EDAP05: Concepts of Programming Languages - Reference Sheet

Karl Hallsby

Last Edited: November 17, 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.

Remark 19.1. The earlier that type checking can occur reduces the potential errors, and corrective actions can be taken.

There are 2 types of type checking possible:

1. Strong Type Checking
2. Weak Type Checking

There are also 2 times when type checking is possible:

1. Static Type Checking
2. Dynamic Type Checking

Defn 20 (Strong Type Checking)**.** *Strong type checking* is a property of a programming language that states that any language implementation finds all type errors and prevents the operations that caused the type error from taking place.

A language that has strong type checking is called a *strongly-typed language*. Such languages are:

- Haskell
- Python
- Ruby
- JavaScript

Remark 20.1 (Cost). Strong Type Checking improves the Reliability of a language, at a Cost to the language's Expressivity, and sometimes also at a Cost to the language's execution time.

Usually this improvement to Reliability is worth it. However, sometimes it is necessary to write directly to memory without regard to the Data Type system. These languages provide "backdoors" to do this.

Defn 21 (Weak Type Checking)**.** *Weak type checking* is the absence of Strong Type Checking.

A language that has weak type checking is called a *weakly-typed language*. Such languages are:

- C
- C++

Defn 22 (Static Type Checking)**.** Any Type Checking that is performed before runtime (at compile time, as part of the static semantics) is called *static type checking*.

Languages that perform static type checking are called *statically typed language*s.

Remark 22.1 (Undecidability of Static Type Checking). Not all Data Types can be checked statically. For example, the subscript used in an array access must be checked dynamically.

This deference is actually **required** for a language to be strongly-typed.

Defn 23 (Dynamic Type Checking)**.** Any Type Checking that is performed at runtime is called *dynamic type checking*.

Languages that perform dynamic type checking are called *dynamically typed language*s.

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 24 (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	$\langle \mathrm{X} \rangle \to a\langle \mathrm{Y} \rangle$ or $\langle \mathrm{X} \rangle \to a$ or $\langle \mathrm{X} \rangle \to \epsilon$	3
Context-Free	$\langle \mathrm{X} \rangle \to \gamma$	2
Context-Sensitive	$\alpha\langle \mathrm{X} \rangle\beta \to \alpha\gamma\beta$	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 25 (Canonical Form)**.** The *canonical form* of a Context-Free Grammar is the most formal use of a Context-Free Grammar.

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

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 26 (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).

$$\langle \text{A} \rangle \to \langle \text{B} \rangle \; d \; e \; \langle \text{C} \rangle \; f \mid g \; \langle \text{A} \rangle \tag{4.2}$$

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 27 (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 28 (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 29 (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

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

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

Defn 30 (Language)**.** A *language* is the set of **all** strings that can be formed by the Productions in the Context-Free Grammar.

Defn 31 (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:

$$p_0 : \langle \text{A} \rangle \to \alpha \tag{4.3}$$

Remark 31.1. There *can* be multiple productions for the same Nonterminal Symbol

Defn 32 (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 33 (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 33.1. 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 34 (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 35 (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 36 (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 26. 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,

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

can be combined to

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

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.

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

4.3.3 Grammars and Derivations

Defn 37 (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

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

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

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 38 (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 39 (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

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

With the grammar

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

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

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 40 (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`".

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

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).

$$
\begin{aligned}
\langle \text{stmt} \rangle &\rightarrow \langle \text{matched} \rangle \mid \langle \text{unmatched} \rangle \\
\langle \text{matched} \rangle &\rightarrow \texttt{if}\ \langle \text{logic expr} \rangle\ \texttt{then}\ \langle \text{matched} \rangle\ \texttt{else}\ \langle \text{matched} \rangle \\
&\mid \text{any non-if statement} \\
\langle \text{unmatched} \rangle &\rightarrow \texttt{if}\ \langle \text{logic expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle \\
&\mid \texttt{if}\ \langle \text{logic expr} \rangle\ \texttt{then}\ \langle \text{matched} \rangle\ \texttt{else}\ \langle \text{unmatched} \rangle
\end{aligned}
\tag{4.8}
$$

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.

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

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.

$$\begin{aligned} \langle \text{factor} \rangle &\rightarrow \langle \text{expression} \rangle * * \langle \text{factor} \rangle \\ &\quad | \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow (\langle \text{expression} \rangle) \\ &\quad | \langle \text{id} \rangle \end{aligned} \tag{4.10}$$

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 41 (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 42 (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 42.1 (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 43 (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 44 (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 44.1 (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 45 (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 46 (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 46.1 (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 47 (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 48 (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 49 (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 50 (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 51 (Static)**.** *Static* Binding of Variables means that the Type of a Variable is given to the program, either Explicitly or Implicitly before run time begins. 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. Explicitly
2. Implicitly

Defn 52 (Explicit)**.** An *explicit* Static Type Binding is a statement that explicitly sets each Variable to its respective Type. These are statements in a program that lists variable names and specifies that they are of a particular Type. For example,

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

Defn 53 (Implicit)**.** An *implicit* Static Type Binding declaration associates Variables with Types through default conventions, rather than Explicit declaration statements. The first appearance of a Variable name is its implicit declaration.

Remark 53.1 (Effects on Reliability)*.* While Implicit can be helpful for programmers, they can be quite detrimental to Reliability because the compilation process cannot determine some type errors and some programmer errors.

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 54 (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.

2 programming languages that use this are:

1. Python
2. Ruby

Remark 54.1 (Benefits of Dynamic Binding)*.* The primary benefit of having Dynamic Binding is the programming flexibility it provides.

Remark 54.2 (Drawbacks of Dynamic Binding)*.* 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 55 (Allocation)**.** *Allocation* is the act of binding a Value to a Memory cell for a Variable.

Defn 56 (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 57 (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 58 (Static Variable)**.** *Static variable*s are those that are bound to Memory cells before program execution begins and remain bound until the program terminates. They are placed in the "static" section of Memory. 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 59 (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 Call Stack. Thus, when a function on the Call Stack is `return`ed from, all the variables here lose their value.

Remark 59.1. These are the variables that are most commonly used, and are usually function-local variables

Remark 59.2. 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, thus starting the variable's Lifetime. In these cases, the Variable becomes visible at the declaration, but the storage Binding occurs when the block begins execution. So, it is both in Scope and has begun its Lifetime, but has no useful value.

Defn 60 (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 61 (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. There are 2 ways to manage the Heap.

1. Explicit Deallocation
   - The programmer must explicitly free the Memory themselves.
   - C and C++ require this with their `free` and `delete` subprograms/operators, respectively.
2. Implicit Deallocation
   - The programming language has facilities, called *garbage collection* that automatically manages the Heap.
   - There are many algorithms that handle garbage collection, some of which are faster, others slower.

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 and completely separate discussion).

5.5.3.4 Implicit Heap-Dynamic Variables

Defn 62 (Implicit Heap-Dynamic Variable)**.** *Implicit heap-dynamic variable*s are bound to Heap storage **only when they are assigned Values**. In many regards, Implicit Heap-Dynamic Variables and Explicit Heap-Dynamic Variables are quite similar.

However, Implicit Heap-Dynamic Variables have **ALL** their attributes bound **EVERY** time they are assigned.

