Programming Language Concepts

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Learning Group Activity

With your learning group:

1. Share your definitions of a programming language. Discuss what each definition entails, and whether you think it could be improved.
2. Create one collective definition.

Got a good one? There will be a chance for you to share with the class.

Two Parts to a Language

Design vs. Implementation

- **Design:** The language specification; that is, given an input, how *should* the implementation behave?
- **Implementation:** A compiler or interpreter which behaves according to the language design.

Example

C is a programming language, but GCC is an implementation of the C programming language.

Discuss

1. Someone tells you a language is *fast*. Are they referring to the design, or the implementation? Or both? Why?
2. Is it always correct to separate the concepts of design and implementation? When can we get into murky water?

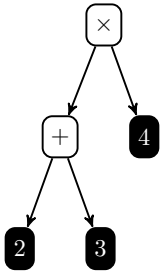
Abstract Syntax Trees

Abstract Syntax Trees

Consider this simple mathematical expression:

$$(2 + 3) \times 4$$

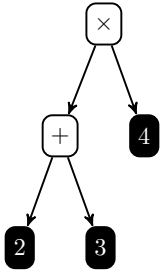
We could convert this to a tree structure that represents the same expression:



This kind of tree structure which represents the syntax of an expression is called an **Abstract Syntax Tree** (AST).

Symbolic Expressions

Since drawing abstract syntax trees is a lot of work, there exists a notation called **symbolic expressions** (or **s-expressions**) that makes it a little easier.



This converts to the following s-expression:

```
(* (+ 2 3) 4)
```

Exercise

With your learning group, for each math expression, draw an abstract syntax tree, and write out the resulting s-expression.

1. $6 \times 7 + 8$
2. $\dfrac{1}{2 \times 3 \times 4}$
3. $2^3 + 5$

Evaluating an AST

To get the result from abstract syntax tree, we could write a simple program to do this:

```
procedure eval(node):
    if node is a literal then return node
    otherwise, if node.operator is...
        +, then:
            sum <- 0
            for each child in node:
                sum <- sum + eval(child)
            return sum
        *, then:
            ...
```

A program which does this is called an **interpreter**. We'll present a more formal definition of this in a few more slides.

Compiling an AST

You could imagine a program which takes in an AST and creates machine code (pseudocode omitted):

```
                              ADD     R1 <- 2, 3
(* (+ 2 3) 4)     ->         MUL     R1 <- R1, 4
                              RETURN  R1
```

This kind of a program is called a **compiler**. Again, formal definition coming soon.

Language Implementation Techniques

Compiled Languages

Compiler: *A computer program which translates a high-level language (such as C) into machine code.*

Machine Code: *A set of instructions which can be directly executed by a CPU.*

Typically, when we speak of a **compiled language**, we refer to one which can be compiled to machine code. Compilers which translate to virtual machine bytecode (e.g., Java and Python) are better categorized as **interpreted languages**.

Compiler Implementations

Advantages:

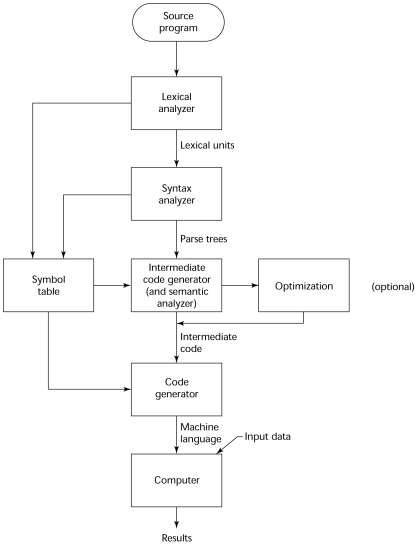
- Runtime is *fast!*

Disadvantages:

- Compile time is slow
- Source code cannot be a part of the input data

Examples

C, C++, and FORTRAN are generally implemented as compiled languages



Interpreted Languages

Interpreter: *A computer program which reads a high-level programming language and **directly executes** the instructions of the language itself.*

An **interpreted language** is a language designed to be implemented using an interpreter.

Discuss

What could be the advantages of executing the language directly? Disadvantages? Try and come up with an example of something that could be done with an interpreter model but not a compiler model.

Interpreter Implementations

Advantages:

- No need to compile
- Source code *can* be a part of input data: you can transmit functions across the network to be run!

Disadvantages:

- Runtime is slow

Examples

BASIC, PHP, and Perl are generally implemented as interpreted languages

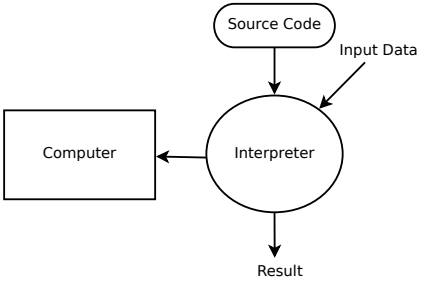


Figure: Model of a classic interpreter. Modern interpreters are slightly more complicated.

Hybrid Interpreters

To speed up the execution of interpreted languages, implementers started getting clever:

- **Interpreted VM Bytecode:** Input is lexed, parsed, then translated to bytecode. The bytecode gets optimized, then the low level bytecode is interpreted. Examples: CPython, OpenJDK (Java), Ruby MRI
- **Just In Time Compiler:** Source code is compiled as it's executed, putting machine code on the processor "just in time". Examples: PyPy, LuaJIT, Chrome V8

Advantages include all the benefits of interpreted languages, with run times occasionally approaching compiled languages.

Bytecode Example

In CPython, the following Python function:

```
def hello()
    print("Hello, World!")
```

will be translated to the following bytecode[1]

```
2           0 LOAD_GLOBAL              0 (print)
            2 LOAD_CONST               1 ('Hello, World!')
            4 CALL_FUNCTION            1
```

Note

These instructions cannot be performed directly by the CPU, however, if you can create a translation from every CPython bytecode instruction to a given CPU architecture, you can run Python on that architecture.

[1] the actual bytecode is a binary, non-human-readable format. This is a human-readable translation. (Source: https://opensource.com/article/18/4/introduction-python-bytecode)

Typical Interpreter Structure

1. **Lexer:** Source Code $\rightarrow$ Tokens
2. **Parser:** Tokens $\rightarrow$ Abstract Syntax Tree (AST)
3. **Analyzer (optional):** AST $\rightarrow$ AST (optimized)
4. **Evaluator:** AST + Context $\rightarrow$ Result + Context