Chapter 1: History of Programming Languages

The first programming languages were the machine and assembly languages of the earliest computers, beginning in the 1940s. Hundreds of programming languages and dialects have been developed since that time. Most have had a limited life span and utility, while a few have enjoyed widespread success in one or more application domains. Many have played an important role in influencing the design of future languages. Most new programming languages arise as a reaction to some language that the designer knows (and likes or dislikes) already, so one can propose a family tree or genealogy for programming languages, just as for living organisms. The following are some of the important events in the history of programming languages:

Ada Lovelace machine algorithm development in 1843

Ada Lovelace created the very first machine algorithm in 1843, which was the beginning of the invention of programming languages. She created this algorithm for the Difference Machine, which was a calculating machine that was constructed in the 1820s by the mathematician Charles Babbage.

Konrad Zuse created Plankalkul in early 1940s

In the early 1940s, Konrad Zuse created the programming language that was called Plankalkul. This was the very first high-level programming language developed for computers. It stored several codes that could be used repeatedly by engineers to perform certain operations routinely.

Assembly language and shortcode were first put into effect in 1949

In 1949, the Electronic Delay Storage Automatic Calculator was the first to use assembly language, which was considered to be a low-level computer programming language that broke down the complex language of machine codes. This means that the instructions to work a computer were put into more simplified terms.

Shortcode was first mentioned by John McCauley in early 1949, but it was not used until late 1949 and early 1950 by William Schmitt to benefit the procedures of the BINAC computer as well as the UNIVAC.

Autocode was the first compiled programming language to be used in 1952

In 1952, Alick Glennie created the term Autocode which means “a family of programming languages.” He was a computer scientist who initially used autocode for the Mark 1 computer. It was the very first compiled programming language to be used to translate machine code through a special program called the compiler.

John Backus developed FORmula TRANslation (FORTRAN) in 1957

In 1957, John Backus implemented the FORmula TRANslation (FORTRAN), which is the oldest computer programming language that is still being used today around the nation. The FORTRAN was developed to solve mathematical, statistical, and scientific problems.

Algorithmic language (ALGOL) and List Processor (LISP) both created in 1958

Algorithmic language (ALGOL) and List Processor (LISP) were both created in 1958. ALGOL was developed by several European and American computer scientists and was considered to be the very beginning of many popular programming languages including C, Java, C++, and Pascal.

John McCarthy created LISP for artificial intelligence (AI) purposes. It is currently one of the oldest computer programming languages still being used. Many individuals and businesses are continuing to use LISP instead of other programs such as Python or Ruby.

Common Business Oriented Language (COBOL) was invented by Dr. Grace Murray Hopper in 1959

The Common Business Oriented Language (COBOL) was first invented in 1959 by Dr. Grace Murray Hopper. This was a huge milestone for the history of programming languages as it is the computer programming language behind ATMs, cellular device and landline telephones, traffic signals, credit card processors, and so much more. COBOL was put into place to run on all computer types and is still regularly used by bank systems today.

Beginner's All-Purpose Symbolic Instruction Code (BASIC) was created in 1964

Students attending Dartmouth College created the Beginner's All-Purpose Symbolic Instruction Code (BASIC) in 1964. Its purpose was to help students who did not have much comprehension of computers or math. Later on, Paul Allen and Bill Gates worked further on this programming language, and it became Microsoft's first sold product.

Simula 1965

Simula is considered the first ever object-oriented programming language, developed around 1965 at the Norwegian Computing Centre in Oslo, by two Norwegian computer scientists Ole-Johan Dahl and Kristen Nygaard. It was operating on the UNIVAC 1107.

PASCAL was implemented in 1970 by Niklaus Wirth

Niklaus Wirth implemented PASCAL and named it after the famous mathematician Blaise Pascal. It was originally created to help individuals learn how to use programming languages, and Apple was one of the top companies supporting it due to its simplified and easy process.

Smalltalk, C, and SQL all originated in 1972

Alan Kay, Dan Ingalls, and Adele Goldberg designed Smalltalk to help computer programmers alter programming languages. This eventually led to the creation of several programs such as Java, Ruby, and Python.

C was created by Dennis Ritchie to use solely with the operating system called Unix. The reason it was given the name "C" is because it was a newer programming language that derived from the language "B.". Apple, Google, and Facebook are a few of the top tech companies that are still using the C programming language.

