1 Introduction

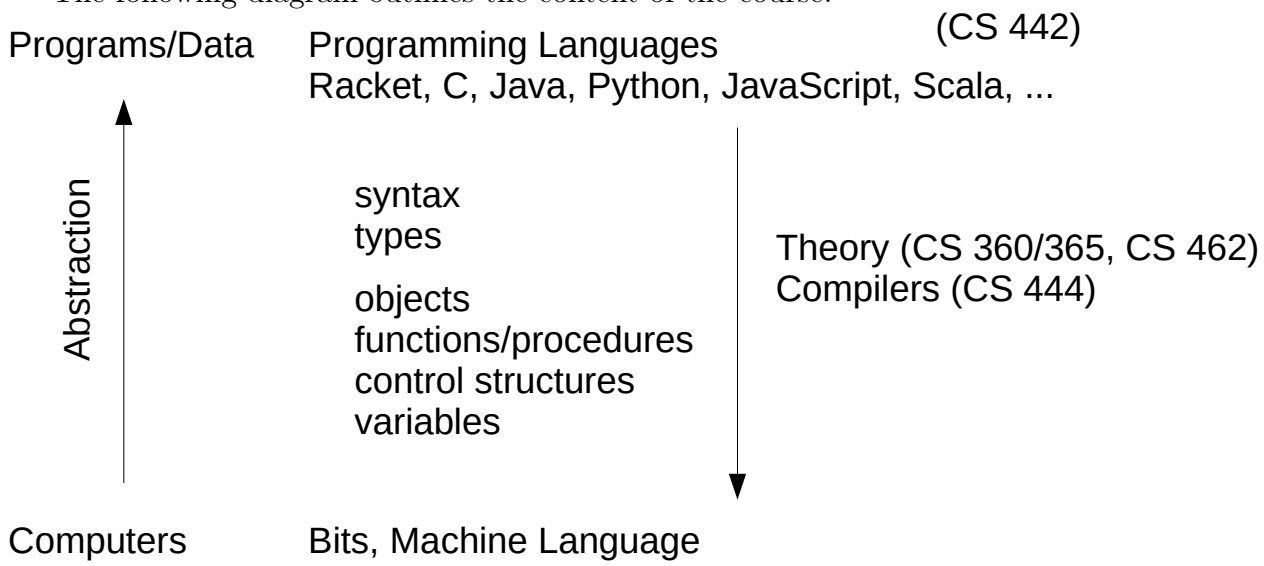
1.1 CS 241E: About the course

This course is about high-level programming languages and the machine architecture that is used to implement them.

We write programs to solve problems that involve real-world concepts such as numbers, text, images, sounds, or video. In those programs, we use programming languages features like variables, expressions, control structures, functions, and objects.

A computer is just a machine for manipulating bits, zeros and ones. To make a computer work with high-level concepts, we need to encode those concepts using bits. We will start the course with just a computer, and we will gradually build up *abstractions* to represent higher-level concepts. The first abstractions will be implemented directly in terms of the computer, but later abstractions can be layered on top of earlier abstractions that we have already defined.

The following diagram outlines the content of the course:



The arrow on the left shows that we progressively add abstractions when moving from physical computers to programs that manipulate real-world data and solve real-world problems. CS 251 and ECE 222 cover more details about computer hardware. The arrow on the right illustrates compilers, which are programs that remove abstractions, translating a program from a high-level programming language to the machine language supported by the computer hardware. We will learn some basics about compilers in this course, but more details are covered in CS 444. The practice of writing compilers is facilitated by theoretical results. Although we will not prove any theorems in this course, we will learn about and apply some of them. The theoretical aspects are covered in more detail in CS 360 or 365, and in CS 462. Finally, in this course, we translate features of high-level programming languages into low-level implementations. More details about programming languages and their features are covered in CS 442.

1.2 Interpreting bits

Consider a sequence of bits that can appear in a computer, such as 1000011. On its own, this bit sequence has no inherent meaning; they are just bits. We can define *conventions* for interpreting bits by mapping specific bit sequences to concepts. For example, under one convention, this bit sequence might mean the number 67. Under another convention, it might mean the letter C. Under still other conventions, the same bit sequence might mean the negative number -61 or the negative number -3. The conventions for interpreting bits are our choice, and there is no meaning inherent in the bits themselves.

1.3 Binary numbers

One concept that we often work with are numbers. We could choose any mapping we want from bit sequences to numbers, but some mappings are particularly convenient. A common mapping is the *binary* encoding of numbers. In this encoding, each bit is a coefficient of a power of two. To convert a bit sequence to its corresponding number, we sum the powers of two that correspond to 1-bits in the sequence.

Example 1. *In binary, the bit sequence 1000011 represents the number*

$$1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 67$$

To convert in the other direction, we need to decompose the desired number into a sum of powers of two. A greedy approach works: always include the greatest power of two that is less than or equal to the remaining number.

