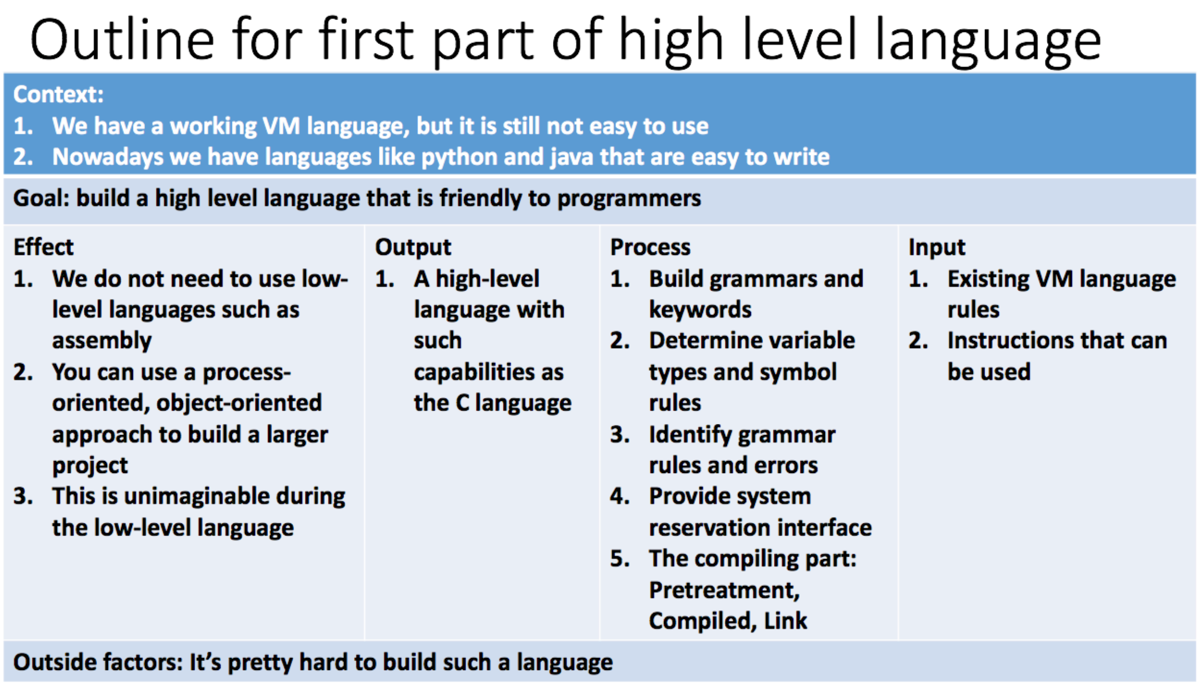
High-Level Language

NaXin



Outline for first part of high level language

Context:
1. We have a working VM language, but it is still not easy to use
2. Nowadays we have languages like python and java that are easy to write

Goal: build a high level language that is friendly to programmers

Effect	Output	Process	Input
1. We do not need to use low-level languages such as assembly 2. You can use a process-oriented, object-oriented approach to build a larger project 3. This is unimaginable during the low-level language	1. A high-level language with such capabilities as the C language	1. Build grammars and keywords 2. Determine variable types and symbol rules 3. Identify grammar rules and errors 4. Provide system reservation interface 5. The compiling part: Pretreatment, Compiled, Link	1. Existing VM language rules 2. Instructions that can be used

Outside factors: It’s pretty hard to build such a language

- This section is recommended for viewing along with **The Element of Computing system**.

Introducing Jack language

- So far,The book is all about the hardware and the lower layer of the system, and usually users will not directly deal with them. Therefore, we introduce here a simple, object-oriented, high-level language called **Jack**.
- One feature of Jack language is that it is very similar to Java, but its grammar structure (or Syntax) is very simple. Using **context-free grammar** to describe the language clear enough requires only a little space(The Element of Computing system P208~209).
- Of course, even so, Jack still has the same ability as other languages. This means that anything that can be done in other languages can be implemented using Jack. (Of course it may not be so fast)

background

- Why do we need high-level languages?

We use staging strategy to rise from the bottom up step by step, and they are all dealing directly with the logic gates. Imagine if our underlying architecture changed, a specific program we wrote would not be able to run on the new architecture. To solve this problem, we can design an "idealized" high-level language, as long as the "compiler" provided by the underlying architecture can successfully compile the high-level language into the language of this architecture, and the language we design can be used in all architectures, means to run on them.

High-level language

Definition

A simple way to define it is that high-level languages are a system off the hardware that are written in a way that is easier for people to understand.

- Departing from the hardware system means that its instructions are **independent** and are just **convenient representations** that are **easier to express**. At the same time, the high-level language code is more **readable**. An example is declaring an array (as a data structure). The high-level language only cares about how to use it. It doesn't have to worry about how to save the array in memory.
- The readability is: The encoding process can be simply understood as telling the machine what to do next.

Category

We briefly introduce the classification of four high-level languages:

1. **Imperative language**: The basis is a computable Turing machine model that simulates “data storage/operations”, which is a way to meet the current mainstream architecture.
2. **Functional languages**: languages based on mathematical function concepts and mappings, such as Haskell
3. **Logical language**: language built based on a set of known formal logic systems
4. **Object-Oriented**: Most high-level languages now have object-oriented as a basic model

Jack Language Specification Details

This section refers directly to ** The Element of Computing system 9.2**. We recommende readers also read the contents of the** 9.1** section here. So that you will have a good understanding of the syntax of the high-level language.

Additional points

We believe that after reading **9.2** of the book, you already have a fairly good understanding of Jack's language. Let's take a look at some of the parts that we think are important and interesting.

Subroutine call

- In the VM language, we already have the concept of a subroutine call. For example, when I need the product of two numbers, I don't need to implement multiplication **on the spot**. Instead, I call a function, which reduces the workload. And moreover, the code is also simpler.
- This involves the **modularization** issue. We divide the problem into some typical modules. Each module runs step by step to get the results we want. The advantage is that if we need

to use a certain part in the future, we can directly call this module without re-implementation, code reuse.

expression

In Jack's expression section, we find that when two expressions are connected by one symbol we get another expression. Here is the idea of the so-called **"recursion"** in which expressions can continuously generate expressions.

Standard Library

The standard library is the function **provided by the system** itself and can be called directly. We will explain this in detail in Chapter 12.

Some notes about this chapter

The project in Chapter 9 does not require coding. but you need to have enough knowledge of Jack to start building Jack's compiler in the coming chapter.