Structured Query Language (SQL) was first generated by Donald Chamberlain and Raymond Boyce to change and view important data that was stored on computers. Many businesses still use SQL today including Accenture and Microsoft.

Jean Ichbiah led the creation of Ada in the early 1980s

Jean Ichbiah led the construction of Ada, which was named after the first person to ever use programming languages, Ada Lovelace. Ada is a high-level computer programming language and is commonly used to control air traffic in various countries such as Belgium, Germany, and Australia.

C++ and Objective-C both were produced in 1983

In 1983, the C programming language was altered and transformed to become C++ by Bjarne Stroustrup. C++ had new features that C was missing, such as templates, classes, and virtual tasks. C++ is known to be one of the best programming languages and even won an award in 2003. Many programs use C++ including Microsoft Office, gaming platforms, Adobe Photoshop, and so much more.

Objective-C was produced in 1983 by Tom Love and Brad Cox and is the leading computer programming language used for all Apple systems including iOs and macOS.

Larry Wall launched Perl in 1987

Perl was launched in 1987 by Larry Wall. It originally was created to edit text, but it is now mostly used for a variety of different purposes within database applications, graphic programs, network programs, and more.

Haskell was formed in 1990

Haskell was formed and named after the famous mathematician Haskell Brooks Curry in 1990. This programming language is typically used for mathematical procedures. Many businesses and organizations use Haskell for different reasons, but it is also often used to create video games.

Visual Basic and Python were both established in 1991

Microsoft constructed Visual Basic in 1991 to help users drag and drop codes through a geographical user interface. By using Visual Basic, individuals and companies are able to easily select and change a large set of codes at once.

Python is one of the most well-known computer programming languages. It was established in 1991 by Guido Van Rossum to provide support to various programming styles. Yahoo, Spotify, and Google are just a few of the companies still widely using this program to this day.

Yukihiro Matsumoto developed Ruby in 1993

Ruby was developed by Yukihiro Matsumoto in 1993. Many programs inspired the creation of Ruby, such as Perl, Smalltalk, Lisp, Ada, and many more. Users who use Perl usually have an objective to develop certain web applications. A few of the companies that use Ruby often are Hulu, Groupon, and Twitter.

Java, JavaScript, and PHP were first introduced in 1995

James Gosling created Java for the purpose of an interactive television project. It is one of most favoured programming languages still being constantly used today on cellular devices and computers.

Brendan Eich developed JavaScript in 1995 to help individuals with creating webpages, browsers, widgets, and PDF documents. Majority of websites that you access use JavaScript such as Mozilla Firefox, Gmail, and Adobe Photoshop.

Hypertext Preprocessor (PHP) was first introduced in 1995 as Personal Homepage. Its goal is to help individuals and businesses build and update their websites. Several companies still depend on PHP such as Wikipedia, WordPress, Facebook, and many more.

C# came about in the year 2000

In 2000, C# was put into effect by Microsoft with the purpose of merging the computing features of C++ with Visual Basic's simplified features. C# is described as being similar to Java even though it derived from C++ and Visual Basic. All Microsoft tools and products currently use C#.

Scala and Groovy were formed in 2003

In 2003, Martin Odersky produced Scala, a mathematical programming language. It is compatible with Java, which is essential to Android's system development. Some of the companies still using Scala are LinkedIn, Netflix, Foursquare, and Twitter.

Bob McWhirter and James Strachan developed Groovy, which is a programming language that originated from Java. The purpose of Groovy is to improve efficiency and production. Starbucks and Craftbase are a couple of the well-known companies using Groovy to complete their daily tasks.

Google launched Go in 2009

In 2009, Google launched Go to solve and handle any issues that may occur when dealing with larger software systems. It consists of a very modern and easy-to-use structure, which has made it popular for companies and organizations throughout the globe to adopt. Some of these include Uber, Google, Twitch, and more.

Dart by Google in 2011

Dart is an open source web-based programming language developed by Google. Introduced in October 2011, Dart was created in the hopes that it would one day replace JavaScript as the primary programming language for web applications. It has a syntax similar to the C programming language, and supports object-oriented constructs such as classes and inheritance.

