Interpreter Architecture Style

Group 17

Yajie Hao
Yehao Wang
Qiuhan Li

What is the Interpreter Architecture Style (IAS)?

The Interpreter Architecture Style (IAS) interprets a higher level language to a lower level language to directly execute a series of commands, without requiring programs being compiled ahead of time.

What are components?

- Engine
  - Receives program's current progress and generates a new state of program and interpreter
- Interpreter State
  - The current state of the interpreter itself, it receives the progress of source codes being interpreted and sends the interpreter current state
- Program State
  - The progress of source codes being interpreted, it receives and stores user's command, sends and stores program's current progress of interpretation
- Source Code
  - The program being interpreted

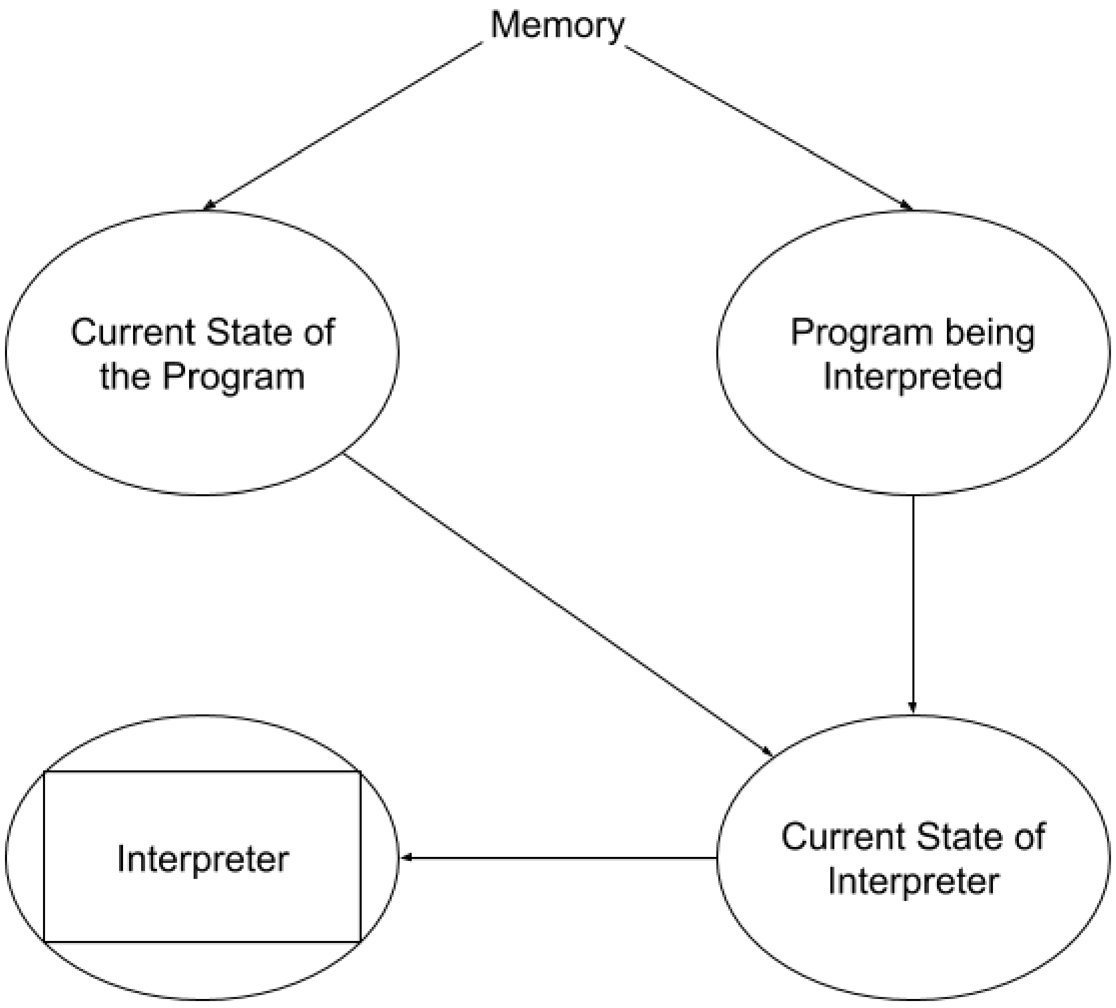
What are connectors?

