing machine by Alan Turing.

Alonzo Church and the Lambda Calculus

The period 1880–1940 was an active one for mathematics—it was having a crisis of confidence. The foundations of mathematics were brought into question by Russell’s Paradox: Informally it states that one cannot have a set of all sets. One of the questions this raised was exactly how recursive functions worked in terms of variable binding and substitution. Space prohibits describing the theory in full.

The major operational ideas of the lambda calculus are a fundamental part of programming languages. All functional languages such as LISP, ML, and OCAML have a construct called an anonymous function—a
function with no name—which is just lambda expressions. Most other common languages hide this by requiring all functions (and subroutines) to have a name. We are all familiar with the functional schema:

Name (arg₁, ..., argₙ) Body.

One of Church’s insights was an understanding of binding and scope. Church laid out the scope and binding rules we use today. But he also realized that name was just a binding for the remainder of the statement. Thus, in the lambda calculus,

Name : = λ(arg₁, ..., argₙ) Body

where “:=” means “equal by definition.” But how do we make sense of these symbols?

We introduce three new concepts: evaluation, application, and rewriting. Evaluation means ordering the calculations, application is the use of a function, and rewriting is obvious. For example, suppose we have “add by one” or incr : = λ(n)(n + 1).

<table>
<thead>
<tr>
<th>Incr (3)</th>
<th>Apply “incr” to 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(λ(n)(n + 1))</td>
<td>Substitute “incr’s” definition</td>
</tr>
<tr>
<td>(3)</td>
<td>Apply lambda’s semantics: substitute and apply “+” to its arguments (3,1)</td>
</tr>
<tr>
<td>4</td>
<td>Rewrite as “4”</td>
</tr>
</tbody>
</table>

Alan Turing and the Turing Machine

In 1928, the German mathematician reminded researchers of a problem, originally posed by Gottfried Leibniz, known in German as the Entscheidungsproblem (in English, “decision problem”). The problem asks for an algorithm that takes as input statements of a first-order logic and answers “Yes” or “No” according to whether the statement is universally valid (true in every structure satisfying the axioms). In 1936 and 1937, Alonzo Church and Alan Turing, respectively, published independent papers showing that a general solution to the Entscheidungs problem is impossible.

Turing reformulated an earlier proof by Kurt Gödel’s (1931) on the limits of proof and computation, replacing Gödel’s universal arithmetic-based formal language with the formal and simple hypothetical devices that became known as Turing machines. In modern terms, think of a computer with a very simple instruction set and an indefinitely long linear storage. Basically, the CPU can read and write a very simple alphabet (0, 1, blank) and move to an address one higher or one lower than its current location. Even with such a simple machine, Turing was able to prove that some such machine would be capable of performing any conceivable mathematical computation if it were representable as an algorithm. He also introduced a “Universal Machine” (Universal Turing machine), with the idea that such a machine could perform the tasks of any other machine, or in other words, is provably capable of computing anything that is computable.

These two theoretical results are the basis of the theory of computation and play a key role in the development of algorithms and programming languages.

Short History of Programming Languages

The evolution of programming languages begins in the mid-1850s and continues to the present day. Even more fascinating is the claim that there are 2,500 programming languages.[⁴]

Programming languages from the beginning

We recognize Augusta Ada King, Countess of Lovelace (10 December 1815–27 November 1852), as the world’s first programmer and programming language inventor. She documented all her work on Babbage’s “Engines” and developed a notation for programming. But even she stood on the shoulders of giants: Leibniz provided a machine design in 1671 (constructed in 1673) and was a champion of the binary system. Algebraic and logical notations expanded more or less intuitively until Church and Turing introduced the Church–Turing conjecture: “Every effectively calculable function (effectively decidable predicate) is general recursive.”[⁵] In 1963, John McCarthy wrote a seminal paper “A basis for a mathematical theory of computation” that cemented this relationship with programming languages.[⁶]

Historical development of computer languages beginning with FORTRAN, ALGOL, LISP, and COBOL focused on algorithmic information needed: 1) to describe representation of information structures in computer memory, 2) to specify fundamental operational “algebras,” and 3) to prescribe operational order. ALGOL, FORTRAN, and COBOL represent imperative languages, so-called because they are based on a command-style paradigm. LISP is the original functional language, being based on the idea that every computation is a function that returns a value. Logic languages, a third major language group, arrived in 1972 when Prolog became available. Logic languages are based on the concept that a program is a proof of a theorem.

Language developments are based on the experiences and needs of the using community. The concept of object-oriented languages began with Simula-67, a discrete-event simulation language that naturally used objects in modeling. This led eventually to Smalltalk and today’s object-oriented languages. Objects became a major paradigm for design with languages such as Java.
so that today object-oriented programming systems abound in all problem areas.

