EDAP05: Concepts of Programming Languages - Reference Sheet

Karl Hallsby

Last Edited: November 11, 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

1.1 Influences on Language Design

The 2 other major influences on the design of programming languages have been:

1. Computer Architecture
2. Programming Design Methodologies

1.1.1 Computer Architecture

The prevalent computer architecture used is the von Neumann Architecture. This is in contrast to the Harvard Architecture, and its descendant Modified Harvard Architecture.

Defn 1 (von Neumann Architecture)**.** In the *von Neumann architecture*, named after John von Neumann, instructions and data are stored in a shared memory location. The central processing unit, CPU, is separate from the memory, meaning it must fetch the instructions and data from memory before doing something. When the CPU computes something, it needs to store the result *back* in memory. The constant fetching of instructions/data and storage of results in memory means there is a bottleneck, the *von Neumann Bottleneck*.

Remark 1.1 (von Neumann Bottleneck)*.* The shared bus between the program memory and data memory leads to the *von Neumann bottleneck*, the limited throughput (data transfer rate) between the CPU and memory compared to the amount of memory. Because the single bus can only access one of the two classes of memory at a time, throughput is lower than the rate at which the CPU can work. This seriously limits the effective processing speed when the CPU is required to perform minimal processing on large amounts of data. The CPU is continually forced to wait for needed data to move to or from memory.

Since CPU speed and memory size have increased much faster than the throughput between them, the bottleneck has become more of a problem.

The execution of a machine code program on a von Neumann Architecture computer occurs in a process called the *fetch-execute cycle*. To find where each instruction is in memory, the CPU needs to have a *program counter*.

Functional or applicative programming languages, where applying functions to parameters does not lend itself to the von Neumann Architecture.

Remark 1.2 (Cache in the von Neumann Architecture)*.* In the original von Neumann Architecture, there was no such thing as *cache* on the CPU. In modern computers, cache is located on the CPU directly, and acts similarly to memory. However, it copies a block of memory into the cache and feeds the CPU from that, refreshing the cache less periodically, and allowing for fater instruction/data access rates. This is an example of the Harvard Architecture in the traditional von Neumann Architecture making a Modified Harvard Architecture.

Remark 1.3 (Alternative Names)*.* The von Neumann Architecture can also be called:

- von Neumann Model
- Princeton Architecture
- Dataflow Model

Remark 1.4 (Alternative Architectures)*.* The von Neumann Architecture is one way to implement a computational model. There are alternatives, namely the Harvard Architecture and its descendant Modified Harvard Architecture.

Defn 2 (Harvard Architecture)**.** The *Harvard architecture* is a computer architecture with separate storage and signal pathways for instructions and data. It contrasts with the von Neumann Architecture, where program instructions and data share the same memory and pathways.

This partition of instructions and data means the CPU can simultaneously read an instruction and perform data memory access. Additionally, the address space for the instructions and data are separate, meaning instruction address zero is not the same as data address zero.

Defn 3 (Modified Harvard Architecture)**.** Most modern computers act as *both* von Neumann Architecture machines and Harvard Architecture machines. These have been called *modified Harvard architectures*. The *modified Harvard architecture* is also a variation of the Harvard Architecture that allows the contents of the instruction memory to be accessed as data. The different types of modified Harvard architectures are discussed in Remark 3.2.

Remark 3.1 (Modern CPU Architecture)*.* In modern CPUs, with both their system memory and on-chip cache, they act as both von Neumann Architecture machines and Harvard Architecture machines. The CPU acts as:

- A Harvard Architecture machine when the CPU is accessing its on-chip cache.
- A von Neumann Architecture machine when the CPU is accessing the system memory.

Remark 3.2 (Types of Modified Harvard Architectures)*.* There are many different types of Modified Harvard Architectures. Some of the major ones are discussed here:

- Split-cache (or almost-von Neumann Architecture architecture)
  - The most common modification builds a memory hierarchy with a CPU cache separating instructions and data.
  - This unifies all except small portions of the data and instruction address spaces, providing the von Neumann model.
- Instruction-Memory-as-Data Architecture
  - Another change preserves the "separate address space" nature of a Harvard Architecture machine, but provides special machine operations to access the contents of the instruction memory as data.
  - Because data is not directly executable as instructions, there are 2 different operations possible:
    1. Read access: initial data values can be copied from the instruction memory into the data memory when the program starts. Or, if the data is not to be modified (it might be a constant value, such as pi, or a text string), it can be accessed by the running program directly from instruction memory without taking up space in data memory (which is often at a premium).
    2. Write access: a capability for reprogramming is generally required; few computers are purely ROM-based. For example, a microcontroller usually has operations to write to the flash memory used to hold its instructions. This capability may be used for purposes including software updates. EEPROM/PROM replacement is an alternative method.
- Data-Memory-as-Instruction Architecture
  - A few Harvard Architecture processors can execute instructions fetched from any memory segment
  - Unlike the original Harvard processor, which can only execute instructions fetched from the program memory segment.
  - Such processors, like other Harvard Architecture processors, and unlike pure von Neumann Architecture, can read an instruction and read a data value simultaneously, **if they're in separate memory segments**, since the processor has (at least) two separate memory segments with independent data buses.
  - The most obvious programmer-visible difference between this kind of modified Harvard architecture and a pure von Neumann architecture is that – when executing an instruction from one memory segment – the same memory segment cannot be simultaneously accessed as data.

The von Neumann Architecture models variables incredibly well, as memory cells, assignment statements as the writing of data back to memory, and iteration. In fact, the von Neumann Architecture models iteration so well, that it encourages iteration over recursion (when possible), sometimes at the detriment of the overall program.

1.1.2 Programming Design Methodologies

Starting in the 1960s, bigger and more complicated programs were being written for more complicated things (controlling whole facilities, worldwide airline reservation systems, etc.). New software development methodologies appeared, and a shift from procedure-oriented to data-oriented design methodologies emerged.

Data-oriented models emphasize:

- Data design
- Abstract data types to solve problems

This data-oriented design led to the to development of object-oriented design.

1.2 Language Categories

There are 3 main categories that languages fall into (that we are considering in this course):

1. Imperative Programming Language

2. Functional Programming Language
3. Logical Programming Language

If you want to view all possible language categories, visit Wikipedia's Programming Paradigms.

Defn 4 (Imperative Programming Language)**.** *Imperative programming languages* have a programming paradigm that uses statements that change a program's state. An imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates.

Defn 5 (Functional Programming Language)**.** *Functional programming languages* treat computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm in that programming is done with expressions or declarations instead of statements. In functional code, the output value of a function depends only on its arguments, so calling a function with the same value for an argument always produces the same result.

This is in contrast to Imperative Programming Languages where, in addition to a function's arguments, global program state can affect a function's resulting value. Eliminating side effects, that is, changes in state that do not depend on the function inputs, can make understanding a program easier, which is one of the key motivations for the development of functional programming.

Defn 6 (Logical Programming Language)**.** *Logic programming languages* are a type of programming language which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.