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

Scope might seem similar to Lifetime, but they are different. Here are 2 examples that illustrate this point:

1. At the second `print(x)`, `x` is *in scope* (visible), but is *dead* (deallocated).

```
int f(void) {
  int *x;
  x = (int *) calloc(sizeof(int), 1);
  print(x);
  free(x);
  print(x);
}
```

2. When executing inside of `g(y)`, the `x` in function `f` is *alive* (allocated), but is *out of scope* (not visible).

```
def f(x):
  return g(7)
def g(y):
  print (y)
  return
```

Defn 64 (Visible)**.** A Variable is *visible* in a statements if it can be referenced in that statement.

Remark 64.1 (In-Scope). Sometimes, a Variable that is Visible is called *in-Scope*.

Defn 65 (Local Variable)**.** A Variable is a *local variable* in a program unit or block if it is declared there.

Defn 66 (Nonlocal Variable)**.** The nonlocal variables of a program are visible with that particular program unit or block, but are not declared there.

Remark 66.1 (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 67 (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 67.1 (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 68 (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 69 (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 70 (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 71 (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 72 (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 73 (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`.

6 Data Types

Defn 74 (Data Type)**.** A *data type* defines a collection of data values and a set of predefined operations on those values. The data types present in a language used for a particular problem should closely mirror the objects in the real-world the program is solving.

They can be mathematically defined as

$$v : \tau \tag{6.1}$$

where v is a value and τ is a type.

Remark 74.1 (Has the Type). To say "v has the type τ"

$$v \in \tau \tag{6.2}$$

User-defined data types allow for:

- Improved readability with better named Data Types.
- Improved modifiability with programmers having to change just one common data type somewhere for a large change throughout a program.

If we take user-defined Data Types further, we end up with *abstract data types*. These force an interface for a particular data type, which is then visible to the user, and the data and background operations are hidden away.

Because of the wide variety of Data Types present today, it is more useful to think about Variables in terms of Descriptor.

Defn 75 (Descriptor)**.** A *descriptor* is the collection of attributes of a Variable. In an implementation, a descriptor is an area of Memory that stores the attributes of a Variable.

There are a 2 cases for these:

1. If all attributes are static, then they are known at compile-time, and the Compiler can use the symbol table to construct everything.
2. If all attributes are dynamic, then the symbol table and all attributes must be stored in Memory during program execution.

Descriptors are used for Type Checking and building the code for Allocation and Deallocation operations.

Defn 76 (Type Error)**.** A *type error* is an attempt to perform an operation that requires an input value of type τ with a value v even though $v : \tau$ does not hold.

Defn 77 (Type Preservation)**.** A type system has the *type preservation* (or *subject reduction*) property if for any $e \Downarrow v$, $e : \tau$ implies $v : \tau$.

There are 3 properties we want to have a type preserving type system to have:

1. *Type Preservation*: The predictions of the type system agree with the evaluation rules.
2. *Progress*: The type system only assigns a type if the evaluation rules will not get "stuck" due to a missing semantic rule. This is not the same as guaranteeing that the program itself terminates, meaning but it does guarantee that the language implementation will never run into a situation in which it doesn't know what to do next.
3. *Termination*: We want the type system to be decidable, that is, we want an automatic mechanism that performs type checking.

6.1 Primitive Data Types

Defn 78 (Primitive Data Type)**.** *Primitive data type*s are Data Types that are **not** defined in terms of other data types. Nearly all programming languages provide these. Some are reflections of hardware, like integers, and others require only little software support for their impmlementation, floating-point numbers for instance.

6.1.1 Numeric Types

This section will discuss the 4 main types of numeric Data Types present in most programming languages.

Defn 79 (Numeric Data Type)**.** *Numeric data type*s are Data Types that handle numbers.

6.1.1.1 Integer

Integers are the most common Numeric Data Type. Many languages support several sizes. Java supports 4: `byte`, `short`, `int`, and `long`. There can be unsigned integers as well.

If there are signed integers, the negative integers are stored in Memory in Twos Complement.

Defn 80 (Twos Complement)**.** *Twos complement* is a way to store negative integers. To find the twos complement:

1. The magnitude of the integer is found in binary
2. The logical complement of that is computed
3. One (1) is added to the logical complement

Remark 80.1. Using Twos Complement is similar to adding by a negative number to an integer instead of performing subtraction.

6.1.1.2 Floating-Point

Defn 81 (Floating-Point)**.** *Floating-point* Data Types model real (fractional/rational) numbers. However, these representations are only approximations for many real values. For example, π cannot be represented in floating-point notation.

Most programming languages implement 2 types of floating-point Data Types.

1. `float`: The standard size, 32 bits (4 bytes). Represent the real number as a decimal and exponent, like scientific notation.
   - The first bit is a *sign bit* (1 for negative)
   - The next 8 bits are for the *exponent*, normalized so that a real number raised to -127, i.e. x^{-127}, has an exponent bit value of 0.
     - This does mean that x^{128} would have a bit-valued exponent of 255.
   - The last 23 bits are for the fraction, called the *mantissa*. This is the fractional portion of the scientific notation, represented in binary, with the first 1 of the bit sequence left off.
2. `double`: Used in cases where larger/smaller fractions or larger/smaller exponents are needed, 64 bits (8 bytes).
   - The first bit is a *sign bit* (1 for negative)
   - The next 11 bits are for the *exponent*, normalized so that a real number raised to -1023, i.e. x^{-1023}, has an exponent bit value of 0.
     - This does mean that x^{1024} would have a bit-valued exponent of 2048.
   - The last 52 bits are for the fraction, called the *mantissa*. This is the fractional portion of the scientific notation, represented in binary, with the first 1 of the bit sequence left off.

Floating-Point numbers are are specified in IEEE Floating-Point Standard 754.

Defn 82 (Floating-Point Precision)**.** *Precision* is the accuracy of the fraction part of the Floating-Point number, and how well it represents the real number's value.

Defn 83 (Floating-Point Range)**.** *Range* is a combination of the range of fractions and the range of the exponents.

6.1.1.3 Complex

Some programming languages support complex numbers natively, and they also support complex-number mathematical operations natively. The imaginary portion of the number is typically denoted with `j` or `J`.

6.1.1.4 Decimal

In a computer, decimal numbers are stored in Binary Coded Decimal. There is also special hardware to support hardware-level mathematical operations on these types of numbers. If this hardware is not present, the calculations can be simulated in software.

Defn 84 (Binary Coded Decimal)**.** *Binary Coded Decimal*, or *BCD*, is a way to represent decimal numbers with perfect accuracy, albeit at the expense of some space. There is a one-to-one mapping of the binary representations of these numbers to decimal, and anything greater than 9 is discarded.

Remark 84.1. These numbers are usually stored 2 per byte, because each only takes 4 bits.

6.1.2 Boolean Types

Defn 85 (Boolean Data Type)**.** *Boolean data type*s only store 2 values: `true` and `false`. Some older language implementations did not support these, but most do today. If a language does not support a boolean data type, then 0 is considered false, and 1 is considered true.

Remark 85.1 (Storage in Memory). Although a single bit can represent a Boolean Data Type, single bits of Memory cannot be efficiently access on many machines. Thus, Boolean Data Types are usually stored in a single byte.

Decimal	Binary Coded Decimal
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
X	1010
X	1011
X	1100
X	1101
X	1110
X	1111

Table 6.1: Binary Coded Decimal

6.1.3 Character Types

Characters are stored in Memory as numeric encodings. These are usually single characters, **not multiple characters together (strings)**.

Characters were originally handled by ASCII, but now there are several encodings, with Unicode being more commonly used now. Unicode supports all human languages, glyphs, and other characters, like emojis. The first 128 characters of Unicode match up with ASCII for intercompatibility.

ASCII required 8 bits, Unicode (UTF-16) uses 16.