As another example, E. Dijkstra analyzed the concept of the “goto” construct and showed that the unrestricted use of the “goto” construct was bad practice; current languages such as Java do not have the “goto” among its constructs. The programming language syntax does not have the goto, but the machine language does. This shows that programming languages can be used to control the use of machine capabilities.

Between 1965 and 2000, computing hardware also radically changed, leading eventually to the supercomputers and multicore (multiple CPUs). Hardware changes made programming languages evolution inevitable. The Cray I and Cray II computers required FORTRAN to introduce constructs that enabled the programmer to control parallel computation. In 2012, the frontier is multi-core. NVIDIA’s CUDA was specifically designed to take advantage of multiple cores.

Language zoo

From the beginning of language design, there have been different paradigms on how the features of the language should be presented. The ongoing discussions are often called the “Language Wars,” which leads to the “Language Zoo.” This entry cannot describe each language in the Zoo or even do justice to the unique paradigms in use. But we do not have to. The Church–Turing conjecture means that any programming language that can implement recursion is equivalent to any other such language. So why are there so many programming languages? It has to do with the fact that the human mind works using mental models in symbolic systems. For that reason, the next section explores semiotics, how humans process language.

Keeping it all straight: Language and meta-language

Before proceeding, it is necessary to make a distinction between a “programming language” and “the language used to describe the programming language.” The former is called the object language and the latter is called the meta-language. We will drop the term “object” unless required by context to differentiate the two. In the classroom, English (or the local native language) is the meta-language because it is used to describe how the object programming language being taught operates.

So why does it matter? Let us take the C programming language as an example. The C compiler can be written in C. So C as the object language is described by the C used as a meta-language. So a programming language that can serve as its own meta-language is bootstrap-able. As another example, there are several notations used to describe programming language grammars. Is there just one notation that can describe them all? According to the Church–Turing conjecture: no, there are many such languages.

PART II: SEMIOTICS AND PROGRAMMING LANGUAGES

In Part I, we have explored many of the ideas and experiences that have gone into programming language evolution. We look at the more general issue of the technical qualities of all language. While linguistics is the study of natural language, we abstract to all languages and use semiotics as our framework.

LANGUAGE FROM THE SEMIOTICS VIEWPOINT

We are interested in language as a formal system of signs governed by formal grammatical rules to communicate meaning. This is known as the structuralist view of language, introduced by Ferdinand de Saussure, and remains foundational for most approaches to language study today. Noam Chomsky, who defines language as a particular set of rules, helped popularize the structuralist view in computing languages. Chomsky’s hierarchy is a categorization of computing languages and the problems they can solve.

We use language to maintain or transmit information, and language must be interpreted to recover this information. The interpreter’s function is to convey every semantic and pragmatic element that the source-language utterance carries to target-language utterance. (“Interpreter” has many different technical meanings. We use the most general meaning: it takes in a source sentence and outputs sentences in the target language.) Information must be preserved in this process. Whether human or computer, the interpreter will take in a complex concept from one language, choose the most appropriate vocabulary in the target language to faithfully render the source message in a completely equivalent target message.

Semiotics proposes that languages are composed of three elements: syntax, semantics, and pragmatics. Syntax is comprised of orthography and grammar, which specifies the form of correctly formed words and sentences in the language, respectively. Semantics gives the meaning of the individual words by associating the word with its meanings. Pragmatics, on the other hand, is not mathematically rigorous. Pragmatics explains how the language is used in practice—think of common, everyday conversation as pragmatics.
Syntax

Syntax deals with the form of words and sentences. In order to do this, the compiler has to recognize words, associate words with definitions, determine the classification of the words, and relate the words by their classification by predetermined relationships in the grammar.

Words: Orthography, lexicon, and vocabulary

A word is a single distinct conceptual unit of language. In every language, there are rules governing orthography, rules for the formation of proper words—Chinese is an example of a symbolic language in which each word has a distinct symbol: English has 26 characters, Chinese over 20,000. A lexicon is a set of words in a language and a vocabulary is a lexicon, each word of which has one or more definitions associated with it.

Grammar

Grammars are bodies of rules that describe the structure of “sentences” in the language.

Formal grammars are prescriptive rules that delineate the entire language, and the development of a programming language grammar is based on the structures we want to incorporate in the language.