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 7 (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. When a programming language is interpreted, there is **no** translation of the high-level source language to anything else. The *interpreter* uses the high-level source code directly. 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

Interpretation allows for easy implementation of source-level debugging. Meaning when semantic analysis occurs on the program while running, the errors are returned in a fashion that makes sense with relation to the high-level language. For instance, if there is an array index error, the error could refer to the index itself, the name of the array, and its line.

Remark 7.1 (Interpretation Drawbacks). There is roughly a 10–100 times performance slowdown. The main bottleneck in an interpreted language is the instruction decoding from the high-level source to something the interprett can use. This is because *every* instruction must be decoded *every* time.

Additionally, interpreted programs take up more space on disk in a form not designed to be space efficient. In memory, interpreted programs take up more space because the symbol table and interpreter must be in memory at the same time to make the program run.

Some examples of languages with an Interpretation implementation are:

- Python
- Perl
- Ruby
- Bash
- AWK
- ...

2.2 Compilation

Defn 8 (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 9 (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 text in the input file into a set of tokens
2. Syntactic Analysis/Parsing: Convert the tokens into a parse tree representing all the tokens in the program in a hierarchical and prioritative manner
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 typically 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 9.1. 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 10 (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 10.1. This particular program is not discussed heavily in this course.

Defn 11 (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 11.1. This particular program is not discussed in this course.

Defn 12 (Loader)**.** The *loader* is the program provided by the operating system that loads the specified program into main memory and begins execution.

Remark 12.1. This particular program is not discussed in this course.

Some examples of languages with a Compilation implementation are:

- C
- C++
- SML
- Haskell
- FORTRAN
- ···

2.3 Hybrid Implementation

Defn 13 (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.

Typically what happens is the high-level source language translates the source language to an intermediate language that allows for easy interpretation. This is faster because instructions in the source language are only decoded once.

For example, Java does this with their Just-In-Time (JIT) compilation scheme, which translates all instructions to an intermediate language, then translates those to machine code on-the-fly when needed.

Some examples of Hybrid Implementation are:

- Java
- Scala
- C#
- JavaScript
- ···

One way to implement a language with Hybrid Implementation 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.

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.

Some additional criteria that could be used to evaluate programming languages are:

- Portability: Ease with with programs can be moved from one implementation to another

	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

- Generality: The applicability to a wide range of applications
- Well-Definedness: The completeness and precision of the language's official defining document

Some of criteria are given different weightings/importance by different people, thus making each slightly subjective. Additionally, many of these criteria are not precisely defined, nor are they exactly measurable.

3.1 Readability

Defn 14 (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 15 (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 16 (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 17 (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 17.1 (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 What words have been made either Reserved Words or Keywords by the programming language specification?

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

Writability is a measure of how easy it is to write a program in a language for a given problem domain. This is closely related to the language characteristics presented in Section 3.1, Readability. The definition of the problem domain is incredibly important, because C would not be used to make a GUI, and Visual BASIC would not be used to make an operating system.

3.2.1 Simplicity and Orthogonality

Programmers might not know all the features for a language. Or, the might know about them, but use them incorrectly. This means there should be a smaller number of primitive constructs and a consistent set of rules for combining them. This reduces the number of primitive constructs in a language, and allows a programmer to design a complex solution by only using a simple set of primitive constructs. By reducing the orthogonality of a program, the total possible combinations of constructs is reduced, simplifying the process of reading and writing the program.

3.2.2 Support for Abstraction

Defn 18 (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

Expressivity has several characteristics.

1. Verbosity of the language
   - The amount of code needed to describe some computation to the computer.
2. Powerful/Convenient way to specify computations.
   - `count++` vs. `count = count + 1` to increment a value in Java

3.3 Reliability

Reliability is a measure of the program performing to its specifications reliably under all conditions.

3.3.1 Type Checking

Defn 19 (Type Checking)**.** *Type checking* is a process for testing for type errors in a given program, by the compiler or the interpreter, depending on its implementation (Interpretation vs. Compilation). Runtime type checking is expensive, so compile-time checking is preferred.

The earlier that type checking can occur reduces the potential errors, and corrective actions can be taken.

3.3.2 Exception Handling

The programming language should have the ability to intercept runtime errors, along with other unusual conditions, take corrective actions, the continue normally is traditionally called *exception handling*.

3.3.3 Aliasing

Defn 20 (Aliasing)**.** *Aliasing* is having 2 or more distinct names that can be used to access the same memory location. Most languages allow for 2 pointers to point to the same thing in memory, but others prevent this completely.

Sometimes, Aliasing is used to overcome deficiencies in the language's Support for Abstraction. Others greatly restrict possible Aliasing to increase their Reliability.

3.3.4 Readability and Writability

The Readability and Writability greatly influence a program's Reliability. A program written in a language that exceeds the languages original problem domain will use unnatural approaches to solve the problem. These unnatural approaches are less likely to be correct for all possible situations. Thus, the easier a program is to write, the more likely it is to be correct for all possible situations.

Programs that are difficult to read will affect the writing and maintenance phases of the software's life cycle.

3.4 Cost

There are several parts that increase the cost of a programming language.

1. Training programmers in a new language
   - Function of Simplicity and Orthogonality
   - Function of programmer experience
2. Writing software
   - Function of Writability of the language
3. Compilation time
   - Time to compile a program
   - Resources required to compile a program in a language
4. Run time
   - Performance during runtime
   - Dependent on the effort made to optimize the input source code
5. Financial cost of special software
   - The cost of using the Compiler for a language for instance.
   - Languages with free Compilers or interpreters tend to be accepted more quickly than languages with a financial cost
6. Cost of limited reliability
   - Maintenance time
     - Corrections made to errors in the program
     - Modifications to add new functionality
   - Insurance cost, in special cases
     - Airplanes
     - Nuclear power plants
     - X-Ray machines

4 Backus-Naur Form and Context-Free Grammars

In the 1950s, there were 2 men, Noam Chomsky and John Backus, that were working completely separately who were trying to formally describe language. They actually ended up developing very similar answers to that problem.

Remark. Context-Free Grammars are referred to only as grammars throughout this document. Also, the terms BNF (Backus-Naur Form) and grammar are used interchangeably.

4.1 Context-Free Grammars

Chomsky, a linguist, described 4 classes of grammars that define 4 classes of languages, which are given in Table 4.1

There exists a hierarchy for the definition of Grammars that define Languages. It is called the *Chomsky Hierarchy of Formal Grammars*.

Grammar	Rule Patterns	Type
Regular	$X \to aY$ or $X \to a$ or $X \to \epsilon$	3
Context-Free	$X \to \gamma$	2
Context-Sensitive	$\alpha X \beta \to \alpha \gamma \beta$	1
Arbitrary	$\gamma \to \delta$	0

Table 4.1: Chomsky Hierarchy of Formal Grammars

Regular grammars have the same power as regular expressions, meaning they can be used to find tokens in a program.

Remark. Where a is a Terminal Symbol, α, β, γ, and δ are *sequences* of symbols (Terminal Symbols or Nonterminal Symbols).

Type(3) $\subset$ Type(2) $\subset$ Type(1) $\subset$ Type(0)

4.2 Backus-Naur Form

It is important to discuss where Backus-Naur form came from, and how ti has been modified since. Originally, there was the Canonical Form. This only allowed for one production per line, and did not support options, repetition, etc.

