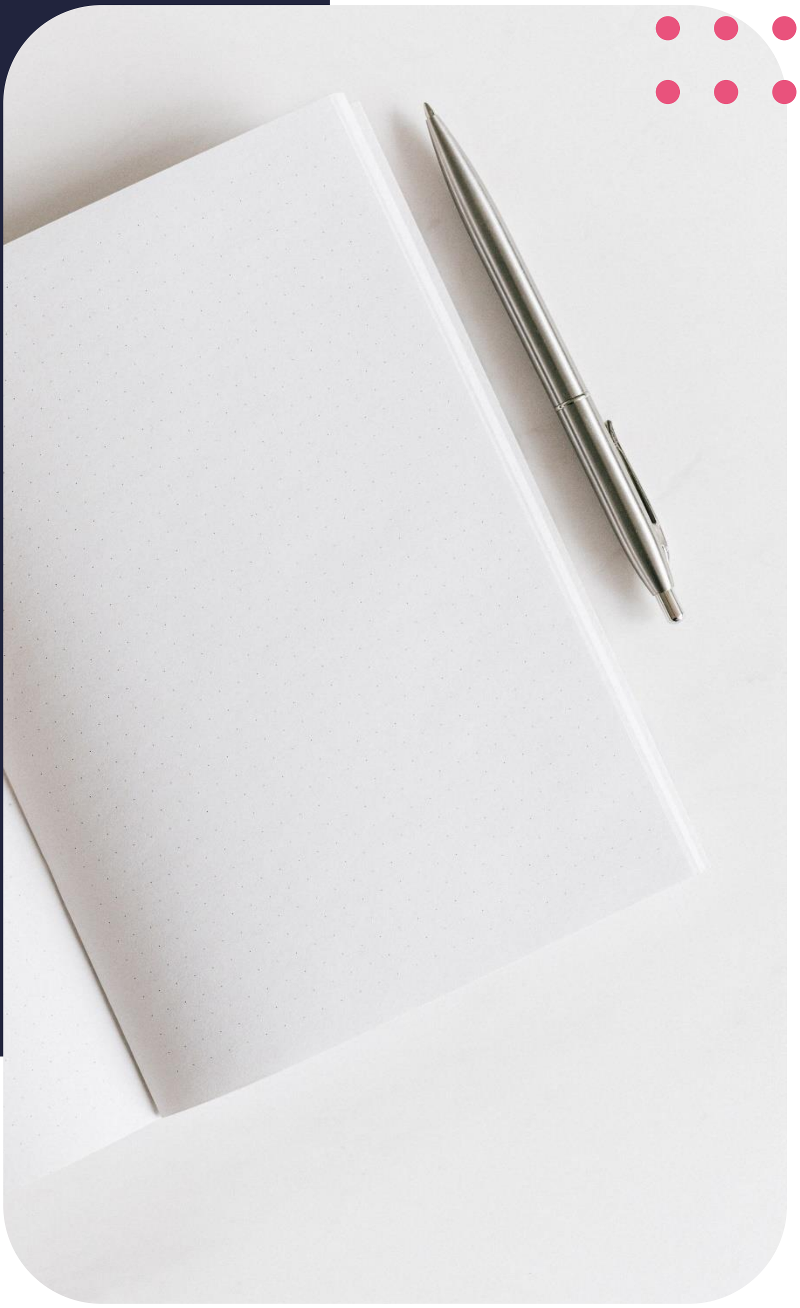
Diploma in Computer Science

Language construction

Contents

3	Introduction
3	Lesson outcomes
3	Natural human language
4	Computer language
6	Programming language theory
8	The Chomsky Hierarchy
9	Conclusion
8	References



Lesson outcomes

By the end of this lesson, you should be able to:

- Explain the difference between human natural language and computer language
- Discuss various aspects of computer language
- Understand the theory behind programming language and modelling using abstract machines

Introduction

In this lesson, we explore the language that computers speak, and how this relates to our human language. We will also examine how computer components communicate. Our final section of this lesson will focus on the theory of programming language and how the various abstract machines use different computer languages.

Human language

Natural language

In Computer Science, whenever you hear ‘natural language’, it is referring to human language. It’s always been a quest in Computer Science to make computers completely understand human language. And although we’ve made great strides, we are still not quite there yet. Computers still struggle with human language. While some computer scientists are working on making that dream come true, others are studying human language to make it easier for humans to write code for computers.

Human language has been around for as long as history has been recorded. This is how we communicate; from the time we learn to say our first word for the rest of our lives. Language forms the basis of most human interaction. But what exactly is language? The simplest way of describing it is “a system of spoken, manual or written symbols that humans use to express themselves, their identity, imagination and emotions. ” There are many different languages spoken around the world and their main function is communication.

