EDAP05: Concepts of Programming Languages - Reference Sheet

Karl Hallsby

Last Edited: November 6, 2019

Contents

1 Language vs. Language Implementation

It is important that we make the distinction between a programming language and the programming language's implementation.

- A programming language and its implementation are completely separate things
    - Technically, they are related, in that a programming language implementation is one way to fulfill the specifications that the programming language introduces
    - You can implement a programming language in different ways. For example, C has these well-known implementations:
        - gcc
        - LLVM/clang
        - MSVC

2 Programming Language Implementations

There are 3 main ways for a programming language to be implemented:

1. Interpretation
2. Compilation
3. Hybrid Implementation

There are benefits and drawbacks for each of these implementations:

Property	Interpretation	Compilation	Hybrid Implementation
Execution Performance	Slow	Fast	Fast
Turnaround	Fast	Slow (Compile and Link)	Fast (Compile when needed)
Language Flexibility	High	Limited	High

Table 2.1: Pros and Cons for Programming Language Implementations

There is a trade-off to be made between:

- Language flexibility
- CPU time / RAM time

2.1 Interpretation

Defn 1 (Interpretation)**.** If a programming language is implemented with *interpretation*, is *interpreted*, then there is an intermediate program that runs between the source code and what the CPU can run on. This *interpreter* reads the high-level source code, then alternates between:

- Figure out next command
    - This means that the current instruction is parsed in
    - Equivalent commands are generated in the CPU-specific or VM-specific instruction sets from the high-level source code
- Execute Command

Some examples of languages with a Interpretation implementation are:

- Python
- Perl
- Ruby
- Bash
- AWK
- ···

2.2 Compilation

Defn 2 (Compilation)**.** If a programming language is implemented with *compilation*, is *compiled*, then there are several programs that must be run before the high-level source code can be run.

1. The Compiler
2. The Assembler
3. The Linker
4. The Loader

Defn 3 (Compiler)**.** The *compiler* is the main program needed in a compiled language implementation. It is responsible for taking the high-level source code written in some language, and converting it to assembly code, which can then be run through an Assembler.

The steps involved in a compiler are:

1. Lexical Analysis/Tokenizing: Convert the input file into a set of tokens
2. Syntactic Analysis/Parsing: Convert the tokens into a tree representing all the tokens in the program
3. Semantic Analysis: Interpret the program and ensure that everything expressed in the program is correct.
   - This is where compile-time errors are **usually** caught. Though, this is just a generalization.
   - Type analysis is handled here for instance
4. Optimize the Code: The output assembly code could be optimized before actually making the output. Take care of that here.
5. Output Assembly: With the potentially optimized machine-equivalent code from our program, write out the equivalent assembly, and finish the compilation process.

Remark 3.1. The specifics of a Compiler's implementation are **not** discussed in this course, but it is useful to know the basics of the compilation process. For both the implementation details, please refer to EDAN65:Compilers-Reference Material.

Defn 4 (Assembler)**.** The *assembler* is an intermediate program used after the Compiler has been run. The assembler takes the assembly code that the Compiler outputs and applies a one-to-one mapping. Since all assembly code is just an abstraction and humanization of machine code in a one-to-one mapping fashion, the assembler takes the assembly code and converts it to its equivalent machine code.

Remark 4.1. This particular program is not discussed heavily in this course.

Defn 5 (Linker)**.** The *linker* is an intermediate program, that may be provided by the operating system or may be provided by that language implementation's tooling. It is run after the Compiler and/or the Assembler have been run.

- Provided by operating system
  - If the programming language implementation relies on the operating system and critical portions of the system.
- Provided by the language implementation's tooling
  - If the implementation provides certain libraries, it will likely have their own linker too.

Remark 5.1. This particular program is not discussed in this course.

Defn 6 (Loader)**.** The *loader* is the program provided by the operating system that loads the specified program into main memory and begins execution.

Remark 6.1. This particular program is not discussed in this course.

Some examples of languages with a Compilation implementation are:

- C
- C++
- SML
- Haskell
- FORTRAN
- $\cdots$

2.3 Hybrid Implementation

Defn 7 (Hybrid Implementation)**.** A programming language can be implemented with a *hybrid implementation.* This means that it takes some aspects of a language implemented by Interpretation and some aspects of the language implemented with Compilation.

For example, Java does this with their Just-In-Time (JIT) compilation scheme.

One way to do this is with Dynamic Compilation.

2.3.1 Dynamic Compilation

- Idea: behind dynamic compilation is that code is compiled *while executing*.
- Theory: The best of Interpretation and Compilation worlds.
- Practice:
  - Difficult to build
  - Memory usage can increase (sometimes dramatically)
  - Performance can be higher than pre-compiled code, because only the code needed is compiled.