Defn 21 (Canonical Form)**.** The *canonical form* of a Context-Free Grammar is the most formal use of a Context-Free Grammar.

$$\begin{aligned} A &\to B\ d\ e\ C\ f \\ A &\to g\ A \end{aligned} \tag{4.1}$$

The Canonical Form is:

- The core formalism for Context-Free Grammars
- Useful for proving properties and explaining algorithms

When John Backus was working on ALGOL 58, his published paper used a new formal notation for specifying programming language syntax. Peter Naur slightly modified Backus's original syntax which developed Backus-Naur Form.

Defn 22 (Backus-Naur Form)**.** The *Backus-Naur form* of a Context-Free Grammar is an extension of the Canonical Form. This form is less formal than the Canonical Form, but is allows for condensation of multiple productions that have the same nonterminal on the left-hand side to the same production. This is done with the | symbol.

For example, Equation (4.2) is equivalent to Equation (4.1).

$$A \to B\ d\ e\ C\ f \mid g\ A \tag{4.2}$$

Backus-Naur Form has some inconveniences, and has been extended to avoid thse issues. These extensions have been formalized and called Extended Backus-Naur Form. Extended Backus-Naur Form will not be used much in this course, but it is a good way to quickly and succinctly express a Context-Free Grammar.

Defn 23 (Extended Backus-Naur Form)**.** The *Extended Backus-Naur form* of a Context-Free Grammar is an extension of the *Backus-Naur Form*. This is a more informal implementation of a Context-Free Grammar. This informality allows for some additional constructs in the Production rules.

These include:

1. Repetition with the Kleene Star (*), or with { repItem }
   - Means that portion of the Production can be repeated 0 or more times.
2. Single Optionals, denoted as (op1 | op2 | ...)
   - Means select one of the options present between the parentheses.
3. Optional portions of the Production, denoted with [op]
   - Means that portion of the Production is an optional part of the entire Production.

The Extended Backus-Naur Form is:

- Compact, easy to read and write
- Common notation for practical use

4.3 Use Today

Defn 24 (Metalanguage)**.** *Metalanguage*s are languages that are used to describe other languages. Context-Free Grammars are one example used as a metalanguage for programming languages.

Defn 25 (Context-Free Grammar)**.** A *context-free grammar* or *CFG* is a way to define a set of *strings* that form a Language. Each string is a finite sequence of Terminal Symbol taken from a finite Alphabet. This is done with one or more Productions, where each production can have both Nonterminal Symbol and Terminal Symbol.

More formally, a Context-Free Grammar is defined as $G = (N, T, P, S)$, where

- N, the set of Nonterminal Symbols
- T, the set of Terminal Symbols
- P, the set of production rules, each with the form

$$X \to Y_1 Y_2 \dots Y_n \text{ where } X \in N, x \geq 0, \text{and } Y_k \in N \cup T$$

- S, the start symbol (one of the Nonterminal Symbols, N). $S \in N$.

Defn 26 (Language)**.** A *language* is the set of **all** strings that can be formed by the Productions in the Context-Free Grammar.

Defn 27 (Production)**.** A *production* is a rule that defines the relation between a single Nonterminal Symbol and a string comprised of Nonterminal Symbols, Terminal Symbols, and the Empty String. These can be though of as abstractions for syntactic structures.

The are denoted as shown below:

$$p_0 : A \to \alpha \tag{4.3}$$

Remark 27.1. There *can* be multiple productions for the same Nonterminal Symbol

Defn 28 (Nonterminal Symbol)**.** A *nonterminal symbol*, or just *nonterminal*, is a symbol that is used in the Context-Free Grammar as a symbol for a Production.

Defn 29 (Terminal Symbol)**.** A *terminal symbol*, or just *terminal*, is a symbol that cannot be derived any further. This is a symbol that is part of the Alphabet that is used to form the Language.

Remark 29.1. These terminals could be tokens defined by a regular grammar or a regular expression. They might just be abstractions of sequences or sets of symbols from the Alphabet.

Defn 30 (Start Symbol)**.** The *start symbol* is a Nonterminal Symbol which is specially designated as the start point of a Derivation for a grammar.

Other than the fact a Derivation starts with this Nonterminal Symbol and its associated Production, it is not special.

Defn 31 (Empty String)**.** The *empty string* is a special symbol that is neither a Nonterminal Symbol nor a Terminal Symbol. The empty string is a *metasymbol*. It is a unique symbol meant to represent the lack of a string. It is denoted with the lowercase Greek epsilon, ϵ or ε.

Defn 32 (Alphabet)**.** The finite set of Nonterminal Symbols that can be used to form a Language.

4.3.1 Multiple Productions on Single Line

This is briefly discussed in Definition 22. What this allows us to do is combine multiple Productions that have the same Nonterminal Symbol on the left-hand side to a single line, or single Production.

For example,

$$\begin{aligned} &< \text{if stmt} > \to \texttt{if}(< \text{logic expr} >) < \text{stmt} > \\ &< \text{if stmt} > \to \texttt{if}(< \text{logic expr} >) < \text{stmt} > \texttt{else} < \text{stmt} > \end{aligned} \tag{4.4}$$

can be combined to

$$\begin{aligned} < \text{if stmt} > \to &\texttt{if}(< \text{logic expr} >) < \text{stmt} > \\ &|\texttt{if}(< \text{logic expr} >) < \text{stmt} > \texttt{else} < \text{stmt} > \end{aligned} \tag{4.5}$$

4.3.2 Describing Lists

A Production is recursive if its left-hand side Nonterminal Symbol appears somewhere ont he right-hand side. This recursive property is useful for constructing variable-length lists.

This is a small extension of using Multiple Productions on Single Line.

$$\begin{aligned} < \text{ident list} > \to &\text{identifier} \\ &|\text{identifier}, < \text{ident list} > \end{aligned} \tag{4.6}$$

4.3.3 Grammars and Derivations

Defn 33 (Derivation)**.** A *derivation* is the use of Production applications to parse a given input string. Example 4.1 demonstrates this.

Example 4.1: Left-Most Derivation.

Perform a Leftmost Derivation of the sentence

$$\texttt{begin}A = B + C; B = C\texttt{end}$$

With the grammar

$$\begin{aligned}
&< \text{program} > \rightarrow \texttt{begin} < \text{stmt list} > \texttt{end} \\
&< \text{stmt list} > \rightarrow < \text{stmt} > \\
&\qquad\qquad\qquad\quad | < \text{stmt} > ; < \text{stmt list} > \\
&\qquad < \text{stmt} > \rightarrow < \text{var} > = < \text{expression} > \\
&\qquad\quad < \text{var} > \rightarrow A|B|C \\
&< \text{expression} > \rightarrow < \text{var} > + < \text{var} > \\
&\qquad\qquad\qquad\quad | < \text{var} > - < \text{var} > \\
&\qquad\qquad\qquad\quad | < \text{var} >
\end{aligned}$$

$$\begin{aligned}
< \text{program} > &\Rightarrow \texttt{begin} < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{var} > = < \text{expression} > ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = < \text{expression} > ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = < \text{var} > + < \text{var} > ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + < \text{var} > ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + C ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + C ; < \text{stmt} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + C ; < \text{var} > = < \text{expression} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + C ; B = < \text{expression} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + C ; B = < \text{var} > \texttt{end} \\
&\Rightarrow \texttt{begin } A = B + C ; B = C \texttt{ end}
\end{aligned}$$

In general, Derivations occur from left-to-right, which is one L in the 2 different types of Derivations. Both types of Derivation, Leftmost Derivation and Rightmost Derivation, will yield the same result when a Derivation is successfully completed.