Swift was established by Apple in 2014

In 2014, Apple implemented Swift to replace Objective-C, C++, and C. The main goal was to create a new programming language that is easier and simpler than Objective-C, C++, and C. Swift offers users a high level of versatility, which gives it the ability to be used on cellular devices, cloud applications, and on desktop computers.

There are many more languages, in particular if one counts more domain-specific languages such as Matlab, SAS and R, and strange “languages” such as spreadsheets.

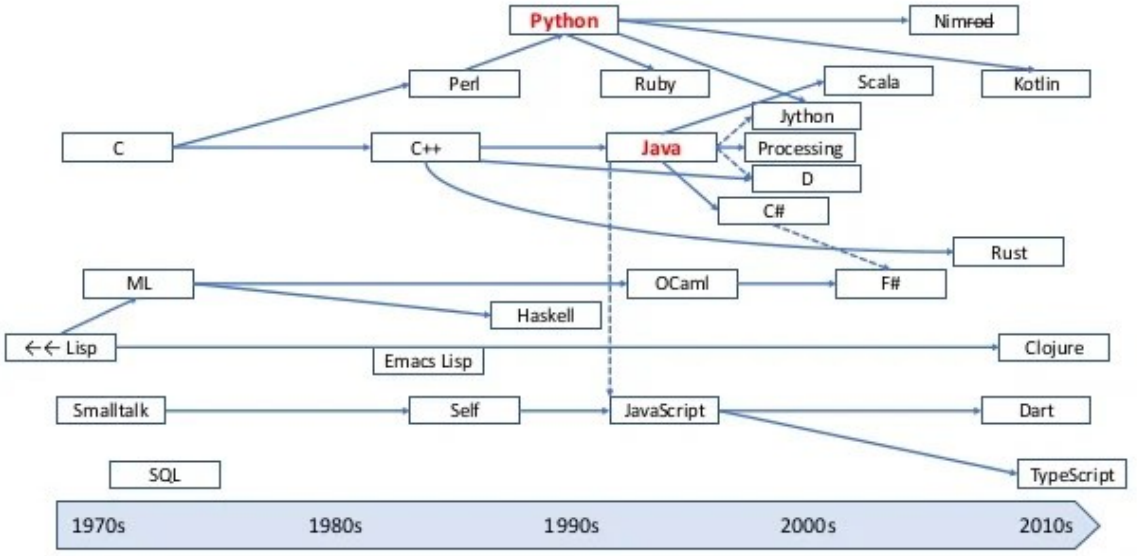

Figure 1.1: Genealogy of some Programming languages

In general, languages left side are closer to the real hardware than those on the right side, which are more “high-level” in some sense. In C, it is fairly easy to predict what instructions and how many instructions will be executed at run-time for a given line of program.

Some of the old programming languages have been put to rest, but there are many still being used today. In fact, FORTRAN has been around since 1953, and the newest version was revealed in 2018. It was created at IBM and has been updated and modified for use on the latest high-speed computers. COBOL is another programming language that is still being widely used. It was developed in 1959, and banks rely on it heavily to this day for in-person transactions as well as ATM transactions.

Chapter 2: Reasons for Studying Concepts of Programming Languages

It is natural for students to wonder how they will benefit from the study of programming language concepts. After all, many other topics in computer science are worthy of serious study. In fact, many now believe that there are more important areas of computing for study than can be covered in a five-year university curriculum. The following are compelling list of potential benefits of studying concepts of programming languages:

1. **Increased ability to express ideas/algorithms:** In Natural language, the depth at which people think is influenced by the expressive power of the language they use. In programming language, the complexity of the algorithms that people Implement is influenced by the set of constructs available in the programming language. Programmers, in the process of developing software, are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited. Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs.

2. **Improved background for choosing appropriate Languages:** Many programmers use the language with which they are most familiar, even though poorly suited for their new project. It is ideal to use the most appropriate language. If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem. Some of the features of one language often can be simulated in another language. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

3. **Increased ability to learn new languages:** For instance, knowing the concept s of object-oriented programming OOP makes learning Python significantly easier than those who have never used those concepts and also, knowing the grammar of one's native language makes it easier to learn a second language. Furthermore, learning a second language has the benefit of teaching you more about your first/native language.