There are many notations that have arisen for specifying orthography and grammars in programming languages. There are many stable programs to specify these elements that remove the tedious coding involved in implementing these elements. The original—still available and highly regarded—programs were YACC and LEX. Conceptually, these are formulated in the same way through the use of productions and grammars in formal languages or automata. Rather than exploring these, we will examine the underlying concept of a transformation based on pattern matching (see Fig. 4).

Semantics

Semantics is concerned with the meanings of expressions. There are a number of branches of semantics in human speech that we will not explore. In programming languages there have been three standard models to semantics: 1) axiomatic (connotative), uses mathematical logic to specify meaning; 2) denotational uses recursive structures to show how values are computed; and 3) operational constructs a specification of an abstract machine that acts as the interpreter. Operational came first and is the most intuitive: Most programmers of the time were assembler-level programmers and could design a machine that would give the “usual” meaning to a construct. Axiomatic came as the desire for high quality, logically correct programs became important. In axiomatic semantics, we are interested in the assertions (inferences) we can make based on the algebraic rules of the language. Axiomatic views develop the abstract meaning or intension of a term, which forms a principle determining which objects or concepts it applies to. Denotational semantics refers to the object or concept to which a term refers, or the set of objects for which a predicate is true. Christopher Strachey and Dana Scott developed denotational semantics from recursive function theory.[8]

Pragmatics

The essence of the usability of language is pragmatics. Have you ever wondered why there are so many programming languages? Have you wondered why it is so hard to learn a completely different programming language? The answer is pragmatics. This is the cause of
the “language wars.” Try this experiment. Use your search engine on the exact term “language wars.” It is not just programming languages that have such problems.

What is the difference between semantics and pragmatics? In the C programming language, there are two obvious ways to implement the thought “increment the value (named by the variable \( x \)) by 1.” One way is to write “\( x + + \)” and a second way is to write “\( x + 1 \)” The semantics are completely different: The value of \( x \) is changed in the former and not the latter. We decide which form to use based on our pragmatic needs at the time. But the two forms are pragmatically useful.

**Holistic Summary of Language and Language Learning**

Language is used to transmit information between the sender and the receiver. In our context, there are two classes of receivers: computers and humans. Computers have defined formal semantics, which we discuss at the appropriate time. Passing information from human to human by means of a formalized computer program is the subject of this section. Passing meaning from one human to another means we have to understand a bit of psychology.

**Cognition and Metacognition**

Cognition is the mental process of acquiring knowledge and understanding through thought, experience, and the senses. Metacognition is awareness and understanding of one’s own thought processes—this is often said to be “cognition about cognition.” In our context, language acquisition is a cognitive process while the realization that one does not understand a sentence is a metacognitive process. Metacognition relates to our knowing strategies to approach problems and control on our thinking (“Is this right?”). These strategies often play a seminal role in programming design. “Structured programming” of the 1970s is a strategy of writing programs with no uncontrolled transfer of control (“goto”). Metacognition increases with experience leading to expertise.

What do cognition and metacognition have to do with programming?

Firstly, language acquisition is a cognitive process. While programming language syntax is formal and perhaps easier to learn than a natural language, the semantics and pragmatics are often byzantine. On the other hand, mastering the pragmatics is a metacognitive task, for example, transferring strategies from one language to another.

Programming language texts that focus on built-in structures and features are primarily cognitive in nature, providing the semantic symbol knowledge. On the other hand, texts that focus on algorithms—especially language implementation independent ones—are metacognitive in nature because they provide strategies. It is important to realize that a certain level of expertise with languages is required to understand and assimilate algorithms.

**Role of pseudocode**

Pseudocodes are informal languages often used to describe algorithms so that any programmer, regardless of programming language experience, can read and understand the intent of the algorithm. One common use for pseudocodes is in textbooks. Programmers use pseudocode to “sketch out” an algorithm in varying levels of detail without strictly adhering to a formal syntax. Another use is to use the pseudocode as a draft mechanism, much as one might write several drafts for a technical paper with each draft more specific and closer to the ultimate implementation form. Pseudocode is a natural product of our cognitive and metacognitive use of language in problem solving.

**SURFACE DETAILS OF COMPUTER LANGUAGES**

Chomsky[2] developed the idea that each sentence in a language has two levels of representation: 1) the surface language and 2) the deep language. The deep structure represented the core semantic relations of a sentence and can be mapped to the surface structure via transformations. Strictly speaking, Chomsky’s concept of natural language is to convert the deep structure to the surface structure using transformations. In programming languages, we have the inverse: We are presented with a surface sentence and must transform it to the deep structure.