6.2 Character String Types

Defn 86 (Character String Type)**.** A *character string type* is one in which the values consist of sequences of characters.

6.2.1 Design Issues

There are 2 questions that need to be answered when designing a language implementation when it comes to strings.

- Should strings be a special kind of character array or a primitive type?
- Should strings have static or dynamic lengths?

6.2.2 Strings and Their Operations

The most common string operations are:

- Assignment
  - What happens when a string is longer than expected? C/C++'s `strcpy` function
- Concatenation
- Substring Reference
  - Discussed more in the context of arrays, where substring references are called slices.
- Comparison
  - How do we compare 2 strings, where one is longer than the other?
- Pattern Matching

In C and C++, strings are terminated with the null character, `00`. This way we do not need to track the length of a string.

Object-Oriented Languages (Java, Ruby, C#) use classes to represent strings. The only field in these objects is a constant string.

Python supports strings as a primitive type, and supports array-like operations on them.

Some languages have Regular Expressions built in, like Perl, JavaScript, Ruby, and PHP. Others have libraries that handle Regular Expressions.

Defn 87 (Regular Expression)**.** A *regular expression*, sometimes called a *regex* is a way to define a sequence of characters to form strings.

6.2.3 String Length Options

Defn 88 (Static Length String)**.** A *static length string* has its length set at the time of string creation. It is static, in that the length cannot be changed later in the program's execution.

Defn 89 (Dynamic Length String)**.** A *dynamic length string* has its length set at the time of string creation. However, strings can change their length, and there is no set maximum size they can have.

Defn 90 (Limited Dynamic Length String)**.** A *dynamic length string* has its length set at the time of string creation. However, the string can be redefined later in the program, so long as the new string is the same length or shorter than when the string Variable was defined.

6.2.4 Evaluation

Primitive string type implementations would require there to be predefined functions for many string operations. If there aren't, then programming in that language becomes more cumbersome.

Dynamic Length Strings are the most flexible, but the overhead of their implementation should be weighed against that flexibility.

6.2.5 Implementation of Character String Types

Software is used to implement string storage, retrieval, and manipulation. When a language uses character arrays to store character string types, the language usually supplies few operations.

A Descriptor for a Static Length String has 3 fields:

1. Name of the type
2. The type's length in characters
3. Address of the first character

A Descriptor for a Limited Dynamic Length String has 4 fields:

1. Name of the type
2. The type's maximum length in characters
3. The length of the currently stored string
4. The address of the first character

A Descriptor for a Dynamic Length String is more difficult to handle because of its dynamic nature. There are 3 approaches to storing these:

1. Strings stored in a linked list. If the string gets longer, individual nodes can be allocated from anywhere in the Heap.
   - A drawback of this is that extra storage of the links
   - The necessary complexity of string operations
2. Store strings as arrays of pointers to individual characters on the Heap
   - This uses more memory, but processing is faster than the linked list approach.
3. Store complete strings in adjacent cells, and when a new longer string comes along, store the whole thing in a new area in the Heap and deallocate the old location.
   - Less storage required compared to the linked list approach
   - Allocation and deallocation of the string is more difficult

6.3 User-Defined Ordinal Types

Defn 91 (Ordinal Type)**.** An *ordinal type* is a Type in which the range of possible values can be associated with the set of positive integers.

For example, in Java, the primitive ordinal types are: `int`, `char`, and `boolean`.

Remark 91.1. There are 2 User-Defined Ordinal Types that are supported by most programming languages:

- Enumeration Types
- Subrange Types

6.3.1 Enumeration Types

Defn 92 (Enumeration Type)**.** An *enumeration type* is one in which all of the possible values, which are named constants, are provided (enumerated) in the definition. Enumeration types provide a way of defining and grouping collections of named constants, called *enumeration constants*.

This is an example of an enumeration type in C#:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

Remark 92.1. Typically, each of the numeration constants is implicitly assigned an integer literal, though they can be given integer literals explicitly too.

The design issues for Enumeration Types are:

- Is an enumeration constant allowed to appear in more than one Enumeration Type definition, and if so, how is the type of an occurance of that enumeration constant in the program checked?
- Are enumeration constants coerced to integers?
- Are any other types coerced to an Enumeration Type?

6.3.1.1 Designs In languages without native support of Enumeration Types, they are simulated with integer values. For example,

```
int red = 0, blue = 1;
```

However, this can lead to unexpected behavior. For example, the variables `red` and `blue` can be added together. In essence, there would be no Type Checking. The value for those variables could be overwritten somewhere. Though, that issue would be solved by making the variable a constant instead.

C and Pascal introduced the use of Enumeration Types. These implicitly use default values, integers, as the enumeration constants. However, the values can be set explicitly, by the programmer. With these Enumeration Types, we have and avoid these issues:

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
myColor++; // Valid code, sets myColor from blue to green
myColor = 4; // Illegal
myColor = (colors) 4; // Legal because 4 is being typecast
```

These help prevent some issues, but not all.

The next iteration was in Ada. They allowed for *overloaded literals* in their Enumeration Types. This means there were enumeration constants shared between 2 Enumeration Types in the same referencing environment. In this case, the value must be determinable from the context of the Enumeration Type. Sometimes, a mroe explicit specification must be used. Additionally, because the enumerations constants were **not** coerced to integers, nor were the **enumeration variables**, the range of operations and range of values for the enumeration constants was limited. This allowed the compiler to pick up many more errors.

Remark. None of the relatively recent scripting kinds of languages include Enumeration Types. These include Perl, JavaScript, PHP, Python, Ruby, and Lua.

6.3.1.2 Evaluation Enhancements to both Readability and Reliability.

- Readability is enhanced by better named values
- Reliability is enhanced by being able to perform Type Checking on the Enumeration Types.
  - No arithmetic operations allowed on Enumeration Types.
  - No enumeration variable can be assigned a value outside the Enumeration Type's assigned range.

6.3.2 Subrange Types

Defn 93 (Subrange Type)**.** A *subrange type* is a contiguous sequence of an Ordinal Type. For example, this is a subrange: `12..14`.

6.3.2.1 Ada's Design

Ada included Subrange Types in Subtypes.

Defn 94 (Subtype)**.** A *subtype* in Ada is an extension, usually constrained, version of existing types. For example,

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
Day1 : Days;
Day2 : Weekdays;
...
Day2 := Day1; -- Will only work if Day1 has Mon-Fri, fails if Day1 = Sat or Sun
```

The compiler generates range-checking code for every assignment to the subrange type. Subranges require run-time range checking.

6.3.2.2 Evaluation

Subrange Types improve Readability by making it clear that Variables of Subtypes can only store a certain range of values. Reliability is increased with Subrange Types because assigning a value to a subrange variable outside its range is detected as an error.

6.3.3 Implementation of User-Defined Ordinal Types

Enumeration Types are usually implemented on integers. However, without restrictions on ranges of values and possible operations, this does **not** improve Reliability.

Subrange Types are implemented the same way as their parent types, except range checks are implicity included by the compiler in every assignment of a variable or expression to a subrange variable.

6.4 List Types

Lists were first supported in LISP.

Defn 95 (List)**.** A *list* is a data structure heavily used in Functional Programming Languages. They are similar to arrays in other languages, but they may lazily evaluated and may be infinite.

Remark 95.1. Lists, because of their (potentially, depends on the language) inherently infinite nature, have always been part of Functional Programming Languages, but are making their way to Imperative Programming Languages too.

Remark. Karl Hallsby went a little ham on this section because he uses Emacs and writes ELisp to customize it. He is also *really* interested in Functional Programming Languages.