4. **Better Understanding of Significance of implementation:** In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts. In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices. Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issue is that it

allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

5. **Better use of languages that are already known:** Most contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

6. **The overall advancement of computing:** Finally, there is a global view of computing that can justify the study of programming language concepts. Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular languages are not always the best available. In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts. For example, many people believe it would have been better if ALGOL 60 had displaced FORTRAN in the early 1960s, because it was more elegant and had much better control statements, among other reasons. That it did not, is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60. They found its description difficult to read (which it was) and even more difficult to understand. They did not appreciate the benefits of block structure, recursion, and well-structured control statements, so they failed to see the benefits of ALGOL 60 over FORTRAN. The fact that computer users were generally unaware of the benefits of the language played a significant role.

Chapter 3: Categories of Programming Languages

If we want to categorize programming languages, we need to look at the look and feel of the language, its execution model, or the kind of programming paradigms most naturally supported. Technical aspects of languages will consider linguistic structure, expressive features, possibility of efficient implementation, direct support for certain programming models, and similar concerns. Some examples:

- *Machine languages*, that are interpreted directly in hardware
- *Assembly languages*, that are thin wrappers over a corresponding machine language
- *High-level languages*, that are anything machine-independent
- *System languages*, that are designed for writing low-level tasks, like memory and process management
- *Scripting languages*, that are generally extremely high-level and powerful
- *Domain-specific languages*, that are used in highly special-purpose areas only
- *Visual languages*, that are non-text based
- *Esoteric languages*, that are not really intended to be used, but are very interesting, funny, or educational in some way

These types are not mutually exclusive: Perl is both high-level and scripting; C is considered both high-level and system. Some languages are partially visual, but you get to type bits of code into little boxes.

Machine Languages

Machine language is the direct representation of the code and data run directly by a computing device. Machine languages feature:

- Registers to store values and intermediate results
- Very low-level machine instructions (add, sub, div, sqrt) which operate on these registers and/or memory
- Labels and conditional jumps to express control flow
- A lack of memory management support — programmers do that themselves

The machine instructions are carried out in the hardware of the machine, so machine code is by definition machine-dependent. Different machines have different instruction sets. The instructions and their operands are all just bits.

Machine code is usually written in hex. Here's an example for the Intel 64 architecture:

```
89 F8 A9 01 00 00 00 75 06 6B C0 03 FF C0 C3 C1 E0 02 83 E8 03 C3
```

Assembly Languages

An assembly language is an encoding of machine code into something more readable. It assigns human-readable labels (or names) to storage locations, jump targets, and subroutine starting addresses, but doesn't really go too far beyond that. It's really isomorphic to its machine language. Here's the function from above on the Intel 64 architecture using the GAS assembly language:

```
        .globl  f
        .text
f:
        mov     %edi, %eax      # Put first parameter into eax register
        test    $1, %eax        # Examine least significant bit
        jnz     odd             # If it's not a zero, jump to odd
        imul    $3, %eax        # It's even, so multiply it by 3
        inc     %eax            # and add 1
        ret                     # and return it
odd:
        shl    $2, %eax         # It's odd, so multiply by 4
        sub    $3, %eax         # and subtract 3
        ret                     # and return it
```

And here's the same function, written for the SPARC:

```
        .global f
f:
        andcc   %o0, 1, %g0
        bne     .L1
        sll     %o0, 2, %g2
        sll     %o0, 1, %g2
        add     %g2, %o0, %g2
        b       .L2
        add     %g2, 1, %o0
.L1:
        add     %g2, -3, %o0
.L2:
        retl
        nop
```

High-Level Languages

A high-level language gets away from all the constraints of a particular machine. HLLs may have features such as:

- Names for almost everything: variables, types, subroutines, constants, modules
- Complex expressions (e.g. 2 * (y) >= 88 && sqrt(4.8) / 2 % 3 == 9)
- Control structures (conditionals, switches, loops)

- Composite types (arrays, structs)
- Type declarations
- Type checking
- Easy, often implicit, ways to manage global, local and heap storage
- Subroutines with their own private scope
- Abstract data types, modules, packages, classes
- Exceptions