Defn 34 (Leftmost Derivation)**.** *Leftmost derivation*, or *LL* derivation, stands for *left-to-right leftmost derivation*. Starting from the left of the sentence, you always derive the left-most Nonterminal Symbol, until you reach a Terminal Symbol. Once all symbols present in the sentence are Terminal Symbol, you are done.

Defn 35 (Rightmost Derivation)**.** *Rightmost derivation*, or *LR* derivation, stands for *left-to-right rightmost derivation*. Starting from the left of the sentence, you always derive the right-most Nonterminal Symbol, until you reach a Terminal Symbol. Once all symbols present in the sentence are Terminal Symbol, you are done.

Example 4.2: Right-Most Derivation.

Perform a Rightmost Derivation of the sentence

$$\texttt{begin}A = B + C; B = C\texttt{end}$$

With the grammar

$$
\begin{aligned}
&< \text{program} > \rightarrow \texttt{begin} < \text{stmt list} > \texttt{end} \\
&< \text{stmt list} > \rightarrow < \text{stmt} > \\
&\qquad\qquad\qquad\quad | < \text{stmt} > ; < \text{stmt list} > \\
&\qquad < \text{stmt} > \rightarrow < \text{var} > = < \text{expression} > \\
&\qquad\quad < \text{var} > \rightarrow A|B|C \\
&< \text{expression} > \rightarrow < \text{var} > + < \text{var} > \\
&\qquad\qquad\qquad\quad | < \text{var} > - < \text{var} > \\
&\qquad\qquad\qquad\quad | < \text{var} >
\end{aligned}
$$

$$
\begin{aligned}
< \text{program} > &\Rightarrow \texttt{begin} < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; < \text{stmt list} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; < \text{stmt} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; < \text{var} > = < \text{expression} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; < \text{var} > = < \text{var} > \texttt{end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; < \text{var} > = C \texttt{ end} \\
&\Rightarrow \texttt{begin} < \text{stmt} > ; B = C \texttt{ end} \\
&\Rightarrow \texttt{begin} < \text{var} > = < \text{expression} > ; B = C \texttt{ end} \\
&\Rightarrow \texttt{begin} < \text{var} > = < \text{var} > + < \text{var} > ; B = C \texttt{ end} \\
&\Rightarrow \texttt{begin} < \text{var} > = < \text{var} > + C ; B = C \texttt{ end} \\
&\Rightarrow \texttt{begin} < \text{var} > = B + C ; B = C \texttt{ end} \\
&\Rightarrow \texttt{begin } A = B + C ; B = C \texttt{ end}
\end{aligned}
$$

4.3.4 Parse Trees

Parse trees are hierarchical structures that describe the same information as a Derivation in a visual format. Every internal node is labeled with a Nonterminal Symbol and every leaf is labeled with a Terminal Symbol

4.3.5 Ambiguities

Defn 36 (Ambiguous)**.** A Context-Free Grammar is said to be *ambiguous* or has *ambiguities* if there is more than one way to derive the same string in a grammar.

The grammar below is ambiguous because there are multiple ways to parse the string: "`statement;statement;statement`".

$$
\begin{aligned}
&p_0 : \text{start} \rightarrow \text{program } \$ \\
&p_1 : \text{program} \rightarrow \text{statement} \\
&p_2 : \text{statement} \rightarrow \text{statement ";" statement} \\
&p_3 : \text{statement} \rightarrow \text{ID "=" INT} \\
&p_4 : \text{statement} \rightarrow \epsilon
\end{aligned}
\tag{4.7}
$$

4.3.5.1 Dangling `if-then-else` `if-then-else` statements are usually defined to have an `else` clause, that when present, matches with the nearest previous unmatched `then`. This can be represented with the Productions shown in Equation (4.8).

$$
\begin{aligned}
< \text{stmt} > &\rightarrow < \text{matched} > \ | \ < \text{unmatched} > \\
< \text{matched} > &\rightarrow \texttt{if} < \text{logic expr} > \texttt{then} < \text{matched} > \texttt{else} < \text{matched} > \\
&\quad | \text{ any non-if statement} \\
< \text{unmatched} > &\rightarrow \texttt{if} < \text{logic expr} > \texttt{then} < \text{stmt} > \\
&\quad | \texttt{ if} < \text{logic expr} > \texttt{then} < \text{matched} > \texttt{else} < \text{unmatched} >
\end{aligned}
\tag{4.8}
$$

4.3.6 Operator Precedence

To handle the precedence of operators, we need to define a "priority level" to our Productions. It is good to note that the further *down* an expression is in the parse tree, the higher its priority in mathematics.

$$\begin{aligned} < \text{assign} > &\rightarrow < \text{id} > = < \text{expression} > \\ < \text{id} > &\rightarrow A \mid B \mid C \\ < \text{expression} > &\rightarrow < \text{expression} > + < \text{multiplicative expression} > \\ &\mid < \text{multiplicative expression} > \\ < \text{multiplicative expression} > &\rightarrow < \text{multiplicative expression} > * < \text{factor} > \\ &\mid < \text{factor} > \\ < \text{factor} > &\rightarrow (< \text{expression} >) \\ &\mid < \text{id} > \end{aligned} \tag{4.9}$$

4.3.7 Operator Associativity

We need to make sure that operators are associated with each other correctly. If we need to make an operator right associative, we just need to flip the terms in Equation (4.9) around. The operators in Equation (4.9) are left associative as they are right now.

$$\begin{aligned} < \text{factor} > &\rightarrow < \text{expression} > ** < \text{factor} > \\ &\mid < \text{term} > \\ < \text{term} > &\rightarrow (< \text{expression} >) \\ &\mid < \text{id} > \end{aligned} \tag{4.10}$$

5 Names

Names or *identifiers* are, obviously, names given to things. They can identify:

- Variables
- Subprograms
- Formal Parameters
- Other program constructs

5.1 Issues

There are 2 questions that need to be asked when designing the names or identifiers possible in a language.

1. Are names case-sensitive? For example, are these identifiers different?
   - `myvariable`
   - `MYVARIABLE`
   - `MyVariable`
   - `myVariable`
2. Are the special words of the language Reserved Words or are they Keywords?

5.2 Name Forms

How is a name/identifier defined?

- Is there a character limit on the identifier/name?
- Are all characters in the identifier/name significant?
- What characters are allowed in the identifier/name?
- Are there special characters required by a language?
  - `$` being required in front of identifiers in PHP
  - `$`, `@`, `%` specifying a "type" in Perl
  - `@` and `@@` to denote an instance or class variable in Ruby, respectively

Some languages are *case-sensitive*. C, Java, C++, etc. would all treat `rose`, `ROSE`, and `Rose` differently. This could be a detriment to readability, because names that *look* similar are actually not. In terms of writability, the programmer must remember the exact typecasing of the identifier/name.