The surface language, which behaves according to the syntax rules, is the visible language that we read. The deep structure captures the semiotic relationships in the surface language and in our context the deep structure is a graph. In order to simplify the presentation of the technical details of implementing a computer language, we think of the transformation as occurring in three steps: 1) user-written code (surface language) is rewritten into a data structure (deep structure); 2) then a series of graph transformations from the source language to the target language; and finally 3) writing the final graph into interpreter/machine usable form. The graph formations in step 2 represent the translation of the source semantics to target semantics via the
Surface Features

This section is organized into subsections: 1) types and operations; 2) memory operations; and 3) control operations.

Types

Data representations and transformations are, technically speaking, algebras. Algebraic structure refers to a set of objects; transformations are “functional rules”; and relations are “logical rules.”

The most familiar of these algebras is the integers with two operations: addition and multiplication. Remember, though, that the computer’s integers are not the same as the formal mathematical integers and this is a metacognitive correction we must make when computing. For example, there is a maximum computer integer—call it maxint—and maxint + maxint is meaningless because the result is not predictable on must modern machines. There are many algorithms for integer operations. The actual bit value for maxint + maxint depends on which algorithms are used and hence the value is unpredictable.

There are many formal ways to present algebra systems but the most common mechanism for computing is the Martin-Löf type system.

A type consists of a set of constants, operations on those constants, and relations on those constants. Each operation or relation has a signature that gives the number and types of the arguments (arity). A common computer type is the 32-bit integers (but the size is set by the hardware). In this case the constants range from $-2^{31}$ to $2^{31} - 1$ with the usual operators (+, -, *, /, mod) and relations (=, $\neq$, <, >, $\leq$, $\geq$). We immediately notice that there are more negative constants than positive ones; therefore, $-(2^{32}) = 2^{32}$ in the arithmetic integers but the value is unrepresentable in 32-bit integers. This is just the tip of the iceberg, which is why there are both cognitive and metacognitive adjustments to be made when programming; you expect the numbers to work the way they always did in grade school, but they don’t.

A type system maps values and expressions into types. The type system also defines how the types themselves interact.

From the programming language designers’ standpoint, then, the rule is simple: For every type, one must be able to represent the constants and have algorithms for each of the operators and relations. Some of the types are predefined by the compiler and whatever else is needed requires “user-types.”

Variables and bindings

The reader undoubtedly knows the difference between mathematical variables and programming language variables: We solve for mathematical variables for values but we use programming variables to store values. A value is bound to a variable. Variables have both a type and a scope. A variable’s type defines what types of constants can be stored in a variable and the scope of a variable is the time the variable is defined and its value is available in memory.

A word about scope. Scope rules originally came from quantifiers in logic. The concept is simple: A variable’s value is “available” from the beginning of its scope to the end. In practice, this is not so simple. Typical programming practice has several different scopes:

- Language definition
- Compiler implementation
- Library implementation
- Compile time (there could be several here)
- Run time

Names that are not predefined must be declared for type in many (but not all) modern languages. In terms of syntax, such declarations are a sublanguage; this means that it appears under very specific rules.

Memory operations

During the execution of a program, we can follow its computation by looking at the state of the program. The state of a program is the list of all the variables (including, perhaps, operating system variables, program counter, registers) and their current value. The state of a program is changed by the assignment of a different value to some variable, hardware/software interrupts, and keyboard keystrokes to name just a few. During declaration, a name is assigned to a memory location; this location is called an l-value. The value stored in the memory location is called the r-value. These terms come from the usual syntax for assignments:

\[ l\text{-value} = r\text{-value} \]

with pointers being a type of l-value and a legal constant of the type of the variable.

Let lvalueof (x) return x’s l-value and rvalueof (x) its r-value. We would like the following identity to be true:

\[ \text{rvalueof (lvalueof (x))} = x = \text{lvalueof (rvalueof (x))} \]
This is not true in most languages (try it in your favorite language). The right hand side of this expression is what we mean by associative memory.

Control operations

The difference between the practice of (nonconstructive or Platonic) mathematics and the practice in computing is that the values in computing must be constructed. Foundationally, we want to know what is the minimum set of operations that are needed to control these algorithms and constructions. There are three: conditionals, iterations, and invocations (call-return).

**Conditionals.** Conditional statements are composed to three things: a logical value and two sets of statements. This is the first nonintuitive construct we have encountered, primarily since it does not normally occur in usual mathematics. We can use this to illustrate the whole thought process for compiling.

**Syntax:** if (expression) then statement-1 else statement-2 fi

If, then, else, and fi are known as “keywords.” In most modern languages, keywords are symbols that look like variables but cannot be used as variables. The parentheses are known as “grouping symbols.” The expression indicates that any valid pattern recognized by the “expression recognizer” is acceptable. Statement-1 and statement-2 are similar to expression but rather a pattern recognized by the “statement recognizer.” The output of the pattern recognizer for such a statement is (IF tree (expression) tree (statement-1) tree (statement-2)). Notice that then, else, and fi are not represented because they have no semantic value. The if has been replaced by the IF statement, indicating a semantic value/function.