The previous example looks like this in Fortran 90:

```
integer function f (n)
    implicit none
    integer, intent(in) :: n
    if (mod(n, 2) == 0) then
        f = 3 * n + 1
    else
        f = 4 * n - 3
    end if
end function f
```

and like this in C and C++:

```
int f(const int n) {
    return (n % 2 == 0) ? 3 * n + 1 : 4 * n - 3;
}
```

and like this in Java and C#:

```
class ThingThatHoldsTheFunctionUsedInTheExampleOnThisPage {
    public static int f(int n) {
        return (n % 2 == 0) ? 3 * n + 1 : 4 * n - 3;
    }
}
```

and like this in JavaScript:

```
function f(n) {
  return (n % 2 === 0) ? 3 * n + 1 : 4 * n - 3;
}
```

and like this in Prolog:

```
f(N, X) :- 0 is mod(N, 2), X is 3 * N + 1.
f(N, X) :- 1 is mod(N, 2), X is 4 * N - 3.
```

and like this in Python:

```
def f(n):
    return 3 * n + 1 if n % 2 == 0 else 4 * n - 3
```

System Languages

System programming languages differ from *application programming languages* in that they are more concerned with managing a computer system rather than solving general problems in health care, game playing, or finance. In a system language, the programmer, not the runtime system, is generally responsible for:

- Memory management
- Process management
- Data transfer
- Caches
- Device drivers
- Directly interfacing with the operating system

Scripting Languages

Scripting languages are used for wiring together systems and applications at a very high level. They are almost always extremely expressive (they do a lot with very little code) and usually dynamic (meaning the compiler does very little, while the run-time system does almost everything).

Esoteric Languages

An esoteric language is one not intended to be taken seriously. They can be jokes, near-minimalistic, or despotic (purposely obfuscated or non-deterministic).

Chapter 4: Programming Paradigms

Very often a programming language is created to help people program in a certain way. A programming paradigm is a style, or "way," of programming. Some languages make it easy to write in some paradigms but not others. A paradigm is a way of doing something (like programming), not a concrete thing (like a language). Now, it's true that if a programming language L happens to make a particular programming paradigm P easy to express, then we often say "L is a P language" (e.g. "Haskell is a functional programming language") but that does not mean there is any such thing as a "functional language paradigm".

You should know these common paradigms:

- ***Imperative*:** Programming with an explicit sequence of commands that update state.
- ***Declarative***: Programming by specifying the result you want, not how to get it.
- ***Structured***: Programming with clean, goto-free, nested control structures.
- ***Procedural***: Imperative programming with procedure calls.
- ***Functional*** (Applicative): Programming with function calls that avoid any global state.
- ***Function-Level*** (Combinator): Programming with no variables at all.
- ***Object-Oriented*:** Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
  - Class-based: Objects get state and behaviour based on membership in a class.
  - Prototype-based: Objects get behaviour from a prototype object.
- *Event-Driven*: Programming with emitters and listeners of asynchronous actions.
- *Flow-Driven*: Programming processes communicating with each other over predefined channels.
- *Logic* (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.
- *Constraint*: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
- *Aspect-Oriented*: Programming cross-cutting concerns applied transparently.
- *Reflective*: Programming by manipulating the program elements themselves.
- *Array*: Programming with powerful array operators that usually make loops unnecessary.

Paradigms are not meant to be mutually exclusive; a single program can feature multiple paradigms! Here is an overview of some of the major paradigms:

Imperative Programming

Control flow in *imperative programming* is *explicit*: commands show *how* the computation takes place, step by step. Each step affects the global **state** of the computation.

```
    result = []
    i = 0
start:
    numPeople = length(people)
    if i >= numPeople goto finished
    p = people[i]
    nameLength = length(p.name)
    if nameLength <= 5 goto nextOne
    upperName = toUpper(p.name)
    addToList(result, upperName)
nextOne:
    i = i + 1
    goto start
finished:
    return sort(result)
```

Structured Programming

Structured programming is a kind of imperative programming where control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

```
result = [];
for i = 0; i < length(people); i++ {
    p = people[i];
    if length(p.name)) > 5 {
        addToList(result, toUpper(p.name));
    }
}
return sort(result);
```

Early languages emphasizing structured programming: Algol 60, PL/I, Algol 68, Pascal, C, Ada 83, Modula, Modula-2. Structured programming as a discipline is sometimes though to have been started by a famous letter by Edsger Dijkstra entitled Go to Statement Considered Harmful.

Object Oriented Programming

OOP is based on the sending of *messages* to objects. Objects respond to messages by performing operations, generally called *methods*. Messages can have arguments. A society of objects, each with their own local memory and own set of operations has a different feel than the monolithic processor and single shared memory feel of non-object oriented languages.