Example 2. *To write* 42 *in binary, the greatest power of two is* $2^5 = 32$*, since* 64 *would be too great. After choosing to include* 32 *in our sum, we need to find powers of two that add to* $42 - 32 = 10$*. The greatest such power of two is* $2^3 = 8$*, leaving* $10 - 8 = 2$*, which is* 2^1*. We conclude that*

$$42 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

so we write it using the bit sequence 101010.

An alternative method to perform this conversion is to repeatedly divide the desired number by 2, record the remainders, then read them in reverse.

Example 3. *42 divided by 2 gives 21 with remainder 0.*
21 divided by 2 gives 10 with remainder 1.
10 divided by 2 gives 5 with remainder 0.
5 divided by 2 gives 2 with remainder 1.
2 divided by 2 gives 1 with remainder 0.
1 divided by 2 gives 0 with remainder 1.
Reading the remainders in reverse gives 101010, the binary representation of 42.

Exercise 4. *Prove that this method always yields the correct bit sequence.*

Exercise 5. *We will need to encode and decode binary numbers frequently throughout the course. If these concepts are new to you, try converting some numbers of your choice to and from binary for practice.*

1.4 Negative numbers – two's complement

In many situations, only some fixed number of bits are available to encode a concept. After all, the total number of bits in the memory of a physical computer is finite. We may choose some fixed number of bits n, such as $n = 32$, and represent numbers using sequences of n bits.

Although there are infinitely many integers, there are only 2^n sequences of n bits. Since we can distinguish at most 2^n of the infinitely many integers, we need to carefully choose a convenient subset of the integers to represent. The binary number convention from the previous section maps sequences of n bits to the non-negative integers in the range from 0 to $2^n - 1$.

It is often useful to be able to represent negative integers. We can choose a different convention that maps some bit sequences to negative integers. Again, we could choose any mapping from bit sequences to integers as our convention, but some mappings are particularly convenient.

A common convention for representing signed (positive and negative) integers is called *two's complement*. This convention maps sequences of n bits to integers in the range from -2^{n-1} to $2^{n-1} - 1$. In this range, about half the integers are positive and half are negative. In a two's complement representation, the bits are also coefficients of powers of two, with the difference that the first power of two is negated.

Example 6. *In two's complement, the bit sequence 1000011 represents the number*

$$\mathbf{1} \cdot -2^6 + \mathbf{0} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0 = -61$$

Exercise 7. *Convince yourself of the range of integers that can be represented using n-bit two's complement numbers by finding the bit sequences representing the least and greatest integers representable with n bits.*

The property that makes two's complement convenient is that if any sequence S of n bits is interpreted both as a binary number b and as a two's complement number t, then $b \equiv t \pmod{2^n}$. To understand why this is the case, consider that the first coefficient in the binary interpretation is 2^{n-1}, while in the two's complement interpretation, it is -2^{n-1}, and that all the other coefficients are unchanged. Since $2^{n-1} \equiv -2^{n-1} \pmod{2^n}$, the corresponding coefficients in the binary and two's complement interpretations are all congruent to each other modulo 2^n, so their sums are also congruent to each other.

The benefit of this property is that if the computer hardware implements arithmetic modulo 2^n, then the same hardware can be used to perform addition, subtraction, and multiplication on both unsigned binary numbers and on signed two's complement numbers. For example, since $-2 \equiv 14 \pmod{2^4}$ and $-3 \equiv 13 \pmod{2^4}$, it is also the case that $-2 + -3 \equiv 14 + 13 \pmod{2^4}$. A single arithmetic circuit implementing addition modulo 2^4 can add both $-2 + -3$ and $14 + 13$. A caveat is that such a circuit does not produce an explicit integer, but a bit sequence that represents an equivalence class of all the integers that are congruent to each other modulo 2^4. In this case, the output of the addition will be the bit sequence 1011, which represents all integers that are congruent to 11 modulo 2^4, including -5, 11, and 27.

With a large number of bits, it can be difficult to determine the two's complement encoding of a small negative number, but there is a trick to make it easier. For example, the 32-bit two's complement encoding of -1 is the same as the 32-bit unsigned binary encoding of $2^{32}-1$ since $-1 \equiv 2^{32}-1 \pmod{2^{32}}$, but $2^{32}-1 = 4294967295$ and determining the binary encoding of such a large number requires a lot of tedious arithmetic. The trick to find the two's complement representation of any negative number $-x$ is to write the unsigned binary encoding of the positive number x, flip all its bits (replacing each 0 with a 1 and each 1 with a 0), and add 1 to the resulting binary number. For example, to encode -1, we would write the unsigned binary encoding of positive 1, which is 00000000000000000000000000000001, flip all the bits to get 11111111111111111111111111111110, and add 1 to get 11111111111111111111111111111111. This is the 32-bit two's complement encoding of -1 and also the unsigned binary encoding of 4294967295.

Exercise 8. *Prove that this trick works. As a hint, notice that* $1-0=1$ *and* $1-1=0$*, and think about what flipping bits means in an arithmetic sense.*