Some examples of these are:

- Java
- Scala
- C#
- JavaScript
- ⋯

3 Language Critique

There are several very open-ended questions that need to be asked when categorizing and critiqueing languages:

1. What programming language is best for *what task*?
2. *What criteria* do we measure?
   - Most criteria do not have good measurement tools.
3. *How* do we obtain measurements for these criteria?

These are all qualities of:

- The language
- The language implementation(s)
- The available tooling for the language and that particular implementation
- The available libraries for the language and that particular implementation
- Other infrastructure
  - User groups
  - Books
  - etc.

	Criteria		
Characteristic	Readability	Writability	Reliability
Simplicity	✓	✓	✓
Orthogonality	✓	✓	✓
Data Types	✓	✓	✓
Syntax Design	✓	✓	✓
Support for Abstraction		✓	✓
Expressivity		✓	✓
Type Checking			✓
Exception Handling			✓
Restricted Aliasing			✓

Table 3.1: Language Evaluation Criteran and the Characteristics that Affect Them

3.1 Readability

Defn 8 (Readability)**.** *Readability* is how easily a program can be read and understood *by a human.* Some languages do not support certain functions, but programmers try to make the language do what it is not designed to do. This will lead to complicated and difficult-to-read programs.

The idea of program readability was first presented as the software life-cycle concept (Booch 1987). The initial coding was downplayed compared to earlier, and the maintenance and improvement of the code was brought to the forefront.

3.1.1 Simplicity

There are 2 main factors and 1 minor factor for a language's simplicity:

1. The number of features present in the language.
2. The Feature Multiplicity of a language.
3. The ability to Overload Operators.

Assembly language is on the most-simple end of the simplicity spectrum. In assembly, the form and meaning of most statements are incredibly simple, but without more complex control statements, the program's structure is less obvious.

Defn 9 (Feature Multiplicity)**.** *Feature multiplicity* is when there is more than one way to accomplish a particular operation with language built-in features. For example, in Java these are all equivalent when evaluated as standalone expressions:

1. `count = count + 1`
2. `count += 1`
3. `count++`
4. `++count`

```
def d(x):
    r = x[::-1]
    return x == r
```

Defn 10 (Overload Operator)**.** An *overloaded operator* is one where a single symbol has more than one meaning. For example the `+` operator can be overloaded to add 2 integers, 2 floating-point numbers, or an integer and a floating-point number. This overloading helps *improve* the Simplicity of a language.

```
x = 3 + 4  # Evaluates to 7
y = 3.0 + 4.0  # Evaluates to 7.0
z = 3 + 4.0  # Evaluates to 7.0
```

3.1.2 Orthogonality

Defn 11 (Orthogonality)**.** *Orthogonality* in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of a language. Additionally, every possible combination of primitives is legal and meaningful.

The more orthogonal a language, the fewer exceptions to the language rules there can be. These fewer exceptions means there is a higher degree of regularity in the language design, making it easier to learn, read, and understand.

Remark 11.1 (Over-Orthogonality). Too much orthogonality can cause problems. Having too much combinational freedom with primitive constructs and their combinations can make for an extremely complex compound construct. This leads to unnecessary complexity.

```
// global variable section

float f1 = 2.0f * 2.0f;
float f2 = sqrt(2.0f); // error
```

3.1.3 Data Types

The use of data types conveys intent when reading and writing the program. For example, a boolean data type conveys a true/false value better than an integer that is 1/0 for true/false respectively.

- `timeOut = 1`
- `timeOut = true`

```
enum Color {
  Red, Green, Blue
};

Color c = readColorFromUser();
```

3.1.4 Syntax Design

There are 2 main syntactic design choices that affect Readability:

1. Reserved/Special Words
2. Form and Meaning

3.1.4.1 Reserved/Special Words

Defn 12 (Reserved Word)**.** *Reserved word*s are words that are reserved by the language constructors because those particular words have a meaning in the language. For example:

- `while`
- `class`
- `for`

There are also special words and matching characters that can denote groups of instructions.

- C and its decendants
  - Matching brances
  - { and }
- Ada/Fortran 95 and their decendants:
  - Distinct closing syntax for each statement group
  - `end if` to end an if statement

Also, can these Reserved Words be used as names for program variables? If so, this will increase overall complexity of a program. The code block below, from Fortran 95, illustrates this point.

```
program hello
  implicit none
  integer end, do
  do = 0
  end = 10
  do do=do, end
    print *,do
  end do
end program hello
```

3.1.4.2 Form and Meaning Statements should be designed such that their appearance partially indicates what their purpose is. For example, the UNIX command `grep` gives no hint at what it is supposed to do, unless you know the text editor `ed`.