One of the more visible aspects of the more pure-ish OO languages is that conditionals and loops become messages themselves, whose arguments are often blocks of executable code. In a Smalltalk-like syntax:

```
result := List new.
people each: [:p |
  p name length greaterThan: 5 ifTrue: [result add (p name upper)]
]
result sort.
^result
```

This can be shortened to:

```
^people filter: [:p | p name length greaterThan: 5] map: [:p | p name upper]
sort
```

Many popular languages that call themselves OO languages (e.g., Java, C++), really just take some elements of OOP and mix them in to imperative-looking code. In the following, we can see that `length` and `toUpper` are methods rather than top-level functions, but the `for` and `if` are back to being control structures:

```
result = []
for p in people {
    if p.name.length > 5 {
        result.add(p.name.toUpper);
    }
}
return result.sort;
```

The first object oriented language was Simula-67; Smalltalk followed soon after as the first "pure" object-oriented language. Many languages designed from the 1980s to the present have labelled themselves object-oriented, notably C++, CLOS (object system of Common Lisp), Eiffel, Modula-3, Ada 95, Java, C#, Ruby.

Exercise: Do some research into the way the term "object-oriented" has evolved and whether or not there is some controversy around its meaning. You might want to start your research with the famous Alan Kay quote "I made up the term 'object-oriented', and I can tell you I didn't have C++ in mind." What *did* he have in mind? Does he care that deeply about the way C++ and Java deviated from the intended meaning?

Declarative Programming

Control flow in *declarative programming* is *implicit*: the programmer states only *what* the result should look like, **not** how to obtain it.

```
select upper(name)
from people
where length(name) > 5
order by name
```

No loops, no assignments, etc. Whatever engine that interprets this code is just supposed go get the desired information, and can use whatever approach it wants. (The logic and constraint paradigms are generally declarative as well.)

Functional Programming

In *functional programming*, control flow is expressed by combining function calls, rather than by assigning values to variables:

```
sort(
  fix(f => p =>
    ifThenElse(equals(p, emptylist),
      emptylist,
      ifThenElse(greater(length(name(head(p))), 5),
        append(to_upper(name(head(p))), f(tail(p))),
        f(tail(people)))))(people))
```

Yikes! We'll describe that later. For now, be thankful there's usually syntactic sugar:

```
let
    fun uppercasedLongNames [] = []
      | uppercasedLongNames (p :: ps) =
          if length(name p) > 5 then (to_upper(name p))::(uppercasedLongNames
ps)
          else (uppercasedLongNames ps)
in
    sort(uppercasedLongNames(people))
```

Huh? That still isn't very pretty. Why do people like this stuff? Well the real power of this paradigm comes from passing functions to functions (and returning functions from functions).

```
sort(
    filter(s => length s > 5,
        map(p => to_upper(name p),
            people)))
```

We can do better by using the cool `|>` operator. Here `x |> f` just means `f(x)`. The operator has very low precedence so you can read things left-to-right:

```
people |> map (p => to_upper (name p)) |> filter (s => length s > 5) |> sort
```

Let's keep going! Notice that you wouldn't write `map(s => square(x))`, right? You would write `map(square)`. We can do something similar above, but we have to use function composition, you know, `(f o g)x` is `f(g(x))`, so:

```
people |> map (to_upper o name) |> filter (s => length s > 5) |> sort
```

With functional programming:

- There are no commands, only side-effect free expressions
- Code is much shorter, less error-prone, and much easier to prove correct
- There is more inherent parallelism, so good compilers can produce faster code

Some people like to say:

- *Functional, or Applicative, programming* is programming without assignment statements: one just applies functions to arguments. Examples: Scheme, Haskell, Miranda, ML.
- *Function-level programming* does away with the variables; one combines functions with *functionals*, a.k.a. *combinators*. Examples: FP, FL, J.

Logic and Constraint Programming

Logic programming and *constraint programming* are two paradigms in which programs are built by setting up relations that specify *facts* and inference *rules*, and asking whether or not something is true (i.e. specifying a *goal*.) Unification and backtracking to find solutions (i.e.. satisfy goals) takes place automatically. Languages that emphasize this paradigm: Prolog, GHC, Parlog, Vulcan, Polka, Mercury, Fnil.