**Semantics.** The informal semantics of this statement can be expressed by the following statements. “Evaluate the expression and convert to a Boolean value (either true or false). If the value is true, then arrange that statement-1 is executed and statement-2 is not. Similarly, if false, then statement-2 is executed and statement-1 is not.” Regardless of what form of semantics we use (algebraic, denotational, operational), the informal meaning of the statement must be translated to the formal.

**Pragmatics.** The pragmatics of the if-statement is when and how one uses such a statement. Since in programming we are always working with values, the if-statement can be used to prevent errors such as dividing by zero, checking sort order, controlling formatting as simple examples.

**Iteration.** Iteration is a “looping” construct that causes a series of statements to be executed repeatedly.

There are two general forms, one that stops at the end of a count and one that stops when a particular Boolean value is true.

**Invocation and Packaging.** Programming would not be practical if we did not have a way of packaging statements in such a way as to reuse the packaged code.

**Functions and subroutines.** Most languages borrow from the mathematical notation of name (argument1, ..., argumentn). This notation is used in two ways: 1) to invoke the code; and 2) to define the code that should be executed. In practice, the term function implies that a value is returned and subroutine implies no value is returned. Here we will use the term routine unless the distinction is needed.

**Defining functions and subroutines.** Definition is the association of a signature (head) such as name (argument1, ..., argumentn) with a body of code. Depending on the particular language, the body may contain other definitions or executable code such as iterations, conditionals, and assignments. Originally, the desire was that the code should act as if it were substituted in place of the head—for technical reasons, this does not work well in practice.

**Binding argument names.** How the arguments in the head and the body are bound is specific to the language. In the simplest case, the arguments must be single variables and those names are taken as the value holders in the body. There can be many more complex rules.

**Invoking a routine.** Most languages interpret a head without an accompanying body declaration as a routine invocation. That is, we expect the computation to jump to the body of the routine (which must be previously defined), execute the body, and then return to the point of invocation.

**Other Packaging Issues.** While routines are the most common packaging concept, many languages have more: 1) structured data; 2) objects; and 3) file structures.

**Structured data.** Structured data is defined in terms of primitive types (the provided types) and other structured data, for example, a program that processes a file of addresses into mailing labels. Each address has five fields: name, street address, city, state, zip code. The data for these are “packaged together” as a unit.

**Objects.** One can think of objects as an extended form of structures. Objects act like algebras in that the object definition includes constants, functions, and relations.
Many applications naturally lend themselves to object-oriented concepts.

Files. Since programs are written and stored, there is a question of how the files may be organized. Some languages will only accept one definition per file; others will accept any number. In current systems, files, networks, and so forth are primarily the purview of the operating system. Therefore, the interface is established and the programming language works within that framework. Some languages have extensive file operations as part of their language (COBOL, PL/I) or none at all (C).

Application interfaces. The 1960 ALGOL Report mentions that the main purpose of programming languages is to develop libraries of programs for applications. Many of the innovations by object-oriented languages and strict typing languages were initiated by the difficulties in accomplishing the purpose. An application programming interface (API) is a software specification that defines an interface by which software components communicate. An API may include specifications for routines, data structures, and variables. In other words, APIs are algebra specifications.

Summary
Not surprisingly, our familiarity with various programming languages is through the surface languages presented to the programmer. How much work the programmer has to do and how much the programming language must do is the decision of the using community. But our total understanding of any particular language includes the association of semantics and pragmatics to the surface constructs. This means that our understanding of the exact workings of the language is filtered.

DEEP STRUCTURE OF PROGRAMMING LANGUAGES
The parser converts the surface structure of the program into an equivalent deep structure. These structures go by many names (parse tree, abstract parse tree, annotated parse tree) in the literature but we will use directed acyclic graph to encourage the reader to visualize programs as graphs. The language LISP was originally developed to accomplish the tree transformations so its input is in the form of a linearized tree.

Most parsers have two components: a scanner (or lexer) and the parser proper. The scanner reads the input character by character, forms individual words, and constructs a token consisting of the input word and its type. The parser takes the tokens and applies rule patterns that make up the language’s grammar. As described in Fig. 5, the “parse tree” is formed based on the substitution rules used. Both the scanner and parser rules typically are developed from automata and/or formal language principles. There are many tools that can be used to develop both the scanner and the parser so there is little reason to hand-code them.