5.3 Special Names

There are Reserved Words and Keywords. They are similar in that the programming language specification defines that these words have special meanings when constructing programs. However, the 2 differ in how these words can potentially be reused.

Defn 37 (Keyword)**.** *Keyword*s are words that are defined by the language constructors to have some special meaning. However, it only has these special meanings in ***certain contexts***. This means you can define a keyword as a variable and use it together with the keyword. For example, this is a perfectly valid piece of Fortran code:

```
1 Integer Apple
2 Integer = 4
```

Defn 38 (Reserved Word)**.** *Reserved word*s are words that are reserved by the language constructors because those particular words have a meaning in the language. These words cannot be used as identifiers for **ANYTHING** else. For example:

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

Remark 38.1 (Too Many Reserved Words). The potential problem with Reserved Words is that if a language has a large number of reserved words, the user might have a hard time creating names for themselves. Unfortunately, the most commonly chosen words by programs are usually Reserved Words. For example,

- `LENGTH`
- `BOTTOM`
- `DESTINATION`
- `COUNT`

5.4 Variables

Defn 39 (Variable)**.** A program *variable* is an abstraction of a computer Memory cell or a collection of Memory cells. A variable can be characterized by a sextuple of attributes:

1. Name
2. Address
3. Value
4. Type
5. Storage Bindings and Lifetime
6. Scope

5.4.1 Name

Most Variables have names. These are symbolic references to the value that is actually stored. There are various issues that may arise with the name of a variable, which were discussed earlier.

5.4.2 Address

Defn 40 (Address)**.** The *address* of a Variable is the machine's memory address with which the Variable is associated.

The address of a variable is sometimes called its *L-Value*. This is because the address is required when the name of a variable appears on the left-hand side of an assignment statement.

Remark 40.1 (Alias). An *alias* is having another Variable have the same Address, so the 2 Variables point to the same value in Memory.

For some languages, it is possible for the same Variable to be associated with different addresses at different times during the Variable's lifetime.

5.4.3 Type

Defn 41 (Type)**.** The *type* of a Variable determines the range of values that Variable can store. For example, the `int` type in Java specifies a value range of -2147483648 to 2147483647. It is a 32-bit signed integer.

5.4.4 Value

Defn 42 (Value)**.** The *value* of a Variable is the contents of the Memory cell or cells associated with the Variable. The value of a variable is sometimes called it *R-Value*. This is because the value of the Variable is required on the right-hand side of an assignment statement. To access the r-value, the l-value must be determined first.

Remark 42.1 (Abstract Memory Cells). While in hardware, the individual sizes of Memory are fixed, we can think of Memory as having *abstract memory cells*, that can accomodate anything we attempt to put into Memory. This means that a single-precision floating point number technically takes up 4 bytes, 32 bits, of Memory cells, that number only takes one abstract memory cell.

5.5 Binding

Defn 43 (Binding)**.** A *binding* is an association between an attribute and an entity. This association can be between:

- A variable
  - Its type
  - Its value
- An operation
  - Its symbol

The time at which a binding occurs is called the Binding Time.

Defn 44 (Binding Time)**.** The time at which Binding occurs is called the *binding time*. These include:

- Language Design Time
  - Defining `*` to represent multiplication
- Language Implementation Time
  - Having an `int` in C be a 32-bit signed integer
- Compiler Time
  - The type of a variable in a Java program
- Link Time
  - A call to a library subprogram is bound to the subprogram code
- Load Time
  - Variable bound when loaded into Memory
  - Could happen at run time too
- Run Time
  - Variable bound when loaded into Memory
  - Could happen at compile time too

We need to know the Binding Times for the attributes of a program to understand the semantics of the programming language.

5.5.1 Binding of Attributes to Variables

Defn 45 (Static)**.** A Binding is *static* if the Binding first occurs before run time begins and remains unchanged throughout program execution. An example of this is declaring a Variable as an `int` in C. Throughout the whole C program, that Variable can only hold signed 32-big integers.

```
int x = 4;
float x = 4.0; // Error here, x already declared
x = 4.0 // Error here, x is of int type
```

Defn 46 (Dynamic)**.** A Binding is *dynamic* if the Binding occurs during run time, or can change in the course of program execution. An example of this is declaring a Variable in Python.

```
x = 4
x = [1, 2, 3]
x = 'dynamically bound string'
```

All three lines have a variable declaration, where the Binding occur during the program's execution and changed during it.

We are only concerned with the distinction between Static and Dynamic Variable Binding. Meaning, we will ignore how hardware may bind and unbind things repeatedly when it is switching and moving things around.

5.5.2 Type Bindings

Before a Variable can be used or referenced in a program, its Type must be declared. A Variable's *type* determines the range of values that can be stored in the Variable. In a more abstract sense, it also determines what kind of operations make sense and are possible to use on these Variables. There are 2 important aspects of this Binding:

1. How the Variable Type is specified
2. When the Binding takes place

There are 2 ways to bind Types to Variables:

1. Static Type Binding
   - Explicit
   - Implicit
2. Dynamic Type Binding

5.5.2.1 Static Type Binding

Defn 47 (Static)**.** *Static* Binding of Variables means that the Type of a Variable is given to the program, either Explicitly or Explicitly. Once the Type is declared, it cannot be changed throughout the entire program's execution.

There are 2 ways to Staticly bind a Type to a Variable:

1. Explicit
2. Implicit

Defn 48 (Explicit)**.** An *explicit* declaration is a statement that explicitly sets each Variable to its respective Type. For example,

```
int x = 0;
float x = 0.0;
char x = 'x';
```

Defn 49 (Implicit)**.** An *implicit* declaration associates Variables with Types through default conventions. The first appearance of a Variable name is its implicit declaration.

Implicit declarations are handled by the language processor (Compiler or Interpreter). There are several ways to have implicit declarations work, some of which are:

- Naming conventions
  - In **Fortran**, if an identifier starts with
    - `I`, `J`, `K`, `L`, `M`, or `N`, or their lowercase versions, it is Implicitly declared to be an `Integer` type.
    - Otherwise, it is Implicitly declared to be a `Real` type.
  - In **Perl**, an identifier must be preceded by a special character denoting the Type. This method forms separate namespaces for each Variable Type.
    - `$`, is a scalar. This holds numbers and strings
    - `@`, is an array.
    - `%`, is a hash structure.
    - The separate namespaces means that all 3 of these variables are considered unique, and potentially unrelated.
      - `$apple`
      - `@apple`
      - `%apple`
- Context or type inference
  - In **C#**, a `var` declaration for a Variable must include an initial value, which determines the Type of the Variable.

    ```
    var sum = 0;
    var total = 0.0;
    var name = "Fred";
    ```

  - `sum`, `total`, and `name` are an `int`, `float`, and `string`, respectively.

Remark. Both Explicit and Implicit declarations create Static Bindings to Types.