The exact same principle applies to computers. Humans wanted to find a way to “talk” to computers; to create a language to communicate what they want them to do in a way that makes sense to both parties.

Language structure

Structure is common to all languages on Earth. This enables anyone who learns the language to speak to others and be understood. Language structure is primarily formed around syntax and semantics.

Syntax refers to the set of rules that tells us how to arrange words and combine phrases. **Semantics** refers to the meaning of a word or phrase and its connection to a certain concept. In other words, syntax is the order of words and spelling and semantics is grammar.

For example, the word “purr” is something that only a cat can do. Put in a sentence, you would say: “I heard my cat purr when I walked in.” The syntax and semantics work together to create a fully functioning statement.

You wouldn’t say: “I heard my carpet purr.” , for example. Why? Because the syntax and semantics simply don’t work well together to create a fully functioning statement in that sentence.

Understanding how languages work is important to anyone wanting to take up programming. Not only does it help you understand program flow, but languages are continually being studied and refined to improve computing fields such as natural language processing and voice synthesis.

Computer language

In ‘computer language’, people can’t change voice tone or facial expression as they speak to indicate a different meaning. In computer language, every sentence has to mean just one thing. This is generally how code is judged. If you look at a code and you can quickly get an idea of what it means, then it’s properly written code. If you look at a code and start wondering what on earth is going on, then that code was probably not well written.

Basically, computer code is a bunch of instructions, and naturally, instructions need to be as clear as possible. Imagine if someone gave you the following instructions: “Catch a plane to the continent, buy some items and come back”. That is all really confusing. Which plane should you catch? To which continent? What exactly must you buy? It would be a lot clearer if the person said: “Catch a plane to Asia and find a place to buy Himalayan salt before you return.”

Computer language, or what is more commonly known as ‘programming language’ is a lot simpler than human language. There isn’t much grammar to speak of, just syntax (remember, that’s words and phrases) because the statements need to be as clear as possible When computers with stored first became available, people started developing all sorts of new programming languages. In those days, programming languages were nothing more than a bunch of ones and zeros. Programming requires a lot of expertise and, as you can imagine, many errors occurred. For example, can you imagine making sense out of a code that reads 1010101111101010100000! Your head would be spinning by the time you wrote the 4th line!

This pushed scientists to come up with better ways to write programs for computers.

The binary system

By now you know that the binary system forms the basis of all computer operations. Computers use binary codes to represent characters, images, and sound – to communicate in their own language.

ASCII

A well-known binary code is ASCII, which stands for the American standard code for information interchange. Here, a 7-bit binary code is used to represent text and other characters within computers and communications equipment. Each letter or symbol is assigned a number from 0 to 127. Your computer keyboard has all 128 representations, because in computing, even zero is counted! This is a very important point; you will notice that even in your programs, counting starts at zero.

The ASCII number system assigns a 7-digit binary number to each character, for example, the number zero is represented as 0000000, the number 1 is 0000001 and so on. The very last number is 1111111, representing the delete button.

Unicode

ASCII works well if you speak English. All the letters of the alphabet and a good number of special characters are represented. But what if you speak Chinese? This is a problem for the computer because we can only represent 128 characters. This is where Unicode comes in.
Unicode comes in 4 forms: UTF-7, UTF-8, UTF-16, and UTF-32. (UTF stands for Unicode Transformation Format).

- UTF-7 was designed to represent ASCII characters in email messages that require Unicode encoding.
- UTF-8 uses 8 bits to maximise compatibility with ASCII, but when things get ugly, it can also allow for variable-width encoding expanding to 16, 24, 32, 40 or 48 bits to deal with larger sets of additional characters. It is the most popular type of Unicode encoding, using 8 bits for English characters, 16 bits for Latin and Middle Eastern characters and 24 bits for Asian characters.
- UTF-16 is less flexible. It starts at 16 bits with the ability to extend to 32 bits. It can represent up to one million characters.
- UTF-32 is as rigid as ASCII in that it can represent a whole lot of characters.

To find out how many characters a particular binary code can hold, you raise 2 to the number of bits that the binary code uses. For example, ASCII uses 7 bits, so 2^7gives us 128. Similarly, for UTF-32, 2^32 gives us a whopping 4.2 billion characters!

Parity

Parity is an important part of programming. What is it and why is it necessary? Parity is a technique that checks whether data has been lost or written over when it is moved or transmitted.