Diagramming Sentences as Parse Trees
A diagram is a graphical picture of the relationships of the words of a natural language sentence. If you were fortunate enough to have a teacher who required extensive diagramming exercises you can skip this section. This section is for those who were not so fortunate.

English grammar is not about words per se, but about parts of speech and how these parts function in the sentence. There are eight parts of speech in English from which we can construct complicated sentences. Unfortunately, many words have many possible ways they could be used making the understanding context sensitive. In formal languages, where every word has a type, we use type as part of speech.

As an example, consider the sentence “man bites dog.” This has the form of “noun verb noun” and has a simple diagram of “noun | verb /noun.” Now consider the simple assignment statement “x = 1.” This has the form “variable = constant.” Now, rather than putting this into diagram form, we put in into the form of a tree with the operator at the root, which linearizes as “( = variable constant).”

We now have enough information to understand the entire surface to deep structure transformation. The scanner reads the program and classifies each word as
A simple example illustrating the generation of a simple declarative sentences are formed. In this example, S is a sentence, D is a determiner, N a noun, V a verb, NP a noun phrase, and VP a verb phrase.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S := NP VP</td>
<td>Start with S</td>
</tr>
<tr>
<td>NP := D N</td>
<td>Use &quot;NP VP&quot; as substitute for &quot;S&quot;</td>
</tr>
<tr>
<td>VP := V NP</td>
<td>Use &quot;D N&quot; as substitute for &quot;NP&quot;</td>
</tr>
<tr>
<td>D N V NP</td>
<td>Use &quot;V NP&quot; as substitute for &quot;VN&quot;</td>
</tr>
<tr>
<td>D N V D N</td>
<td>Use the lexicon to substitute for letters</td>
</tr>
<tr>
<td>The dog ate the food</td>
<td>Final form. We have generated the target sentence</td>
</tr>
</tbody>
</table>

Our lexicon is simple: (1) dog and food are each nouns; (2) the is a determiner; and (3) ate is a verb.

Example. The dog ate the food.

Derivation.

Deep Structure Manipulations

There are three standard ways to describe the manipulation of deep structures: algebraic/axiomatic, denotational, and operational. The author believes that the denotational is the most natural (your taste may differ), partially because it is based on functional programming (actually the other way around) and partially because it more naturally fits the manipulation of natural and formal languages.

IMPLEMENTATION STRUCTURE

This section discusses how all the concepts can be used to design an implementation. There are many tools available for syntax operations. These tools make it possible to put together an ad hoc command language in short order. A common approach to implementations for portable languages such as C or Java is to “bootstrap”: the compiler is written in itself and a “throw-away” code generator is constructed for the first version (see Fig. 6).

This section is organized to coincide with the syntax, semantics, and pragmatics theme used throughout. The first subsection describes the implementation of dictionaries (“symbol table”), which handles the duties of lexicography. In the syntax subsection we look at orthography, lexicography, vocabulary lookup, and grammars. The outcome of the syntax operation is a graph. The next subsection describes operations on graphs to implement both semantics and pragmatics. Finally, we discuss code generation.

Symbol Tables and the Handling of Dictionary Duties

We all know what dictionaries are, and as the world becomes more integrated, dictionaries fulfill the need for exact understanding of words. Historically in compilers, the dictionary is called a “symbol table.” We will stick with dictionary because it is the exact metaphor we want and something we can go look at for insights.

Dictionary

Dictionaries are sets of entries, with each entry a pair (key, entries). The key could be a single word or more complicated, such as a phrase. Considering a simple program, every word is a symbol and hence a key. Any key can have multiple definitions, or unique definitions.
in different contexts. Some symbols—call them grammatical symbols—need not be in the dictionary: parentheses in most languages have no semantics so they can be resolved by the scanner. Be that as it may, conceptually every symbol has a dictionary entry.

Scope, context, and binding

Most languages allow for a symbol to be defined multiple times in a program provided it is uniquely defined in scope. Most scope delineations are in the program’s control and indicated in using special type constructs: definitions of function heads or begin...end type constructs. How and where scopes may be used is defined in the grammar and in the semantics routines.

Binding refers to the association of a definition with a key. A formal definition of scope is the time between when an entry is bound to a symbol and when it is unbound.

Algorithms

Any searchable list can be used to implement a dictionary. At this point, the choice is strictly a data structure/algorithm choice. Any vocabulary of more than 100 words should be a hash table. There are many APIs for hash table maintenance: choose your favorite.