Lists in Scheme, LISP, and Common LISP are written as such:

```
(A B C D) ; List of 4 elements
(A (B C) D) ; List of 3 elements, with the middle being a 2 element nested list
```

In LISP and its descendants, data and code have the same syntactic form, meaning this could be interpreted as a function call to `A` with `B` and `C` being parameters; or as a list of 3 elements.

```
(A B C)
```

Lists in Scheme, LISP, and Common LISP can be considered linked-lists. This means there are operations to get the current node's data and to get the next nodes in the rest of the list.

```
; The ' in front of a list means to interpret the list as data and not a function call
(CAR '(A B C)) ; Returns the element A
(CDR '(A B C)) ; Returns the list (B C)
```

There are 2 ways these lists can be constructed:

1. `CONS` takes 2 parameters and returns a list with the first parameter as the first element and the second parameter as the remainder of the list.

```
(CONS 'A '(B C)) ; Returns (A B C)
```

2. `LIST` takes any number of parameters and returns a new list with the parameters as the new list's elements

```
(LIST 'A 'B '(C D)) ; Returns (A B (C D))
```

Defn 96 (List Comprehension)**.** A *list comprehension* is an idea from set notation and set theory. Essentially, a list comprehension applies a function to every element in a given array/List, and a new array/List is constructed from the results.

2 examples of this are shown below, the first in Haskell, the second in Python 3.

```
[n * n | n <- [1..10]]
```

```
[x * x for x in range(1, 11, 1)]
```

6.5 Arrays

Defn 97 (Array)**.** An *array* is **TODO**.

6.6 Associative Arrays

Defn 98 (Associative Array)**.** An *associative array* is an unordered collection of data elements that are indexed by an equal number of values, called *keys*. The user-defined keys must be stored in the structure, along with the values to be stored.

To store the keys, they must be *hashed*. This is done with a hash function.

Remark 98.1 (Alternative Names). There are many alternative names for Associative Arrays.

- Hashtable
- Hashmap
- Associative Array
- Dictionary

Remark 98.2 (Improving Capabilities). What the associative array/dictionary/hashtable are allowed to store depends on the language. In Python and Ruby, objects and Primitive Data Types can be stored. In PHP, integers or strings can be keys. In Ruby, any object can be a key.

6.6.1 Structure and Operations

The code examples here are done in Perl, but it is similar in most other languages. For more specific code, you will have to visit the language's documentation.

```
# Defining an associative array
%salaries = ("Gary" => 75000, "Perry" => 57000, "Mary" => 55750);

# Switch from % for hash to £ for single values in the hash
# That should be a dollar sign, not a Stirling Pound symbol

# Changing a value in the associative array
$salaries{"Perry"} = 58850;

# Removing a key-value pair from the associative array
delete $salaries{"Gary"};
```

The size of a Perl Associative Array is dynamic and will grow and shrink as needed.

Associative Arrays are very useful if there is a lot of searching that needs to be done, because the hashing of the key, then the search term allows for $O(1)$ lookup speeds for an element located anywhere in the Associative Array.

6.6.2 Implementing Associative Arrays

The implementation for an Associative Array differs between languages. However, in all languages, eventually the Associative Array can get "full". This is when the collision chance of 2 hashes modulo the length of the structure gets too high. This means that if an element is already stored, and something new were hashed and modulo-d the length of the structure, there is a very high probability that the new element would "collide" with the old one.

To handle this, Associative Arrays are grown. Every key-value pair will need to have their key recalculated. This is a slightly time-costly operation, $O(n)$, but the spatial-cost is much higher, because there are 2 potentially very large arrays in memory at the same time.

6.7 Record Types

Defn 99 (Record)**.** A *record* is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure, similar to arrays. The offset from the head of the record to reach any Field is known statically, because the sizes and Types are known at compile time.

Records are used to model a collection of data in which the individual elements, the Fields are not of the same Type or size.

Remark 99.1 (Clarification). An important clarification here is that the Record, defined in Definition 99 is **NOT** a record in a database **in any way**.

Remark 99.2. A Record is similar to a heterogeneous array, but they differ in one key way. A heterogeneous array is an array of pointers to areas of Memory that may be discontinuous. However, all the Fields in a Record all reside in adjacent Memory locations.

Remark 99.3 (Record vs. Object). A Record and an object are quite similar. However, the differences between them depend on the language.

Defn 100 (Field)**.** A *field* is an element a Record. These are fixed length, with fixed Type, meaning the Memory address can be statically calculated to reach any Field in the Record.

Fields are referenced by their identifier, rather than an index.

6.7.1 Definitions of Records

There are 2 design questions that need to be asked when defining Records.

1. What is the syntactic form of references to Fields?
2. Are elliptical references allowed?

Below are 2 blocks of code, the first from COBOL, the second from Ada. Both describe an employee.

```
01  EMPLOYEE-RECORD.
    02 EMPLOYEE-NAME.
       05 FIRST      PICTURE IS X(20).
       05 MIDDLE     PICTURE IS X(20).
       05 LAST       PICTURE IS X(20).
    02 HOURLY-RATE   PICTURE IS 99V99.
```

The numbers `01`, `02`, and `05` are *level numbers*, which indicate relative hierarchical values. Any line that is followed by a line with a higher-level number is itself a record. `PICTURE` clauses show the formats of the storage locations. `X(20)` is a 20 character alphanumeric string and `99V99` is a 4 decimal digit number with the dcimal in the middle.

However, Ada does not have the level numbers like COBOL, so they allow for nesting Record structures inside Record declarations.

```
type Employee_Name_Type is record
   First : String (1..20);
   Middle : String (1..20);
   Last : String (1..20);
end record;
type Employee_Record_Type is record   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

In Java and C#, Records can be defined as data classes, whith nested Records defined as nested classes. Lua's tables serve this purpose.

6.7.2 References to Record Fields

There are many ways to refer to individual Fields. We will look at the way COBOL referenced Fields, and the Dot Notation.

Defn 101 (Dot Notation)**.** Most programming languages use *dot notation* for Field references, where components necessary to reach the Field are connected with periods. The outermost Record goes on the left, and gets more specific as it grows to the right.

There are 2 examples below of accessing the middle name of the Employee Records we made in the previous section.

```
MIDDLE OF EMPLOYEE-NAME OF EMLOYEE-RECORD
```

```
Employee_Record.Employee_Name.Middle;
```

There are 2 ways to make a reference to a Record Field.

1. Fully Qualified References
2. Elliptical References

Defn 102 (Fully Qualified Reference)**.** A *fully qualified reference* to a record field is one in which to access a Field, the programmer **MUST** specify all intermediate Records to go through.

An alternative to fully qualified reference is the Elliptical Reference.

Defn 103 (Elliptical Reference)**.** An *elliptical reference* allows a programmer to specify the Field and omit any to all parent Records; so long as the resulting reference in unambiguous in the referencing environment (the Scope).

These are a programmer convenience, but are **incredibly** difficult to compile, because of the alaborate data structures and procedures required to correctly identify the referenced field. There is also a slight loss of Readability.

6.7.3 Evaluation of Record Types

Records are valuable to programming and programming languages. Their design is straightforward and use is safe.

Records and Arrays are quite similar, but differ in some key ways

- Arrays:
  - All data values have the same Type. This allows for easy subscripting of Memory addresses
- Record
  - When the collection of data values is heterogeneous
  - The different data Fields are not processes the same way
  - The Fields are not processed in any particular order
  - Fields are like named subscripts
  - Since Field names are static, they provide efficient access to fields.

6.7.4 Implementation of Record Types

The Fields of Records are stored in adjacent Memory locations. But because each Field may be of a different size, the offset address, relative to the beginning of the Record is associated with each field. These are calculated at compile-time. This way, there are no runtime calculations that need to be done.

6.8 Tuple Types

Defn 104 (Tuple)**.** A *tuple* is a Type that is similar to a Record, except the elements are **not** named.

Remark 104.1. Tuples can be used to return multiple values from a function in languages that do not natively support that.

Remark 104.2. Python natively supports these. These behave similarly to Lists, but are immutable.