We said earlier that ASCII uses only 7 bits. But we also said that things should be in powers of 2 to avoid trouble.

In the early days of programming, computers were not very reliable and computer scientists needed a way to check if the data was indeed what was requested. Even today, while computers are so much more accurate, we still need parity because we constantly transmit data over mediums that can distort it.

So, parity is used to check data integrity. It can be implemented as either odd parity or even parity.

In **odd parity**, if the number of bits with a value of one is an even number, the parity bit value is set to one to make the total number of ones in the set (including the parity bit) an odd number. If the number of bits with a value of one is odd, the parity bit value is set to zero, so that the total number of ones in the set (including the parity bit) remains an odd number.

In **even parity**, the number of bits with a value of one are counted. If that number is odd, the parity bit value is set to one to make the total number of ones in the set (including the parity bit) an even number. If the number of bits with a value of one is even, the parity bit value is set to zero, so that the total number of ones in the set (including the parity bit) remains an even number.

Computer instructions and interpretations

Computers use machine language to do the magic. Machine language is the actual 1s and 0s that pass through the machine's hardware. All the fancy programming languages that you can ever learn are ultimately converted to machine language so that the computer can understand what you want it to do.

Each processor has its own instructions that it can understand, called the instruction set. As a programmer, you'd be more concerned with the instruction set architecture. The instruction set architecture is the design of the computer from a programmer's perspective; that is the set of basic operations that it supports.

It also determines the maximum length of each type of instruction, and the format of each instruction. The instruction set architecture offers a level of abstraction from the combinational and sequence circuits that run things. The programmer will not need to know how the circuits are wired in order to program the computer.

When you buy a computer, it is often labelled x86 in the technical specifications. This refers to the CPU instruction set. It also typically refers to the word size of the processor – basically, the size of a unit of data in the CPU. This would mean that a 64-bit CPU has a 64-bit register.

In previous lessons, we learned that computers consist of the CPU, RAM, input/output devices. How do all these computer parts communicate with each other? You would definitely need a fixed size of bits to be considered a single unit of data. That's why computer scientists agreed to standardise this unit to be 32 bits or 64 bits (depending on the manufacturer's choice). It can also loosely define the amount of data the CPU manipulates in one go. You may have come across programs that cannot run on particular versions of an operating system. For example, 32-bit programs can be run on 64-bit computers, but 64-bit programs cannot be run on 32-bit computers.

In computer language, a language statement generally corresponds to a single processor instruction. Each instruction is a designated unique sequence of 1s and 0s that describes a physical operation that the computer must perform. These 1s and 0s actually tell the CPU which transistors to switch on and which transistors to switch off.

A computer instruction comprises three parts: the operation code (usually abbreviated as "op code"), the address field, and the mode field.

- The **op code** field specifies the operation to be performed. It tells the CPU what kind of job needs to be done. It selects the part of the CPU circuitry that will carry out the operation; different circuits do different things. The sequence of 1s and 0s in the op code are part of the instruction set and are unique to that operation. Another type of op code comes along with some data to be processed.

- The **address field** designates the memory address or register.
- The **mode field** specifies the way the operand of the address in effect is determined.

There are three types of operation codes: the memory reference instruction, the register reference instruction, and the input-output instruction. The CPU uses these to determine the type of instruction that is being executed. It is possible for computers to have instructions of different lengths containing a varying number of addresses. The number of address fields in the instruction format depends on the internal organisation of its registers.

Data is not always available in an ideal form for the computer to use. The computer cannot interpret many different types of signals like the human body does. So, we connect input devices known as **transducers** to a computer to convert environmental data to digital form. Transducers have a sensing component that converts environmental phenomena to electrical signals. Using a calibrated control circuit, the electrical signals are converted to digital form so that they are ready to be processed by the computer.

One of the most common transducers around is the microphone. In fact, that is just one of the electrical transducers in your smartphone. Your smartphone is loaded with them. Others include an accelerometer, hall sensor, electric motor, pressure sensor, earpiece, mouthpiece, the display, and a whole lot more!

Programming language theory

Programming language is basically special-purpose language that is used to give machines instructions.

As mentioned earlier, computers that existed before programming languages were extremely hard to work with and were invented to make machines easier to use by expressing their processes logically and taking away the necessity for tedious rewiring.

Programming languages are strikingly different from human languages because they follow strict rules. For a computer scientist, everything is in a language: getting out of bed, going for a walk, unlocking a door, making popcorn…. everything!

Before a computer is built, an abstract machine is drawn up to show the input, output, as well as the set of allowable instructions to turn input into output. The purpose of designing an abstract machine is to present a detailed and precise analysis of how the computer system will work.