5.5.2.2 Dynamic Type Binding

With Dynamic Type Binding, the Type of a Variable:

- Is not specified by a declaration statement
- Cannot be determined by the spelling of the Variable's name

Defn 50 (Dynamic)**.** A *dynamic* Binding happens when a Variable is bound to a Type **when it is assigned a Value.** Such an assignment might also bind the Variable to an Address.

Any Variable can be assigned any Type. A Variable's Type can be changed any number of times during program execution.

The name of the Variable is bound to the Variable, then the Variable is bound to a Type and given its Value.

The primary benefit of having Dynamic Binding is the programming flexibility it provides. The 2 major disadvantages are:

1. Programs are less reliable, because error-detection of the compiler/interpreter is diminished relative to a compiler/interpreter for a language with Static Type Bindings.
2. The Cost is quite high because of the Type Checking that must occur at run time. Also, every variable must have a run-time descriptor to describe the Variable's current Type.

Remark. Dynamic Type Binding is usually implemented with Interpretation. This is because:

- The overall Cost of Type handing is hidden by the Cost of the interpreter.
- The Type of an operation's operands must be known to translate the instruction to the correct machine code instruction, which isn't possible with Dynamic Type Binding.

5.5.3 Storage Bindings and Lifetime

The Memory cell to which a Variable is bound must be pulled from the pool of available Memory. The act of binding the Value to a Variable is called Allocation. The act of unbinding is called Deallocation.

Defn 51 (Allocation)**.** *Allocation* is the act of binding a Value to a Memory cell for a Variable.

Defn 52 (Deallocation)**.** *Deallocation* is the process of placing a Memory cell that has been unbound from a variable back into the pool of available Memory.

Defn 53 (Lifetime)**.** The *lifetime* of a Variable is the time in which the Variable is bound to a Memory cell. The lifetime of a Variable starts when it is bound to a cell and ends when it has been unbound from that cell.

We will split the discussion of scalar Variables into 4 categories, according to their lifetimes.

- Static Variables
- Stack-Dynamic Variables
- Explicit Heap-Dynamic Variables
- Implicit Heap-Dynamic Variables

5.5.3.1 Static Variables

Defn 54 (Static Variable)**.** *Static variable*s are those that are bound to Memory cells before program execution begins and remain bound until the program terminates.

Static Variables can be used as globally accessible Variables, or ensure that subprograms are history-sensitive.

The pros and cons of Static Variables are:

- Pros
  - Efficiency. All Memory addressing is done with absolute addresses, making things very fast.
- Cons
  - Reduced flexibility. If there is a language that only has Static Variables, then recursive subprograms are impossible.

Remark. In C and C++, `static` can be set on functions, making the Variables declared in the function Static Variable.

Remark. In Java, C++, and C#, `static` can appear on classes, meaning class Variables are created statically some time before the class is first instantiated.

5.5.3.2 Stack-Dynamic Variables

Defn 55 (Stack-Dynamic Variable)**.** *Stack-dynamic variable*s are those whose storage Bindings are created when their declaration statements are elaborated. These are allocated from the run-time stack.

Remark 55.1. These are the variables that are most commonly used.

Remark 55.2. In languages that allow for variable declaration anywhere in the function, like Java and C++, the Stack-Dynamic Variables may be bound to storage at the beginning of the block. In these cases, the Variable becomes visible at the declaration, but the storage Binding occurs when the block begins execution.

Defn 56 (Elaboration)**.** *Elaboration* of a Variable declaration refers to the storage Allocation and Binding process indicated by the declaration, which takes place when execution reaches that code.
This occurs at run time.

The advantages and disadvantages of Stack-Dynamic Variables, compared to Static Variables, are:

- Advantages
  - Allows for recursive subprograms that have local variables
  - All subprograms can share the same memory space for their locals, allowing for a smaller memory footprint, by only having some variables bound to storage at once.
- Disadvantages
  - Runtime overhead of Allocation and Deallocation.
  - Slower accessing of Stack-Dynamic Variables because of indirect addressing.
  - Subprograms cannot be history-sensitive with just Stack-Dynamic Variables.

5.5.3.3 Explicit Heap-Dynamic Variables

Defn 57 (Explicit Heap-Dynamic Variable)**.** *Explicit heap-dynamic variables* are namess (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These Variables are allocated to and deallocated from the Heap. They can only be referenced through pointers or reference variables.
The pointer/reference can only be created and returned by:

- An operator (in C++), `new`
- A subprogram (in C), `malloc`

Some languages include ways to destroy these pointers/references:

- An operator (in C++), `delete`
- A subprogram (in C), `free`

An example of an Explicit Heap-Dynamic Variable is shown below.

```
1 int *intnode; // Create a pointer
2 intnode = new int; // Create the heap-dynamic variable
3 ...
4 delete intnode; // Deallocate the heap-dynamic variable to which intnode points
```

The Heap is highly disorganized because of the unpredictability of its use. Garbage collection can help organize it.
There are 2 ways to manage the Heap.

1. Explicit Deallocation
   - The programmer must explicitly free the Memory themselves.
2. Implicit Deallocation
   - The programming language has facilities, called *garbage collection* that automatically manages the Heap.

The advantages and disadvantages of these types of Variables are:

- Advantages
  - Explicit Heap-Dynamic Variable are often used to construct dynamic data structures, like linked lists and trees. These are built conveniently using pointers and data.
- Disadvantages
  - The difficulty of using pointer/reference variables correctly
  - The Cost of using these pointers/reference variables
  - Complexity of the required storage management implementation (Although, this is a question of Heap management, which is costly and complicated).

5.5.3.4 Implicit Heap-Dynamic Variables

Defn 58 (Implicit Heap-Dynamic Variable)**.** *Implicit heap-dynamic variable*s are bound to Heap storage onnly when they are assigned values.

The advantages and disadvantages of these types of Variables are:

- Advantages
  - Highest degree of flexibility, allowing for highly generic code
- Disadvantages
  - Run time overhead of maintaining all the dynamic attributes, which could include subscript types and ranges
  - Loss of some error dectection by the compiler/interpreter

5.6 Scope

Defn 59 (Scope)**.** The *scope* of a Variable is the range of statements in which the Variable is Visible.

Defn 60 (Visible)**.** A Variable is *visible* in a statements if it can be referenced in that statement.

Defn 61 (Local Variable)**.** A Variable is a *local variable* in a program unit or block if it is declared there.

Defn 62 (Nonlocal Variable)**.** The nonlocal variables of a program are visible with that particular program unit or block, but are not declared there.

Remark 62.1 (Global Variables)*.* Global variables are a special case of Nonlocal Variables. These are discussed in Section 5.6.4.

5.6.1 Static Scope

Defn 63 (Static Scoping)**.** *Static scoping* is a way to statically determine the scope of a Variable. When there is a reference to a Variable, the attributes of the Variable can be determined by finding the statement in which it is declared (either explicitly or implicitly). This makes it easy for a human reader and compiler/interpreter to figure out the Type of every Variable in the program.

There are 2 types of statically-scoped languages:

1. Subprograms can be nested inside programs (Python)
2. Subprograms cannot be nested inside programs (Java)

Remark 63.1 (Lexical Scoping)*.* Static Scoping is sometimes called *lexical scoping*.