<table>
<thead>
<tr>
<th>Table 1 BNF grammar for Ol’ faithful</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;F&gt; eof</td>
</tr>
<tr>
<td>2 &lt;E&gt; := &lt;T&gt; + &lt;E&gt;</td>
</tr>
<tr>
<td>3 &lt;T&gt; := &lt;E&gt;</td>
</tr>
<tr>
<td>4 &lt;F&gt; := &lt;F&gt; * &lt;T&gt;</td>
</tr>
<tr>
<td>5 &lt;T&gt; := &lt;F&gt;</td>
</tr>
<tr>
<td>6 &lt;E&gt; := &lt;E&gt;</td>
</tr>
<tr>
<td>7 &lt;F&gt; := Any (integer)</td>
</tr>
</tbody>
</table>

Grammars: Orthography and Structure

The actual process for orthography and grammar checking is similar. Traditionally, programming languages were designed for English speakers and hence the characters were the standard ASCII alphabet. Grammars are due to the work in the 1950s and 1960s, notable by the researchers in computability and computational linguistics. In 1965, Donald Knuth published a paper “On the Translation of Languages from Left to Right” that pulled together the disparate approaches to grammars. This has led to a plethora of tools for constructing programs to do both scanners and parsers. In this section, we present a unified discussion of how to think about designing input for these programs.

The essence of a design is the definition of a set of rules all having the same form variously known as Backus Normal Form, Backus Naur Form, or BNF. BNF can be interpreted either bottom-up or top-down. It is somewhat more natural for humans to interpret it top-down, while bottom-up is generally more efficient given a proper formulation of the grammar:

<symbol> := expression [actions];

Table 1 shows “Ol’ Faithful” as a BNF table. It is called “Ol’ Faithful” because it seems to appear in every textbook with the intent of capturing all syntactically correct integer statements that include “ + ” and “*.” The grammar as written in Table 1 BNF Grammar for Ol’ Faithfulis a language and therefore needs an interpreter.

Put ‘E eof’ as workstring
Do
Start at the front of the string.
Find first Nonterminal in workstring.
Replace Nonterminal with each right hand side.
While there are Nonterminals.
Does one of the generated strings match the input?
YES? Accept
NO? Reject

As an example (Table 2), we show the generation of the string “1 + 2*3 eof” using the recursive descent algorithm. A recursive descent parser is a set of mutually recursive procedures where each such procedure usually implements one of the production rules of the grammar. The pattern of the rule must match before the action can be executed.

This concludes the tutorial part on syntax. Industrial-strength programming languages are much more complex and the rules are correspondingly more complex.
Comparing Semantics and Pragmatics

In many texts the term *pragmatics* is not used, pragmatic issues are mixed in with semantics. Since we are studying the whole idea of developing and implementing a programming language, this is the time to show how they occur. Programming languages are formal languages; hence, a semantics is provided when an interpretation or model is specified. So, for example, when the statement “\(x + 1\)” is encountered, semantics tells us how to make the types match and how to compute the answer.

On the other hand, the principles governing appropriate “conversational” moves are called pragmatics. A pragmatic treatment of a feature of the use of a language explains the feature in terms of general principles governing an appropriate manner of expressing thoughts rather than in terms of a semantic rule. The rule that a variable should be initialized before it is used is a pragmatic rule not a semantic one.

The take-home message is that the syntax defines the surface structure of the language but the semantics and pragmatics define the interpreter. Unfortunately, the term *interpreter* connotes a type of program; where there may be confusion, we will use the term *PL interpreter* to mean the general concepts and just *interpreter* where the programming language system is meant.

Programming Language Interpreter Design

The PL-interpreter takes the syntax trees that represent the (raw) program and transforms it to a final tree. This final tree drives the output. The PL-interpreter can be simple or complex. Interpreting a single DOCTYPE (yes, it requires an interpreter) to produce text can be simple; compiling code for a heterogeneous supercomputer can be quite complex.

The design of an interpreter is again patternbased. In this case, the patterns are based on the format of a graph. The most complicated graphs occur when significant optimization is required. Although the rules can be mixed in various passes over the graph, it is conceptually easier to think of many multiple passes each addressing one issue. For example, the documentation for gcc lists 23 passes (http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_14.html on 29 Mar 2012). These passes perform various actions such as external name declarations, including linkages for library routines, and optimizing operations, for example. The various passes may produce files that will be included later as well as the so-called object file.

While production level compilers produce programs that are processed and set into loadable form, one could also produce programs that are input to another compiler. This can be done in several ways. Source-to-source compilers (transcompiler or transpiler) transform the input language into a source program for another language to be compiled on the same machine. If the target language is on another type machine, then the process is called cross-compiling. Either approach is the fastest way of producing a new compiler.