Let’s now look at the three different types of abstract machines: the finite state machine, push down automata and Turing machine.

Finite state machine

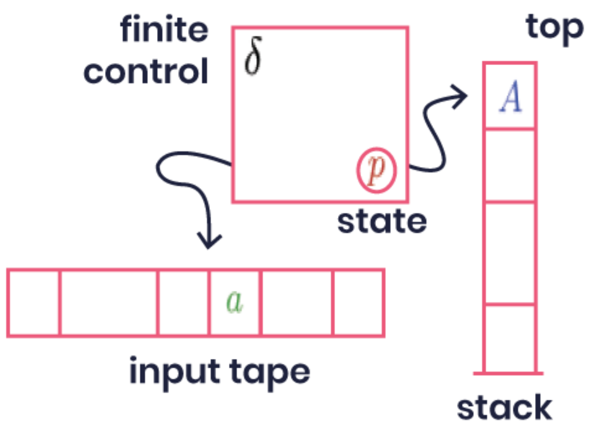
A finite state machine produces regular language to simulate sequential logic and some computer logic. A regular language is a language that can be expressed with a regular expression or a deterministic or non-deterministic finite state machine.

Basically, regular languages are one of the most important building blocks of programming language. They are one of the first things you will learn if ever you decide to design your own programming language. They are the subset of all strings languages that are used in computing models to describe problems. They are also used by computer scientists to group computability problems together. And they are useful in determining if a problem is solvable or not.

Push down automata

Push down automata are non-deterministic finite state machines that are hooked up with additional memory to form a stack. A stack is similar to a packet of biscuits. For you to get to the last biscuit, you have to start at the top. You can’t take a biscuit from the bottom without disturbing the stack. A computer stack works the same way. The last item to be placed on the stack is the first item to be removed.

Push down automata are called this because the elements are pushed down onto a stack. Because of this additional stack, they can do much more than a finite state machine. They accept context-free languages, which include the set of regular languages. This language describes strings that have matching parentheses.



The above diagram explains it clearly. The “input tape” contains the symbols that have matching parentheses. The machine will accept the input if all parentheses are matched. This is one of the theoretical concepts used to build parsers and compilers. Pushdown automata start with an empty stack. They then go through steps that either push (add) or pop (remove) items to the stack. The pushdown automation ends with a final state, that is, when all the input has been read.

Turing machine

A Turing machine is an abstract computational model that performs computations by reading and writing to infinite tape. Of all the models we will look at, this is by far the most powerful. That’s why it is often used as a computational model for solving problems in Computer Science. A Turing machine is also often used to test the limits of computation. Turing machines are similar to finite state machines except that they have infinite memory.

A common problem that a computer can solve, assuming it has enough memory, can also be solved using a Turing machine, and the reverse is also true. A Turing machine uses infinite tape as the memory, a tape head that reads the memory and a transition table that controls the behaviour of the machine. Each cell of the tape can have one of a predetermined finite set of symbols, one of which is the blank symbol.

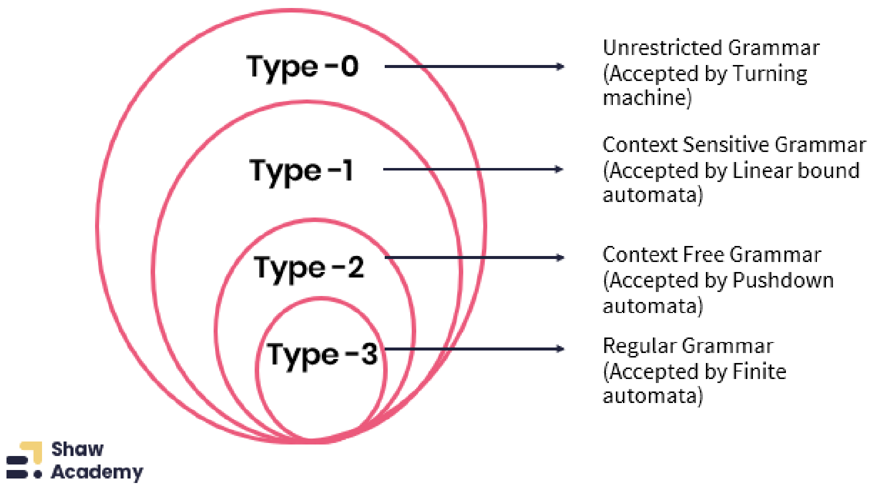
When the Turing machine starts, the head points to the first cell of the string, while all other cells are blank. For every step, the head uses instructions in the transition table and the symbol it is pointing at. It goes ahead with its next choice to either halt, write a symbol to the current cell, change its state, or move to the left or to the right. A Turing machine can either halt or run forever, based on the contents of the tape and the symbol table. A Turing machine generates unrestricted languages.