```
myTuple = (3, 5.8, 'apple')
```

6.9 Union Types

Defn 105 (Union)**.** A *union* is a Data Type whose Variables may store different Type Values at different times during program execution.

The union will allocate the maximum space required by all the Types of the union, and only use the parts it needs, based on what Types are actually in use.

6.9.1 Design Issues

- Should Type Checking be required?
  - This Type Checking will have to be dynamic, running during program execution.
- Should Unions be embedded in Record?

6.9.2 Discriminated vs. Free Unions

The `union` construct in C/C++ is used to specify Union structures. These are called Free Unions. These stand in stark contrast to Discriminated Unions.

Defn 106 (Free Union)**.** A *free union* is a Union that have ***NO*** Type Checking enforced on their use. For example, this C snippet:

```c
union flexType {
    int intEl;
    float floatEl;.
};
union flexType el1;
float x;
...
el1.intEl = 27;
x = el1.floatEl; // NOT type checked, because current type of el1 cannot be determined
```

The last assignment is not type checked, because the the current Type of `el1` cannot be checked. So, `27` is assigned to the `float` variable `x`, which is nonsense.

Defn 107 (Discriminated Union)**.** A *discriminated union* makes use of a *tag* or *discriminant* as a Data Type indicator. These allow for the Type Checking of Unions during runtime.

6.9.3 Ada Union Types

Ada allows the use to specify variables for a variant Record that will only store one of the possible Type values in the variant.

Defn 108 (Constrained Variant Variable)**.** A *constrained variant variable* is when a language allows the programmer to specify the types present in a variant Data Type, allowing for static Type Checking. These enforce that variants can only be changed by assigning the entire record at a time.

This is shown in the code snippet below.

```ada
type Shape is (Circle, Triangle, Rectangle); -- Construct enumeration for types of shapes possible
type Colors is (Red, Green, Blue); -- Construct enumeration for colors
type Figure (Form : Shape) is record -- Figure has variant Form records of type Shape (from enumeration)
    Filled : Boolean;
    Color : Colors;
    case Form is
        when Circle =>
            Diameter : Float;
        when Triangle =>
            Left_Side : Integer;
            Right_Side : Integer;
            Angle : Float;
        when Rectangle =>
            Side_1 : Integer;
            Side_2 : Integer;
    end case
end record;
...
Figure_1 : Figure; -- Unconstrained variant record of the record type Figure, and has no initial values
Figure_2 : Figure(Form => Triangle); -- Constrain the variant record to a Triangle
-- Figure_1's type can be changed by the assignment of a whole record
Figure_1 := (Filled => True,
```

```
            Color => Blue,
            Form => Rectangle,
            Side_1 => 12,
            Side_2 => 3);
```

6.9.4 Evaluation

Unions are potentially unsafe constructs in some languages, because they cannot be type checked. C and C++ are not strongly typed for this reason. However, Ada, ML, Haskell, and F# are strongly typed, because they can perform Type Checking on Unions. Some languages, like Java and C# do not even include the ability to construct a Union.

6.9.5 Implementation of Union Types

Unions are implemented by using the same address for a single Union, no matter its variant. Enough storage is allocated for the largest possible variant. Then, depending on the variant, the space will be used according to how the Union was defined. The tag of the Union variant is stored in its Descriptor.

6.10 Type Equivalence

This section does not deal with Type Compatibility, which works for scalar Data Types, but rather Type Equivalence.

Defn 109 (Type Compatibility)**.** *Type compatibility* dictates the Data Type of operands that are acceptable for eacch of the operations of the language. This is called compatibility because there are cases when the Data Type of the operand can be implicitly converted by the compiler or run-time system to make the operand acceptable to the operator.

An example of this is the addition of an integer number and a real number. The integer number is typecast to a real number, then the addition is performed.

Type Compatibility rules are strict for predefined scalar types. However, structured Data Types such as Arrays, Record, and others require more complex rules. Since Data Type coercion is unlikely, the question is if the 2 Data Types are equivalent.

Defn 110 (Type Equivalence)**.** *Type equivalence* is a strict form of Type Compatibility. It is when an operand of one Data Type can be substituted for another operand of the same type, without Data Type coercion.

The design of type equivalence rules influence the design of Data Types and operations provided for values of those types.

There are 2 approaches to determining Type Equivalence:

1. Name Type Equivalence
2. Structure Type Equivalence

There is a general algorithm for determining if two Data Types are equivalent. "... when all constant expressions are replaced by their values and all type names are replaced by their definitions. In the case of recursive types, the expansion is the infinite limit of the partial expansions ...". Meaning you replace everything you can, and if there is an infinite recursion somewhere, you cut it off eventually, usually when you pass the same point again.

There are 4 types of equivalence that can be established.

1. Primitive Equivalence
2. Structure Type Equivalence
3. Reference Equivalence
4. User-Defined Equivalence

Different languages use different approaches and combinations of these types of Type Equivalence. You would have to look at the language's specification to find out exactly what is used where. Additionally, object-oriented languages present their own, unique, type of Type Compatibility with object compatibility and its relationship to the inheritance hierarchy.

Defn 111 (Name Type Equivalence)**.** *Name type equivalence* means that 2 Variables have Type Equivalence if they are defined in:

- The same declaration
- In Declarations that use the same Data Type name

This is easier to implement, but more restrictive. In a strict interpretation, a Variable whose Type is a subrange of the integers would **not** be equivalent to an integer Type Variable. For example,

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

`count` and `index` could **not** be substituted for each other.

Remark 111.1. To use Name Type Equivalence, all Types must have names. If the language supports anonymous Data Types, then they must be given internal names by the compiler/interpreter.

Defn 112 (Primitive Equivalence)**.** *Primitive equivalence* directly compares two values by comparing their bit patterns in memory.

Defn 113 (Structure Type Equivalence)**.** *Structure type equivalence* means 2 Variables have Type Equivalence if their types have identical structures. The entire structure's Types must be compared to determine equivalence. This may potentially invole recursing through the structure or iterating over the structure. However, this is more flexible than Name Type Equivalence, but more difficult to implement.

Ada, with its hyper-strict Type Equivalence has defined 2 ways to make new types.

1. Derived Type
2. Subtype

Defn 114 (Reference Equivalence)**.** *Reference Equivalence* checks whether two pointers/references point to the same address in Memory.

Defn 115 (User-Defined Equivalence)**.** *User-defined equivalence* performs arbitrary equality checking but isn't provided by the language itself, and must be specified by the programmer.

Defn 116 (Derived Type)**.** A *derived type* is a new Data Type that is based on some previously defined Data Type, that **is not equivalent, but may have an identical structure**. Derived types inherit all the properties of their parent types.

Take the following Ada code snippet as an example.

```
1 type Celsius is new Float;
2 type Fahrenheit is new Float;
```

These are not equivalent, though they have identical structures. They are also not type equivalent to any other `Float` type.

Defn 117 (Subtype)**.** A *subtype* is a possibly range-constrained version of an existing type. A subtype **is equivalent with its parent type**.

Take the following Ada code snippet as an example.