- Procedure calls
  - The interface where users interact with the interpreter
- Memory access
  - The engine access memory to get or modify data

What are examples in practice?

- Python and Ruby
- Implement machine code in software
- Emulate a clever and cheap chip in a calculator
- Excel Formulas
- Display and modify graphs by entering user commands
- Database engine accepts a series of queries from users

What topological constraints does it impose (diagram)?


Memory
Current State of the Program
Program being Interpreted
Interpreter
Current State of Interpreter

IAS includes one computation state machine and three sets of memory as shown in the graph above.
An interpreter reads source code written in a particular programming language and enacts them. Input to the program (source code) is sent to the program state, which is read by the engine. Then the current states of interpreter and program are passed to the engine. Finally, new states of interpreter and program are sent to program state and interpreter state, and/or the engine provides user the output in the user interface.

What specific problems can be tackled?

Applications or machines using IAS are suitable to solve problems when the most appropriate languages or machines for executing the corresponding solutions of the problems are not directly available.

What specific changes can be adapted?

Applications or machines using IAS remain effective while higher level programming language is developed to solve complex problems, or even new programming languages are invented.

What are advantages?

- Dynamic adaption
  - Highly dynamic behaviours are possible even user commands change all the time
- Simulation of hardware
  - Simulate non-implementable hardware that is unaffordable
- High portability
  - Applications or machines using IAS can function well across a variety of platforms
- Easy to understand
  - Turns the how do we do this into the how do we say this is the way we can do it
- Customization
  - Ad hoc behaviours of applications or machines defined by a custom language make development of software much easier.

What are disadvantages?

- Performance
  - Executing code using interpreter style is relative slow compared to compiled style. Interpreting code is slower than compiling code because the interpreter have to check each command according to a library; however, compiling code is faster than interpreting code because each command is optimized.

What NFPs are supported/inhibited?

- Efficiency
  - IAS may be by nature very slow due to indirect execution, checking of each command based on a given library during execution makes interpreter less efficient
- Portability
  - Applications using IAS still work well across various platforms, interpreter allows machine code which are intended to run on one specific hardware architecture to work on another type of hardware architecture

Presentation Scripts

Student A - Interpreter Engine/Interpreter/Compiler
Student B - User
Student C - Narrator
Laptop - Program being interpreted
Card A - Current program state
Card B - Current interpreter state
Surroundings - Platforms

Example 1 - Normal Usage

Here student A is an engine.

Student B asks student A to feed his/her dog as soon as possible because it is time to perform D2 activity. As response, student A checks the laptop to see which instructions are used to feed the dog, then he/she updates current state of him/herself and laptop on card A and B. Finally, student B's request is satisfied and the dog finishes her dishes!

Example 2 - Easy to Understand

First student A will be an interpreter.

Student B requests student A to pour the water into a glass cup and student A performs the task without further explanation.

Then student A becomes a compiler.

Student B requests student A to pour the water into a glass cup, student B is too confused to conduct that request. Then student B explains the process of the task and claims the constraints, and finally student B performs the task.

We can see instruction or command in interpreter is more similar to human language, Therefore it is easier to understand.

Example 3 - High Portability

Here student A is an engine.

Student A is able to conduct student B's requests within various surroundings. For example, student A will perform tasks in the kitchen, common area or bedroom.

Different surroundings aim to emphasize the high portability of interpreter, the interpreter is able to perform in different working environment.

Example 4 - High Flexibility

Here student A is an interpreter.

Student A can perform tasks written in a Mac laptop, but he/she is too confused to understand tasks written in a Windows laptop. Fortunately, we have found student A's twin brother who can understand Windows tasks.

In conclusions, suitable interpreters for programs can always be found, or if such interpreters do not exist we can create and custom one.

Example 5 - Poor Efficiency

Student B requests student A to pour water multiple times in this example.
In this case student A is an compiler.
Student A opens the cap of bottle, pours the water into the glass and does not put the cap back on to the bottle. Then continue pouring the water until all instructions are received, student A eventually put the cap back on to the bottle.
In next case student A becomes a interpreter.
Student A opens the cap of bottle, pour the water into the glass and put the cap back on to the bottle. Then, performing the exact same process until all instructions are received.
Since interpreter has to read through all source codes to execute every time, the process of putting the cap back on to the bottle symbolize this characteristic. Compiler only reads through all source codes once and generates an executable to execute, the process of not putting the cap back on to the bottle symbolize this quickness.