A Turing machine can simulate machines acting on different data in memory, and the model of programs being made up of many different lines of code. A Turing machine can also recognise many different kinds of languages and problems. A language is decidable when a Turing machine accepts only strings in the specified language and rejects those not in the language. Decidable languages are recognisable but recognisable languages are not decidable.

Turing machines are used extensively in modelling problems.

The Chomsky hierarchy

We have seen that different machines generate different types of computer languages. As we moved along, you may have noticed that each language was a subset of the next one. A hierarchy of languages and their associated grammars was created by Noam Chomsky in 1956. It is known as the Chomsky hierarchy. The Chomsky hierarchy has **four** main levels.


Type -0
Type -1
Type -2
Type -3
Unrestricted Grammar (Accepted by Turning machine)
Context Sensitive Grammar (Accepted by Linear bound automata)
Context Free Grammar (Accepted by Pushdown automata)
Regular Grammar (Accepted by Finite automata)
Shaw Academy

Type 0

Type 0 are unrestricted languages that are generated by Turing machines. Turing machines can calculate everything you can imagine (everything is decidable).

Type 1

In Type 1, context-sensitive languages are generated by linear-bound non-deterministic Turing machines. They can deal with different contexts because they are non-deterministic and have complete reference to the past.

Type 2

Type 2 are context-free languages generated by non-deterministic pushdown automata. They have memory – but because a stack can only be viewed from the top, complete reference to the past is not available.

Type 3

Type 3 are regular languages are generated by regular automata. They have limited past knowledge, that is, their compute memory has limits, so when you have a language with suffixes depending on prefixes (palindrome language) this cannot be done with regular languages.

Conclusion

In this lesson, we looked at one of the most critical concepts of computer science – computer language, which gives you a pretty good picture of what we will be trying to do as we write code. As we go along, the pieces will start falling into place as you see where each of these concepts is applied.

References

- Brilliant.org. (2011). Finite State Machines | Brilliant Math & Science Wiki. [online] Available at: https://brilliant.org/wiki/finite-state-machines/
- Aaby, A. (n.d.). Theory Introduction to Programming Languages. [online] Available at: http://pooh.poly.asu.edu/Ser515/Schedule/docs/TheoryIntroductionToPgmmingLangs-Aaby-draft.pdf
- Pushdown Automata CS351. (n.d.). [online] Available at: http://www.math.uaa.alaska.edu/~afkjm/cs351/handouts/hop-chap6.pdf.
- Pushdown Automata. (n.d.). [online] Available at: https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/17/Small17.pdf
- Carter, T. (2005). Introduction to theory of computation. [online] Available at: https://csustan.csustan.edu/~tom/SFI-CSSS/Lecture-Notes/Computation/computation.pdf
- Cam.ac.uk. (2012). Department of Computer Science and Technology – Raspberry Pi: Introduction: What is a Turing machine? [online] Available at: https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.
- Alanturing.net. (2020). AlanTuring.net What is a Turing machine? [online] Available at: http://www.alanturing.net/turing_archive/pages/Reference
- Köhler, S., Schindelhauer, C. and Ziegler, M. (2005). On Approximating Real-World Halting Problems. Fundamentals of Computation Theory, [online] pp.454–466. Available at: https://link.springer.com/chapter/10.1007/11537311_40.
- Harper, R. (n.d.). [online] Available at: https://www.classes.cs.uchicago.edu/archive/2010/fall/22100-1/book/Harper2005.pdf.
- Rakhim Davletkaliyev (2016). Human and computer languages. [online] Hexlet.io. Available at: https://en.hexlet.io/courses/intro_to_programming/lessons/language/theory_unit
- Saylor Academy. (2019). CS201: Wikipedia: "Words in Computer Architecture" | Saylor Academy. [online] Available at: https://learn.saylor.org/mod/page/view.php?id=18960
- Feodor and Dragan, F. (2018). Theory of Computation. [online] Available at: http://www.cs.kent.edu/~dragan/ThComp/lect01-2.pdf
- Engineering LibreTexts. (2018). 4.4: Pushdown Automata. [online] Available at: https://eng.libretexts.org/Bookshelves/Computer_Science/Book%3A_Foundations_of_Computation_(Critchlow_and_Eck)/04%3A_Grammars/4.04%3A_Pushdown_Automata