```
1 subtype Small_type is Integer range 0..99;
```

The `Small_type` is equivalent to the `Integer` type.

7 Expressions

Defn 118 (Expression)**.** An *expression* is a combination of one or more Operands and operators that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce another value.

Remark 118.1 (Overloading). An expression can be overloaded if there is more than one definition for an operator. These operators can be statically overloaded by defining multiple semantic rules with their operation being defined for each of the types τ that can be used in the rule.

Defn 119 (Operand)**.** An *operand* is a:

- Constant
- Variable
- Another Expression
- Result from function calls

Defn 120 (Well-Typed)**.** An Expression is *well-typed* if and only if we can show $e : \tau$ for some type τ.

7.1 Arithmetic Expressions

One of the main goals of high-level languages was to have automatic evaluation of Expressions similar to those in math, science, and engineering. Most of these characteristics came from mathematics directly.

7.1.1 Arity

Defn 121 (Arity)**.** *Arity* is the number of Operands that must be present to evaluate that Expression. In most programming languages, there are 3:

1. Unary
2. Binary
3. Ternary

Defn 122 (Unary)**.** A *unary* operation requires a single Operand. For example, negation, `-x`

Defn 123 (Binary)**.** A *binary* operation requires 2 Operands. For example, addition, `y + z`

Remark 123.1. These are usually use Infix notation.

Defn 124 (Ternary)**.** A *ternary* operation requires 3 Operands. For example, `w = if x ? y : z`.

Remark 124.1. As far as I know, the only ternary operator is a single-line if statement.

7.1.2 Fixity

Defn 125 (Fixity)**.** *Fixity* is the position of an operation relative to an Operand. There are 3 possible positions:

1. Prefix
2. Infix
3. Suffix

Defn 126 (Prefix)**.** *Prefix* notation has the operators before the Operands that it operates on. For example:

- `-x`
- `+ x y`

Defn 127 (Infix)**.** *Infix* notation has the operators between the Operands that it operates on. This necessitates that the expression uses at least 2 Operands, making this a potential Binary operator. For example:

- `x+y`

Defn 128 (Suffix)**.** *Postfix* or *suffix* notation has the operator after the Operands that it operates on. For example:

- `x!`
- `x y +`

7.1.3 Operator Evaluation Order

7.1.3.1 Precedence

Defn 129 (Operator Precedence Rules)**.** *Operator precedence rules* partially define the order in which operators of different precedence levels are evaluated. This is based on the hierarchy or operator priorities, as defined by the language designer.

Remark 129.1. Operator Associativity Rules also define the order in which operators are evaluated in an Expression.

There may be unary addition, called the *identity operator*, because it usually no associated operation. Unary "subtraction" changes the sign of the operand, negating it.

	Ruby	C-Based Languages
Highest	`**`	postfix `++`, `--`
	Unary `+`, `-`	Prefix `++`, `--`, Unary `+`, `-`
	`*`, `/`, `%`	`*`, `/`, `%`
Lowest	Binary `+`, `-`	Binary `+`, `-`

Table 7.1: Precedence of Arithmetic Operators

7.1.3.2 Associativity

Defn 130 (Operator Associativity Rules)**.** *Operator associativity rules* partially define the order in which operators of **the same** precedence levels are evaluated. There can be:

- Left associativity: When evaluating an expression, parentheses that determine the order of evaluation are accumulated to the left, meaning the left-hand side is evaluated first.
- Right associativity: When evaluating an expression, parentheses that determine the order of evaluation are accumulated to the right, meaning the right-hand side is evaluated first.
- Nonassociativity: When evaluating an expression, parentheses that determine the order of evaluation are unknown and must be **explicitly** specified.

Remark 130.1. There are very few operators that are right-associative. As far as I know, only the exponentiation operator that can be right associative.

Language	Associativity Rule
Ruby	Left: `*`, `/`, `+`, `-`
	Right: `**`
C-Based languages	Left: `*`, `/`, `%`, Binary `+`, Binary `-`
	Right: `++`, `--`, Unary `+`, Unary `-`
Ada	Left: All except `**`
	Right: None
	Nonassociative: `**`

Table 7.2: Associativity of Arithmetic Operators

Remark. **This section only applies to integer arithmetic operators.** Floating-point arithmetic is different because of the way numbers are represented and their finite precision.

7.1.3.3 Parentheses Parentheses can alter the Operator Precedence Rules and Operator Associativity Rules. A parenthesized part of an expression has precedence over its adjacent unparenthesized parts.

For example, the addition will be done first here

$$(A + B) * C$$

7.1.3.4 Ruby Expressions Everything (literally everything, including literals) is an object in Ruby. Ruby supports the arithmetic and logic operations that are included in C-based languages, but they are slightly different. For example, the expression `a + b` is a call to the `+` method of the `a` object, and passes a reference to the `b` object as a parameter to the `+` method. This means that the operator can be overloaded by the programmer, which is useful for user-defined Data Types.

7.1.3.5 LISP Expressions LISP is similar to Ruby in that arithmetic and logic operations are computed by subprograms. However, in LISP and its descendants, the subprograms (operators) **must** be called explicitly. For example, to write $A+B*C$ in LISP[1].

```
(+ A (* B C)) ; Computes A + B * C
```

7.1.3.6 Conditional Expressions Conditional expressions can be used as an expression with the Ternary variant. For example,

```
if (count == 0) {
  average = 0;
} else {
  average = sum / count;
}
```

is equivalent to