Semantics, or meaning, should follow directly from syntax or form. In some cases, this principle is violated by 2 language constructs that are identical or similar in appearance, but different in meaning, depending on the context. For example, C's `static` Reserved Words.

3.2 Writability

3.2.1 Simplicity and Orthogonality

3.2.2 Support for Abstraction

Defn 13 (Abstraction)**.** *Abstraction* is the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. This is a key concept in modern programming language design.

There are 2 categories of abstraction:

1. Process Abstraction
2. Data Abstraction

3.2.2.1 Process Abstraction Process abstraction is the use of a subprogram to implement some block of code used multiple times. For example, a sorting algorithm. If the code for the algorithm could not be factored out into a separate piece of code, the algorithm would need to be copied everywhere it was used. This would lead to additional complexity and reduce the Readability of the code.

3.2.2.2 Data Abstraction

For example, representing a binary tree in C++/Java is done by making a tree node class that has 2 pointers and an integer. This abstraction is more natural to think about than what would need to be done in Fortran 77. In Fortran 77, there would need to be 3 parallel integer arrays, where 2 of the integers in each array would be used as subscripts to find their children.

3.2.3 Expressivity

3.3 Reliability

3.3.1 Type Checking

3.3.2 Exception Handling

3.3.3 Aliasing

3.3.4 Readability and Writability

A Trigonometry

A.1 Trigonometric Formulas

$$\sin(\alpha) + \sin(\beta) = 2\sin\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right) \tag{A.1}$$

$$\cos(\theta)\sin(\theta) = \frac{1}{2}\sin(2\theta) \tag{A.2}$$

A.2 Euler Equivalents of Trigonometric Functions

$$e^{\pm j\alpha} = \cos(\alpha) \pm j\sin(\alpha) \tag{A.3}$$

$$\cos(x) = \frac{e^{jx} + e^{-jx}}{2} \tag{A.4}$$

$$\sin(x) = \frac{e^{jx} - e^{-jx}}{2j} \tag{A.5}$$

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2} \tag{A.6}$$

$$\cosh(x) = \frac{e^{x} + e^{-x}}{2} \tag{A.7}$$

A.3 Angle Sum and Difference Identities

$$\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta) \tag{A.8}$$

$$\cos(\alpha \pm \beta) = \cos(\alpha)\cos(\beta) \mp \sin(\alpha)\sin(\beta) \tag{A.9}$$

A.4 Double-Angle Formulae

$$\sin(2\alpha) = 2\sin(\alpha)\cos(\alpha) \tag{A.10}$$

$$\cos(2\alpha) = \cos^2(\alpha) - \sin^2(\alpha) \tag{A.11}$$

A.5 Half-Angle Formulae

$$\sin\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1-\cos(\alpha)}{2}} \tag{A.12}$$

$$\cos\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1+\cos(\alpha)}{2}} \tag{A.13}$$

A.6 Exponent Reduction Formulae

$$\sin^2(\alpha) = \frac{1-\cos(2\alpha)}{2} \tag{A.14}$$

$$\cos^2(\alpha) = \frac{1+\cos(2\alpha)}{2} \tag{A.15}$$

A.7 Product-to-Sum Identities

$$2\cos(\alpha)\cos(\beta) = \cos(\alpha-\beta) + \cos(\alpha+\beta) \tag{A.16}$$

$$2\sin(\alpha)\sin(\beta) = \cos(\alpha-\beta) - \cos(\alpha+\beta) \tag{A.17}$$

$$2\sin(\alpha)\cos(\beta) = \sin(\alpha+\beta) + \sin(\alpha-\beta) \tag{A.18}$$

$$2\cos(\alpha)\sin(\beta) = \sin(\alpha+\beta) - \sin(\alpha-\beta) \tag{A.19}$$

A.8 Sum-to-Product Identities

$$\sin\left(\alpha\right) \pm \sin\left(\beta\right) = 2\sin\left(\frac{\alpha \pm \beta}{2}\right)\cos\left(\frac{\alpha \mp \beta}{2}\right) \tag{A.20}$$

$$\cos\left(\alpha\right) + \cos\left(\beta\right) = 2\cos\left(\frac{\alpha + \beta}{2}\right)\cos\left(\frac{\alpha - \beta}{2}\right) \tag{A.21}$$

$$\cos\left(\alpha\right) - \cos\left(\beta\right) = -2\sin\left(\frac{\alpha + \beta}{2}\right)\sin\left(\frac{\alpha - \beta}{2}\right) \tag{A.22}$$