Static Scoping creates a tree-like structure, where each Variable declared in a program unit/block has a Static Parent. Then, each Static Parent has a list of Static Ancestors.

Defn 64 (Static Parent)**.** If the Variable referenced is not present as a Local Variable, then we have to go to the next outer program unit or block, the *static parent*.

Defn 65 (Static Ancestor)**.** The *static ancestor*s are all the Static Parents to that particular program unit/block.

This is illustrated by finding `x` in `sub2()` in this JavaScript function.

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

The `x` in `sub2()` refers to the `x=3` in `big()`, because `sub1()` is not a Static Ancestor of `sub2()`. However, inside of `sub1()`, the use of the variable `x` would refer to the `x=7` value, and never the `x=3` value.

In some languages, if a Variable is declared in a sub-program unit/block, like in `sub1()`, then preceding the Variable name with the outer program unit/block will give the outer Variable Value.

5.6.2 Blocks

New Static Scopes can be defined in the middle of executing code. This allows a small section to have its own Local Variables.

Defn 66 (Block)**.** A *block* is a section of code that has its own Local Variables. These Local Variables are **not** shared with any Static Ancestors.

The use of Blocks create a Block-Structured Language.

Defn 67 (Block-Structured Language)**.** The use of Blocks to create the Static Scopes creates a *block-structured language*.

Consider the following C function:

```
void sub() {
  int count;
  ...
  while (...) {
    int count;
    count++;
  }
  ...
}
```

The `count` inside the `while` loop is that loop's local count, and does not reference the `count` in `sub`.

5.6.2.1 Blocks in Functional Languages

Since Variables in Functional Programming Languages actually evaluate and store expressions, they behave differently. Each functional language handles this differently, so you will have to look at the language specification to find out exactly how Variables are scoped.

5.6.3 Declaration Order

Some languages require that all Variable declarations occur at the beginning of a function (C89).

In some languages, Variables cannot used before they have been declared.

- Some of these languages allow for Variables to be declared anywhere in the function, but can only be referenced **after** their declaration until the end of their scope.
- Some of these languages allow for Variables to be declared anywhere in the function, but if used before their declaration, they use a value like `undefined` (JavaScript).

This is a highly language-dependent thing, and one must consult with the language specification to figure out exactly how it works.

5.6.4 Global Scope

Defn 68 (Global Variable)**.** These are usually Variables that sit outside of all functions. They can be accessed from anywhere in the program. They can also be defined and/or declared in other files in the program's project.

It important to note the difference between a Global Variable's declaration and definition.

- Declaration: The Types and attributes are bound, but the Memory space required is **not** allocated.
- Definition: The Types and attributes are bound, but the Memory space required **is** allocated.

Remark. This is a highly language-dependent thing, and one must consult with the language specification to figure out exactly how it works.

5.6.5 Dynamic Scope

Defn 69 (Dynamic Scoping)**.** *Dynamic scoping* is based on the calling sequence of subprograms, and not their spatial relationship to each other. Thus, the scope can only be determined at run time.

Some languages that implement this are:

- APL
- SNOBOL4
- Early LISP
- Perl (Allowed, but must be said explicitly)

- Common LISP (Allowed, but must be said explicitly)

To illustrate Dynamic Scoping, look at the code block below, and assume it is in a language that uses Dynamic Scoping.