```
average = (count == 0) ? 0 : sum / count;
```

[1]When a LISP list is interpreted as code, the first element is the function name, and the rest are passed parameters.

7.1.4 Operand Evaluation Order

How do we determine and what steps must be taken to get the value of the Operand used in the current Expression?

7.1.4.1 Side Effects

Defn 131 (Side Effect)**.** A *side effect* of a function is when the function changes either one of its parameters or a Global Variable.

Defn 132 (Pure)**.** A function can be called *pure* when the function **does not** change its parameters or a Global Variable.

Remark 132.1. These are used in both mathematics and Functional Programming Languages.

If a function has Side Effects, then the order in which the Operands are evaluated may affect the operation.
There are 2 possible solutions to this:

1. Language designer can disallow functional Side Effects, essentially making all functions used in an Expression Pure.
   - However, implementing this is difficult in Imperative Programming Languages
   - Eliminates some flexibility for the programmer
   - Access to Global Variables would have to be disallowed, which the compiler may want to do to improve speed.
2. Language definition could state that Operands in Expressions are to be evaluated in a particular order and demand implementors guarantee that order.
   - Some code optimization techniques that reorder the Operand evaluations could no longer be used.

7.1.4.2 Referential Transparency and Side Effects

Defn 133 (Referential Transparency)**.** A program has *referential transparency* if any 2 Expressions in the program that have the same value can be substituted for another, without affecting the action of the program.

Remark 133.1. A function/program that has Referential Transparency should also be a Pure program/function.

Remark 133.2. In a Functional Programming Language ***ALL*** functions/programs have Referential Transparency, by definition.

There are several advantages:

- Semantics of referentially transparent programs are easier to understand.
- A programmed function is equivalent to a mathematical function in that it is Pure.

7.2 Relational and Boolean Expressions

7.2.1 Relational Expressions

Defn 134 (Relational Operator)**.** A *relational operator* is an operator that compares the values of its two Operands. These are ***ALWAYS*** Binary operators.

Defn 135 (Relational Expression)**.** A *relational expression* has 2 Operands and a Relational Operator. The value of a relational expression is Boolean, except when a language does not have a Boolean Data Type. **These expressions have a lower precedence than arithmetic expressions**.
This operation determines truth or falsehood.

Relational Expressions can be very simple (integers) or very complex (strings). The usual list of Relational Operators is shown below

1. `==`, Equal to
2. `!=`, Not Equal to
   - C-based languages use `!=`
   - Ada uses `/=`
   - Lua uses `~=`
   - Fortran 95+ uses `.NE.`[2] or `<>`
   - ML and F# use `<>`
3. `>=`, Greater than or Equal to
4. `<=`, Less than or Equal to
5. `>`, Greater than

[2]This is because the punchcards used when Fortran was first developed did not have the > or < symbols.

6. `<`, Less than

There is also a special case for equality relational operators

- `==`, Equals to, after Data Type coercion
- `===`, Equals to, without Data Type coercion
- `eql?`, Equals to, without Data Type coercion, for Ruby

7.2.2 Boolean Expressions

Defn 136 (Boolean Expression)**.** A *boolean epxression* consists of Boolean variables, constants, Relational Expressions, and Boolean operators. The most common Boolean operators are:

- `AND`
- `OR`
- `NOT`, Logical complement
- `XOR`, exclusive OR

Boolean operators **only** take Boolean operands. Now that we have added 2 more types of expressions onto our precedence list, we need to fill in the rest.

Highest	Postfix `++`, `--`
	Unary `+`, `-`, Prefix `++`, `--`
	`*`, `/`, `%`
	Binary `+`, `-`
	`<`, `>`, `<=`, `>=`
	`==`, `===`, `!=`
	`&&`
Lowest	`||`

Table 7.3: Precedence Table with All Expression Types

Programming languages should implement a Boolean Data Type for truth-based comparisons. However, versions of C before C99 did not have a Boolean type, forcing programmers to use 0 to represent `false` and anything else being `true`.

7.3 Short-Circuit Evaluation

Defn 137 (Short-Circuit Evaluation)**.** A *short-circuit evaluation* of an Expression happens when a result is determined without evaluating all the Operands in the Expression.

This is **mostly** used for Boolean Expressions. The 2 possible Short-Circuit Evaluations are for `AND` and `OR`. They are based off the operator's truth tables, shown in Tables 7.4 to 7.5.

	0	1
0	0	0
1	0	1

Table 7.4: `AND` Truth Table

	0	1
0	0	1
1	1	1

Table 7.5: `OR` Truth Table

The 2 possible Short-Circuit Evaluations are:

1. If the first Operand in an `AND` Boolean Expression is `false`, then it short-circuit evaluates to false.
2. If the first Operand in an `OR` Boolean Expression is `true`, then it short-circuit evaluates to true.

Short-Circuit Evaluation has some side effects:

- The second Operand is **never** evaluated, and if its a function, there might be Side Effects, or a lack of them.

Some languages specify special version of the Boolean operators to explicitly handle Short-Circuit Evaluation separately. Most simply choose to only support Short-Circuit Evaluation, and make sure the programmer keeps it in mind while working.

8 Natural Semantics

Natural semantics assumes that we know the syntax of the language. We will assume that we have a Backus-Naur Form Context-Free Grammar. We will also assume that ambiguities have been resolved, somehow.

Defn 138 (Semantics)**.** *Semantics* is the act of attaching a meaning to a syntactic construct. For instance, we know the value of 1 to be 1, but does one = 1? We have defined the semantics of one to be equivalent to the numeral 1.

We can defined these semantic relationships with Evaluation Relations.

Defn 139 (Evaluation Relation)**.** The *evaluation relation* states the relation between a program p and the program's result v:

$$p \Downarrow v \tag{8.1}$$

This is read as "p evaluates to v".

Remark 139.1. The value of the left-hand side is an arbitrary set of operator(s). For instance, if we wanted to specify addition, but using the `add` operator, then p would look $\texttt{e}_1\ \texttt{add}\ \texttt{e}_2$. However, on the right-hand side, we *MUST* specify the operation that will take place. The right-hand side is a mathematical construction that the reader will understand.

Defn 140 (Metavariable)**.** A *metavariable* is a variable in the Metalanguage. In Equation (8.1), p and v are metavariables.

p is a metavariable that can contain any input program, and v is a metavariable that can contain any result that the language might compute.

Throughout this section, we will use this Backus-Naur Form Context-Free Grammar.

$$\begin{aligned} \langle \text{expr} \rangle &\to nat \\ &\mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\mid \langle \text{expr} \rangle \texttt{div} \langle \text{expr} \rangle \end{aligned} \tag{8.2}$$

Let's start with the programming language specified by the Context-Free Grammar below.

$$\langle \text{W} \rangle \to \texttt{ett} \mid \texttt{två} \mid \texttt{tre}$$

This allows only 3 programs, each of which is a single word in Swedish. We can define the semantics (attach meaning) with the following rules:

$$\texttt{ett} \Downarrow 1 \qquad \texttt{två} \Downarrow 2 \qquad \texttt{tre} \Downarrow 3$$

8.1 Ambiguous Semantics

Just like in Context-Free Grammars, there can be ambiguities in Semantics too. They occur when there is more than 1 specification for a symbol. For example,

$$\langle \text{Q} \rangle \to \texttt{eins} \mid \texttt{zwei} \mid ?$$

with the Semantics below:

$$\texttt{eins} \Downarrow 1 \qquad \texttt{zwei} \Downarrow 2 \qquad ? \Downarrow 0 \qquad ? \Downarrow 3$$

In this course, we are interested in unambiguous Semantics. When language designers design a programming language, they try to avoid these ambiguities.

We can normally check if a set of semantic rules are unambiguous by checking that the left-hand side of each of the Evaluation Relations does not overlap with another Evaluation Relation. If there is an overlap, then it **must** be shown that the same result will be yielded.

8.2 Conditional Rules