A.9 Pythagorean Theorem for Trig

$$\cos^2\left(\alpha\right) + \sin^2\left(\alpha\right) = 1^2 \tag{A.23}$$

A.10 Rectangular to Polar

$$a + jb = \sqrt{a^2 + b^2}e^{j\theta} = re^{j\theta} \tag{A.24}$$

$$\theta = \begin{cases} \arctan\left(\frac{b}{a}\right) & a > 0 \\ \pi - \arctan\left(\frac{b}{a}\right) & a < 0 \end{cases} \tag{A.25}$$

A.11 Polar to Rectangular

$$re^{j\theta} = r\cos\left(\theta\right) + jr\sin\left(\theta\right) \tag{A.26}$$

B Calculus

B.1 Fundamental Theorems of Calculus

Defn B.1.1 (First Fundamental Theorem of Calculus)**.** The *first fundamental theorem of calculus* states that, if f is continuous on the closed interval $[a, b]$ and F is the indefinite integral of f on $[a, b]$, then

$$\int_a^b f(x)\,dx = F(b) - F(a) \tag{B.1}$$

Defn B.1.2 (Second Fundamental Theorem of Calculus)**.** The *second fundamental theorem of calculus* holds for f a continuous function on an open interval I and a any point in I, and states that if F is defined by

$$F(x) = \int_a^x f(t)\,dt,$$

then

$$\begin{aligned} \frac{d}{dx}\int_a^x f(t)\,dt &= f(x) \\ F'(x) &= f(x) \end{aligned} \tag{B.2}$$

Defn B.1.3 (argmax)**.** The arguments to the *argmax* function are to be maximized by using their derivatives. You must take the derivative of the function, find critical points, then determine if that critical point is a global maxima. This is denoted as

$$\operatorname*{argmax}_x$$

B.2 Rules of Calculus

B.2.1 Chain Rule

Defn B.2.1 (Chain Rule)**.** The *chain rule* is a way to differentiate a function that has 2 functions multiplied together.
If

$$f(x) = g(x) \cdot h(x)$$

then,

$$\begin{aligned} f'(x) &= g'(x) \cdot h(x) + g(x) \cdot h'(x) \\ \frac{df(x)}{dx} &= \frac{dg(x)}{dx} \cdot g(x) + g(x) \cdot \frac{dh(x)}{dx} \end{aligned} \tag{B.3}$$

C Complex Numbers

Complex numbers are numbers that have both a real part and an imaginary part.

$$z = a \pm bi \tag{C.1}$$

where

$$i = \sqrt{-1} \tag{C.2}$$

Remark (i vs. j for Imaginary Numbers). Complex numbers are generally denoted with either i or j. Since this is an appendix section, I will denote complex numbers with i, to make it more general. However, electrical engineering regularly makes use of j as the imaginary value. This is because alternating current i is already taken, so j is used as the imaginary value instad.

$$Ae^{-ix} = A\left[\cos(x) + i\sin(x)\right] \tag{C.3}$$

C.1 Complex Conjugates

If we have a complex number as shown below,

$$z = a \pm bi$$

then, the conjugate is denoted and calculated as shown below.

$$\overline{z} = a \mp bi \tag{C.4}$$

Defn C.1.1 (Complex Conjugate)**.** The conjugate of a complex number is called its *complex conjugate*. The complex conjugate of a complex number is the number with an equal real part and an imaginary part equal in magnitude but opposite in sign.

The complex conjugate can also be denoted with an asterisk ($*$). This is generally done for complex functions, rather than single variables.

$$z^* = \overline{z} \tag{C.5}$$

C.1.1 Complex Conjugates of Exponentials

$$\overline{e^z} = e^{\overline{z}} \tag{C.6}$$

$$\overline{\log(z)} = \log(\overline{z}) \tag{C.7}$$

C.1.2 Complex Conjugates of Sinusoids

Since sinusoids can be represented by complex exponentials, as shown in Appendix A.2, we could calculate their complex conjugate.

$$\begin{aligned}\overline{\cos(x)} &= \cos(x) \\ &= \frac{1}{2}\left(e^{ix} + e^{-ix}\right)\end{aligned} \tag{C.8}$$

$$\begin{aligned}\overline{\sin(x)} &= \sin(x) \\ &= \frac{1}{2i}\left(e^{ix} - e^{-ix}\right)\end{aligned} \tag{C.9}$$

References

[Boo87] Grady Booch. *Software Engineering with Ada*. 2nd ed. Redwood City, CA: Benjamin/Cummings, 1987.

[Seb12] Robert W. Sebesta. *Concepts of Programming Languages*. 10th ed. Pearson Education Inc., 2012.