Parallel Processing

Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posse many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.

It has the advantage of solving larger problems in a short point of time. It is better suited for modelling, simulating and understanding complex, real-world phenomena. However, power consumption is huge by the multi-core architectures and costs can sometimes be quite large.

Database/Data driven programming approach

This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example, SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So, it has its own wide application. Below is a sample SQL code for creating a database table.

```
CREATE DATABASE databaseAddress;
CREATE TABLE Addr (
PersonID int,
LastName varchar(200),
FirstName varchar(200),
Address varchar(200),
City varchar(200),
State varchar(200)
);
```

Languages and Paradigms

One of the characteristics of a language is its support for particular programming paradigms. For example, Smalltalk has direct support for programming in the object-oriented way, so it might be called an object-oriented language. OCaml, Lisp, Scheme, and JavaScript programs tend to make heavy use of passing functions around so they are called "functional languages" despite having variables and many imperative constructs.

There are two very important observations here:

- Very few languages implement a paradigm 100%. When they do, they are *pure*. It is incredibly rare to have a "pure OOP" language or a "pure functional" language. A lot of languages have a few escapes; for example in OCaml, you will program with functions 90% or more of the time, but if you need state, you can get it. Another example: very few languages implement OOP the way Alan Kay envisioned it.

  > "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them."

- A lot of languages will facilitate programming in one or more paradigms. In Scala you can do imperative, object-oriented, and functional programming quite easily. If a language is *purposely* designed to allow programming in many paradigms is called a *multi-paradigm language*. If a language only *accidentally* supports multiple paradigms, we don't have a special word for that.

Chapter 5: Programming Domains

Computers have been applied to a myriad of different areas, from controlling nuclear power plants to providing video games in mobile phones. Because of this great diversity in computer use, programming languages with very different goals have been developed. We will briefly discuss a few of the most common areas of computer applications and their associated languages.

1. Scientific Applications
2. Data processing Applications
3. Text processing Applications
4. Artificial intelligence Applications
5. Systems Programming Applications
6. Web software
7. Mobile Applications

Scientific Applications: These are Applications which predominantly manipulate numbers and arrays of numbers, using mathematical and statistical principles as a basis for the algorithms. These algorithms encompass such problem as statistical significance test, linear programming, regression analysis and numerical approximations for the solution of differential and integral equations. Examples are; FORTRAN, Pascal, Math lab.

Data processing Applications: These are programming problems whose predominant interest is in the creation, maintenance, extraction and summarization of data in records and files. Examples are; COBOL is a programming language that can be used for data processing applications.

Text processing Applications: These are programming whose principal activity involves the manipulation of natural language text, rather than numbers as their data. SNOBOL and C language have strong text processing capabilities.

Artificial Intelligence Applications: Those programs which are designed principally to emulate intelligent behavior. They include game playing algorithms such as chess, natural language understanding programs, computer vision, robotics and expert systems. Examples are;

- LISP has been the predominant AI programming language,
- PROLOG using the principle of ‘’Logic programming’’
- Lately AI applications are written in Java, C++ and python.

Systems Programming Applications: These involve developing those programs that interface the computer system (the hardware) with the programmer and the operator. These programs include compilers, assembles, interpreters, input-output routines, program management facilities and schedules for utilizing and serving the various resources that comprise the system. Examples are; Ada and Modula – 2 are examples of programming languages and C.

Web software: The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose

programming languages, such as Java. Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation. This functionality can be provided by embedding programming code in an HTML document. Such code is often in the form of a scripting language, such as JavaScript or PHP. There is also some markup like languages that have been extended to include constructs that control document processing.

Mobile Applications: With the growing number of smart mobile devices, there are programming languages specifically designed for mobile application development. Some of these languages are Apple's swift, flutter, react-native etc.

Chapter 6: Language Evaluation and Design

Criteria for Language Evaluation and Comparison

1. **Expressivity:** The ability of a language to clearly reflect the meaning intended by the algorithm designer (the programmer). An "expressive" language permits an utterance to be compactly stated, and encourages the use of statement forms associated with structured programming (usually "while "loops and "if – then – else" statements).