When we want a Metavariable to only get a value under certain conditions, we have a special notation for that. For example, say we want n to be a natural number that is used in the grammar of Equation (8.2). We can write this condition as:

$$\frac{n \in \mathbb{N}}{n \Downarrow \texttt{asNat}(n)} \tag{8.3}$$

The writers of these semantic grammars favor simplicity, so they might omit the `asNat` function by arguing "separating between natural numbers and their textual representation is overly pedantic, as long as there is no ambiguity". They might write the rule in Equation (8.3) as:

$$\frac{n \in \mathbb{N}}{n \Downarrow n} \tag{8.4}$$

8.3 Recursion

If we want to construct the addition in Equation (8.2), then we can write:

$$\frac{e_1 \Downarrow n_1 \;\; e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2} \tag{8.5}$$

8.4 Completeness

A semantic specification is *complete* when all possible cases have been defined, in some way. For example:

$$\frac{e_1 \Downarrow n_1 \;\; e_2 \Downarrow n_2}{e_1 \;\texttt{div}\; e_2 \Downarrow \left\lfloor \frac{n_1}{n_2} \right\rfloor} \tag{8.6}$$

This rule ignores the case when $n_2 = 0$, which means we divide by 0. In mathematics, this is an undefined operation.

There are 2 ways to solve this problem:

1. **Adding a Meta-Rule:** Add an informal rule that states if the Semantics of an operation is not defined, then execution aborts with an error. This is a solution, but it allows the language designer to solve corner cases with strange behavior. This also prevents any forms of error recovery, harming Reliability.
2. **Error Values:** Extend the $\Downarrow$ Evaluation Relation operator so that we can return *error values* along with intended results. This may require adding **MANY** new rules to the language.

Example 8.1: Natural Semantics. Lecture 3

For the given Context-Free Grammar, construct rules for the Natural Semantics of this language? Then, perform a proof/derivation using those rules for the statement $\max(1+2)(1+1)$? Ignore the parentheses; those are only used to show that the 2 expressions are separate. Assume there is a process before applying the Context-Free Grammar which handles the parentheses.

$$\begin{aligned} \langle \text{expr} \rangle \longrightarrow\; & \underline{\texttt{num}} \\ & \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ & \mid \max \langle \text{expr} \rangle \langle \text{expr} \rangle \end{aligned}$$

NOTE: Green will represent input to the rule and blue will represent rule output. When deriving statements with these rules, there may be additional colors to represent various values.

The easiest rule to define is the rule for $\underline{\texttt{num}}$.

$$\frac{n \in \underline{\texttt{num}}}{n \Downarrow n}(num)$$

Next, we define the rule for the addition.

$$\frac{e_1 \in \langle \text{expr} \rangle \; e_2 \in \langle \text{expr} \rangle \; e_1 \Downarrow n_1 \; e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2}(add)$$

Remark. In this case, the expression's rule actually returns the computed number from the expression $n_1 + n_2$, rather than the 2 expressions added together (which would be had if we had written $e_1 + e_2$ instead).

Remark. From here on out, if there is $e_x \Downarrow n_x$, then we also implicitly say $e_x \in \langle \text{expr} \rangle$.

Now we have to define the `max` operation. In this case, we need 2 rules to specify the 2 cases, when the left expression is the relative maximum, and when the right expression is the relative maximum.

$$\frac{e_1 \Downarrow n_1 \; e_2 \Downarrow n_2 \; n_1 \geq n_2}{\max e_1 \; e_2 \Downarrow n_1}(\text{max left})$$

$$\frac{e_1 \Downarrow n_1 \; e_2 \Downarrow n_2 \; n_1 \leq n_2}{\max e_1 \; e_2 \Downarrow n_2}(\text{max right})$$

Remark. Although the 2 equations for determining the relative maxima both have an equality aspect, in pure mathematics, that's fine. In mathematical terms, if both rules apply, because $e_1 == e_2$, then, both are technically maxima and should be returned.
However, this will change during implementation, because one of the rules will be checked first, which necessitates that the rules be made mutually exclusive.

Now that all the necessary rules have been developed, we can apply them to the statement $\max(1+2)(1+1)$.

$$\frac{\dfrac{\dfrac{1 \in \mathbb{N}}{1 \Downarrow n_1 = 1}(num) \quad \dfrac{2 \in \mathbb{N}}{2 \Downarrow n_2 = 2}(num)}{e_1 = 1+2 \Downarrow n_1 = n_1 + n_2 = 1+2 = 3}(add) \quad \dfrac{\dfrac{1 \in \mathbb{N}}{1 \Downarrow n_1 = 1}(num) \quad \dfrac{1 \in \mathbb{N}}{1 \Downarrow n_2 = 1}(num)}{e_2 = 1+1 \Downarrow n_2 = n_1 + n_2 = 1+1 = 2}(add) \; n_1 \geq n_2 = 3 \geq 2}{\max\underbrace{(1+2)}_{e_1}\underbrace{(1+1)}_{e_2} \Downarrow n_1 = 3}(\text{max left})$$

We chose the max left rule only **after** the values of e_1 and e_2 were calculated.

8.5 Language with Variables

For this section, we are going to use Equation (8.7) as our Context-Free Grammar.

$$\begin{aligned} \langle \text{expr} \rangle &\to nat \\ &\mid id \\ &\mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\mid \texttt{let}\ id = \langle \text{expr} \rangle \texttt{in} \langle \text{expr} \rangle \end{aligned} \tag{8.7}$$

8.5.1 Environments

Defn 141 (Environment)**.** An *environment* is a mathematical object that tells us the bindings of variables. We will use the following notation:

$$E_n = \{\mathtt{v}_1 \mapsto m_1, \mathtt{v}_2 \mapsto m_2\} \tag{8.8}$$

where v is a variable, and n is the number of variables present. So, for example:

$$E_2 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 7\}$$

Remark 141.1 (Empty Environment). The *empty environment* is written

$$E_\emptyset = \{\} \tag{8.9}$$

To retrieve an value from an Environment, where the Environment being used is Equation (8.8), we write:

$$E(\mathtt{a}) \Rightarrow 1 \tag{8.10}$$

If we want to update an Environment with a new binding,

$$E_2[\mathtt{c} \mapsto 42] = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 7, \mathtt{c} \mapsto 42\} \tag{8.11a}$$

$$E_2[\mathtt{a} \mapsto 0] = \{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 7\} \tag{8.11b}$$

We define this as:

$$E[x \mapsto v](y) = \begin{cases} v & \iff x = y \\ E(y) & \text{otherwise} \end{cases} \tag{8.12}$$

This means that if $x = y$, then x is remapped (updated) to the value v.

8.5.2 Defining Semantics with Environments

Environments, E, can now become parameters to our Evaluation Relations, $\Downarrow$. This is written

$$E \vdash p \Downarrow n \tag{8.13}$$

Which means p can be drawn from the environment E, and should evaluate to a number, n (n is defined elsewhere).

Example 8.2: Natural Semantics with Environments. Lecture 3

Given the Context-Free Grammar,

$$\begin{aligned}\langle \text{expr} \rangle \longrightarrow\ & \texttt{num} \\ & |\langle \text{expr} \rangle + \langle \text{expr} \rangle \\ & |\,\text{max}\langle \text{expr} \rangle \langle \text{expr} \rangle \\ & |\,\texttt{let id} = \langle \text{expr} \rangle \texttt{in} \langle \text{expr} \rangle \\ & |\,\texttt{id}\end{aligned}$$

Compute the result of `let x = 2 in x + 3`?

We will start by using the rules that we defined in Example 8.1. We will also need to add new semantic rules for `id` and the `let` statement.

$$\frac{i \in \texttt{id}}{E \vdash i \Downarrow E(i)}(id)$$

$$\frac{E \vdash e_1 \Downarrow n_1 \quad E[i \mapsto n_1] \vdash e_1 \Downarrow e_2}{E \vdash \texttt{let i} = e_1 \text{ in } e_2 \Downarrow n_2}(let)$$

We start by applying the most applicable rule, the (let) rule.

$$\frac{\dfrac{2 \in \mathbb{N}}{E_\emptyset \vdash 2 \Downarrow 2}(num) \quad \dfrac{\dfrac{\texttt{x} \in \texttt{id}}{E_\emptyset[\texttt{x} \mapsto 2] \vdash \texttt{x} \Downarrow E(\texttt{x}) \Rightarrow 2}(id) \quad \dfrac{3 \in \mathbb{N}}{E_\emptyset[\texttt{x} \mapsto 2] \vdash 3 \Downarrow 3}(num)}{E_\emptyset[\texttt{x} \mapsto 2] \vdash \texttt{x} + 3 \Downarrow 2 + 3 = 5}(add)}{E_\emptyset \vdash \texttt{let x} = 2 \texttt{ in x} + 3 \Downarrow 5}(let)$$

Once we've applied the (let) rule, we apply the (num) rule, because the expression being assigned is just a number. The result of that expression evaluation is piped into the next (add) derivation. From there, it is mapped to the input `x`, and the `x+3` is calculated. The first thing is to get the value of `x` from the mapping with the (id) rule. Once the $E(\texttt{x})$ returns the value of the variable `x`, we can use it. The other operand in the addition is found using the (num) rule. Then the addition occurs, and the derivation completes.

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

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.