An intermediate form of this approach is used by the GNU compilers. Simplistically, GNU compilers are two separate programs: a front end and a back end. The front end is a source-specific parser that produces an abstract syntax tree of the source program. This tree is passed to the back end, which is common to the GNU line. The back end produces the machine code for a particular machine.

A third approach is used by Java. The java compiler produces codes for an interpreter known as the Java virtual machine (JVM). The JVM is similar to the series of interpreters in the forth line. Forth input is written in postfix style making it possible to bypass a complicated parsing/semantics phase.

SUMMARY AND THE FUTURE

Summary

The purpose of any language is to communicate information from a sender to a receiver. For programming languages, the information required is that needed to pass an algorithm from human to machine. Programmers use the source language to encode the algorithm. To encode an algorithm, we need information about representation of data, available operations, and the sequencing of operations. At this time (June, 2012), there are at least 2,500 programming languages extant. But based on the Church–Turing conjecture, these 2,500 languages are isomorphic in the sense that they can each be used to encode recursive functions.

Learning a programming language is a metacognitive task, while using a language is a cognitive one. When we learn a language, we must master three aspects: syntax, semantics, and pragmatics. Syntax deals with the surface language: how the language appears to the writer through orthography and grammar. Semantics is the meaning of particular symbols. Semantics comes in two sorts: predefined and userdefined. For the most part, programming language textbooks focus on syntax and the semantics of predefined symbols. Pragmatics is how the language is used—perhaps the clearest examples are application program interfaces.

Most textbooks do not distinguish among syntax, semantics, and pragmatics. Because of this, a learner of a new language—even ones as close as C and C++—may find the switch difficult. By and large, orthography is common among various languages if the standard ANSI character set is used. Grammars pose the major
immediate obstacle, since it defines the surface language—however, learning successive languages should be easier. One would hope that the semantics of the arithmetic operators would be the same. However, once we consider types other than the standard for which a universal vocabulary is available there are plenty of opportunities for confusion. New pragmatics is the most difficult of all because the learner must use a new way of writing and thinking.

History Repeated

For deep understanding of programming languages, it pays to examine the evolution of expression. At the core of each language is a machine, often called a virtual machine. Though the term is greatly extended now, its original meaning indicated the data and functions that were the base on which the language began. The original virtual machine was not so virtual: it was a very rudimentary operating system through the first and second generations. Programming in the late 1950s and early 1960s was accomplished using assemblers. But assembler language programming is very error prone: Large systems were difficult to develop and maintain.

The answer to this engineering and economic problem has been the Language Zoo. Researchers and practitioners both searched for ways to avoid known deficiencies in current languages. By 1965, there were seven standard languages: Algol-60, APL, COBOL, FORTRAN, Jovial, LISP, and SNOBOL. FORTRAN and COBOL were the dominant commercially used languages. Algol was the lingua franca of research along with APL, LISP, and SNOBOL. Jovial was used for U.S. Military applications. Each of these languages had a different semiotics: For example, Lisp is built on recursive functions and SNOBOL was based on pattern matching. From these the 2,500 sprang. For most commercially viable languages, layers of changes were added to some previous language. The introduction of structured programming in the 1970s removed the GOTO construct, making for more easily comprehended programs with fewer bugs.

Occasionally, a completely new paradigm is developed and becomes popular. Smalltalk-80 introduced object-oriented capabilities. Software engineers have developed significant improvements to the software engineering processes by using object-oriented concepts to match with models of problems. The actual viability of C++ came from the fact that there are very few internal changes necessary to C to implement C++. In effect, C++ was hidden in C from Day 1.

The Future

What will the future programming languages look like? What features will they have? What will the next languages look like?

The answer is that we do not know yet. The emphasis on semiotics, information, and (meta)cognition is a conscious attempt to show the reader that programming languages are a natural development of hardware, development methodology, and paradigm preferences. The most obvious candidate for disruptive programming languages is quantum computing. Quantum computing is disruptive because it and classical computers have fundamentally different memory representation and operations.

This should not be surprising: the same disruptions occurred in the mid-1980s when multi-instruction multiple-datastream (MIMD) computers became available. The languages of the day simply could not take advantage of the capabilities of the hardware. This was mainly due to the many different models of computing that have been developed. Throughout this entry we have only investigated the single von Neumann machine model. In the MIMD world, the interconnection among the individual machines eliminates any standard computing model. Who knows what quantum computing and beyond will bring.

For More Information

The Internet provides detailed information concerning available programming languages. Wikipedia has entries including languages organized by types that organize languages into 43 categories. Eric Levenez’s website contains a graph of 50 prominent languages over time and how they are related to one another.

REFERENCES