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
  sub2();
}
```

The call to `sub1()` by `big()`, which then calls `sub2()`. In that running of `sub2()`, the use of `x` cannot be determined locally (within that function). Thus, it goes to its dynamic parent, `sub1()` and finds a declaration for `x`. Thus, the `y` in `sub2()` evaluates to `y=7`.

The next instruction, the call to `sub2()` by `big()`, produces a different result. In that running of `sub2()`, the use of `x` cannot be determined locally (within that function). Thus, it goes to its dynamic parent, `big()` and finds a declaration for `x`. Thus, the `y` in `sub2()` evaluates to `y=3`.

A Computer Components

A.1 Central Processing Unit

Defn A.1.1 (Central Processing Unit)**.** The *Central Processing Unit*, *CPU*, is a chip that performs all actions in the computer. It calculates mathematical and logical values and acts based on them. It has several components built onto it, and can be thought of as the "brain" of the computer.

The design of a CPU determines some of the functionality it has. Therefore, more specialized processors can be made for special tasks, and more general processors can be built to handle a wide variety of calculations.

A.1.1 Registers

Defn A.1.2 (Register)**.** A *register* is a data storage mechanism built directly onto the Central Processing Unit. It is several hundred times faster than the system Memory. Registers are generally used when the currently running program is performing calculations. Since they are so fast, they are used as both source and destination operands in instructions.

Remark A.1.2.1. Depending on the Central Processing Unit architecture, there may be cases when Registers behave slightly differently between processors. This is something that can only be found by checking the Central Processing Unit manufacturer's documentation.

A.1.2 Program Counter

A.1.3 Arithmetic Logic Unit

A.1.4 Cache

A.2 Memory

Defn A.2.1 (Memory)**.** *Memory*, or *RAM* (*Random Access Memory*), is a Volatile data storage mechanism. It is directly connected to the Central Processing Unit. This is the location that the Central Processing Unit writes to when it cannot or should not store something in the Central Processing Unit's Registers.

Remark A.2.1.1 (Volatility). Memory is volatile because each of the cells is a small capacitor. In between the clock cycles on the Central Processing Unit and Memory, the capacitors discharge. On the clock cycle, the capacitors are refreshed with electrical power, which does one of 2 things:

1. Keep the data bits the same, 1 to 1.
2. Update the data bits from 0 to 1.

Defn A.2.2 (Volatile)**.** If a data storage mechanism is called *volatile*, it means that once the storage mechanism loses power, the data is lost. This is in contrast to Non-Volatile data storage mechanisms.

A.2.1 Stack

Defn A.2.3 (Call Stack)**.** The *call stack* is an imaginary construct that resembles the traditional stack data structure. It is a way to visualize and organize the way memory is used during the execution of a program and its function/methods. It is filled with Stack Frames. This **does not** hold the code that is used in the function, rather it is everything that is needed for the function to be able to run.

Defn A.2.4 (Stack Frame)**.** A *stack frame*, *activation frame*, or *activation record* are objects that represent necessary portions of a function. These include:

- A Dynamic Link
- Local Variables
- Temporary Variables
- Static Links
- Function Arguments
- Return Address

Additionally, there are 2 registers used as pointers to move around and interact with the stack frame.

1. `FP` is in register `%rbp`. It is the Frame Pointer.
2. `SP` is in register `%rsp`. It is the Stack Pointer.

Defn A.2.5 (Dynamic Link)**.** The *dynamic link* or *dynlink* is a memory address pointer and sits at the bottom of a Stack Frame. It is a pointer back to the previous function's dynamic link. This ensures that any function can find its parent/calling function.

The dynamic link also serves as a means to access any variable that might be needed by this function. To access any variable in *this* function, you can subtract a byte multiple that you need to access the proper value. To access any variable in the calling function, you can add a byte multiple that would correspond to the proper variable.

In the `x86_ 64` architecture, instruction set, and convention that we used, the register `%rbp` was the dynamic link.

Remark A.2.5.1. Note: Due to the conventions we used, when accessing arguments passed to the function, we treated them as local variables, just further down in the call stack. This also means that we need to skip over the Return Address block in memory.

Remark A.2.5.2. Because we used the `%rbp` register to store our current Dynamic Link's address, the dynamic link might also be called the *base pointer*.

Defn A.2.6 (Local Variable)**.** *Local variables* are handled very simply. They get an appropriate amount of memory allocated to them on the stack, and that is it.

There is no way to give a variable a name in assembly. (Usually. Depends on the architecture and instruction set). However, there is no way to name something in memory. But, because the size of all the objects is known at compile-time, allocating the proper amount of memory required by each variable is possible.

Remark A.2.6.1. This holds true for strongly-typed, static, compiled languages, like Java, C, C++, etc. However, Python is slightly different in this regard, and handles it differently.

Defn A.2.7 (Temporary Variable)**.** A *temporary variable* is one that is allocated on this function's Stack Frame while calculating values. Once the calculations are completed, these values are deallocated. These temporary variables can also point to objects on the Heap. When the function has finished running, then these values are deallocated, along with all other Local Variables in use.

For example, since assembly-level addition only allows for 2 operands, but in general, addition can have more than 2 operands in use, there needs to be a way to store the value used in the addition. While we can accumulate and use that value in the addition, the values being added together are *not* modified.

Defn A.2.8 (Static Link)**.** The *static link* is an implicit argument, meaning it ***ALWAYS*** gets pushed onto the stack as a Function Argument, when appropriate.

Appropriate in this context could mean several things:

- When an object is alive, and when it is being acted on by a function.
  - In this case, the static link points to an instance of a class on the Heap.
- When a language allows for nested function declarations.
  - Then the static link points to the Dynamic Link of the outer function
  - This allows access to the outer function's Local Variables like normal, and allows us to go back later.

Defn A.2.9 (Function Argument)**.** *Function argument*s are handled very simply. If a function call takes an argument, then the argument is calculated, and then that argument is pushed onto the stack in *this* Stack Frame.

Remark A.2.9.1. If more than one argument is passed to a function, there are 2 ways to push values onto the Stack Frame:

1. In the order they are passed to the function
   - Say a function with 3 arguments is called, then the stack would have arguments in this order
     (a) argument0 (Lowest memory address)
     (b) argument1
     (c) argument2 (Highest memory address)
2. In reverse order
   - Say the same function is called with the same 3 arguments, then the stack would have arguments in this order
     (a) argument2 (Lowest memory address)
     (b) argument1
     (c) argument0 (Highest memory address)

When the values that were passed need to be accessed, and if the memory sizes of things are known at compile time, then we can calculate how far down we need to go in the stack to find the value. This is done by adding a positive value to the `%rbp`

Defn A.2.10 (Return Address)**.** The *return address* is used by the `%rip` register. It is calculated and pushed onto the Stack Frame stack when the `CALL` macro is used. It is the thing that allows us to jump around in the code from the `text` area of our program.

Remark A.2.10.1. The `%rip` register is the *register instruction pointer*. It holds the value of the *next* instruction to execute. Technically, it is the Program Counter's value.

Defn A.2.11 (Garbage Collection)**.** *Garbage collection* is the act of deallocating objects that may still be on the heap and organizing the heap. Since the heap allocates continuous "blocks" of memory required by an object, the heap may have the necessary memory to allocate an object, but in discontinuous locations.

Defn A.2.12 (Frame Pointer)**.** The *frame pointer* is a pointer that ***ALWAYS*** points to the current function's Dynamic Link. The frame pointer is commonly abbreviated as *FP*. This is the thing that allows us to access Local Variables, Function Arguments, and everything else inside of the Stack Frame. The value is held in the `%rbp` register.

Defn A.2.13 (Stack Pointer)**.** The *stack pointer* is a pointer that ***ALWAYS*** points to the top of the current Stack Frame. The stack pointer is commonly abbreviated as *SP*. This value is held in the `%rsp` register. This is the pointer that allows us to push and pop onto this function's Stack Frame.

Defn A.2.14 (Class Descriptor)**.** The *class descriptor* is a portion of memory set aside for the methods that are in an object.

A.2.2 Heap

Defn A.2.15 (Heap)**.** The *heap* is a memory organization construct that helps visualize the way memory is used during the execution of a program. It is ***SIGNIFICANTLY*** larger than the Call Stack.

In an object-oriented language, this is where instances of classes exist. These objects are allocated on a first-come first-serve basis. When the object is allocated, the amount of memory that the object requires is allocated ***in a continuous block***. The problem is that if the heap has been extensively used, then there might be enough total memory required to allocate an object, but because it is not in a continuous block, the object cannot be allocated.

This makes deallocation more complex, because when they are deallocated, the program might still view the memory as in-use. Additionally, because of the continuous block allocation nature of the heap, it must be organized every once in a while. This is called Garbage Collection.

Remark A.2.15.1. In an object-oriented language, like Java, when objects are stored on the heap, **ONLY** their class fields are stored there. The class methods are stored elsewhere, in the Class Descriptor.

A.3 Disk

Defn A.3.1 (Non-Volatile)**.** If a data storage mechanism is called *non-volatile*, it means that once the storage device loses power, the data is still safely stored. This is in contrast to Volatile data storage mechanisms.

A.4 Fetch-Execute Cycle

B History of Programming Languages

B.1 Zuse's Plankalkül

B.2 Pseudocodes

B.3 Fortran

B.4 Functional Programming: LISP

B.5 ALGOL 60

B.6 COBOL

B.7 Timesharing: BASIC

B.8 PL/I

B.9 Early Dynamic Languages: APL and SNOBOL

B.10 Data Abstraction: SIMULA 67

B.11 Orthogonality: ALGOL 68

B.12 ALGOL Descendants

B.13 Logical Programming: Prolog

B.14 Ada

B.15 Object-Oriented Programming: Smalltalk

B.16 Combine Imperative and OOP Features: C++

B.17 Java

B.18 Scripting Languages

B.19 Flagship .NET Language: C#

B.20 Markup/Programming Hybrid Languages

C Trigonometry

C.1 Trigonometric Formulas

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

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

C.2 Euler Equivalents of Trigonometric Functions

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

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

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

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

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

C.3 Angle Sum and Difference Identities

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

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

C.4 Double-Angle Formulae

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

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

C.5 Half-Angle Formulae

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

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

C.6 Exponent Reduction Formulae

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

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

C.7 Product-to-Sum Identities

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

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

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

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

C.8 Sum-to-Product Identities

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

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

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

C.9 Pythagorean Theorem for Trig

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

C.10 Rectangular to Polar

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

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

C.11 Polar to Rectangular

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

D Calculus

D.1 Fundamental Theorems of Calculus

Defn D.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

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

Defn D.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

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

Defn D.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

$$\underset{x}{\operatorname{argmax}}$$

D.2 Rules of Calculus

D.2.1 Chain Rule

Defn D.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,

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

E Complex Numbers

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

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

where

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

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.

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

E.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.

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

Defn E.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.

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

E.1.1 Complex Conjugates of Exponentials

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

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

E.1.2 Complex Conjugates of Sinusoids

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

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

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

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.