2. **Well – Defined:** The language's syntax and semantics are free of ambiguity, are internally consistent and complete. Thus, the implementer of a well-defined language should have, within its definition a complete specification of all the language's expressive forms and their meanings. The programmer, by the same virtue should be able to predict exactly the behavior of each expression before it is actually executed.

3. **Data types and structures**: The ability of a language to support a variety of data values (integers, real, strings, pointers etc.) and non-elementary collect ions of these.

4. **Modularity:** Modularity has two aspects: the language's support for sub-programming and the language's extensibility in the sense of allowing programmer – defined operators and data types. By sub programming, we mean the ability to define independent procedures and functions (subprograms), and communicate via parameters or global variables with the invoking program.

5. **Input-Output facilities:** In evaluating a language's "Input-Output facilities" we are looking at its support for sequential, indexed, and random-access files, as well as its support for database and information retrieval functions

6. **Portability:** A language which has "portability" is one which is implemented on a variety of computers. That is, its design is relatively "machine – independent". Languages which are well- defined tend to be more portable than others.

7. **Efficiency:** An "efficient" language is one which permits fast compilation and execution on the machines where it is implemented. Traditionally, FORTRAN and COBOL have been relatively efficient languages in their respective application areas.

8. **Pedagogy:** Some languages have better "pedagogy" than others. That is, they are intrinsically easier to teach and to learn, they have better textbooks; they are implemented in a better program development environment, they are widely known and used by the best programmers in an application area.

9. **Generality:** This means that a language is useful in a wide range of programming applications. For instance, APL has been used in mathematical applications involving matrix algebra and in business applications as well.

Influences on Programming Language Design

Computer Architecture: Languages are developed around the prevalent computer architecture, known as the von Neumann architecture.

- Data and programs stored in memory
- Memory is separate from CPU
- Instructions and data are piped from memory to CPU

Imperative languages, most dominant, because:

- Variables model memory cells
- Assignment statements model piping
- Iteration is efficient

The connection speed between a computer's memory and its processor determines the speed of that computer. Program instructions often can be executed much faster than the speed of the connection; the connection speed thus, results in a bottleneck (Von Neumann bottleneck). It is the primary limiting factor in the speed of computers

Programming Methodologies: New software development methodologies (e.g. object-oriented software development) led to new programming paradigms and by extension, new programming languages

.

Trade-offs in Language Design

1. **Reliability Vs. Cost of Execution:** For example, Java demands that all references to array elements be checked for proper indexing, which leads to increased execution costs.

2. **Readability vs. Writability:** APL provides many powerful operators land a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.

3. **Writability (Flexibility) vs. reliability:** The pointers in C++ for instance are powerful and very flexible but are unreliable.

Implementation Methods

Compilation: Programs are translated into machine Language & System calls. Translated high level program (source language) into machine code (machine language)

- Slow translation, fast execution
- Compilation process has several phases
- Lexical analysis converts characters in the source program into lexical units (e.g. identifiers, operators, keywords).
- Syntactic analysis: transforms lexical units into parse trees which represent the syntactic structure of the program.

- Semantics analysis check for errors hard to detect during syntactic analysis; generate intermediate code.
- Code generation – Machine code is generated
- Source program translated ("compiled") to another language

Interpretation: Programs are interpreted by another program (an interpreter).

- Easier implementation of programs (run-time errors can easily and immediately be displayed).
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more memory space and is now rare3 for traditional high-level languages.
- Significant comeback with some Web scripting languages like PHP and JavaScript.
- Interpreters usually implement as a read-eval-print loop:
- Interpreters act as a virtual machine for the source language:
- Interpreter executes each instruction in source program one step at a time, No separate executable.

Hybrid: Programs translated into an intermediate language for easy interpretation.

- This involves a compromise between compilers and pure interpreters. A high-level program is translated to an intermediate language that allows easy interpretation.
- Hybrid implementation is faster than pure interpretation.
- Examples of the implementation occur in Perl and Java.
- Perl programs are partially compiled to detect errors before interpretation.
- Initial implementations of Java were hybrid. The intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a run time system (together, these are called Java Virtual Machine).

Just –in-time: Hybrid implementation, then compile sub programs code the first time they are called.

- This implementation initially translates programs to an intermediate language then compile the intermediate language of the subprograms into machine code when they are called.
- Machine code version is kept for subsequent calls. Just-in-time systems are widely used for Java programs. Also, .NET languages are implemented with a JIT system.