EDAP05: Concepts of Programming Languages — Reference Sheet
Lund University

Karl Hallsby

Last Edited: July 14, 2020

Contents

List of Theorems

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.

Because of its close relationship to mathematics, it is much easier to develop mathematical techniques for reasoning about the behavior of programs developed in functional languages. These techniques are important tools for helping us to ensure that programs work properly without having to resort to tedious testing and debugging that can only show the presence of errors, never their absence. Moreover, they provide important tools for documenting the reasoning that went into the formulation of a program, making the code easier to understand and maintain.

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

	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

- 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
- 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`

```
1 def d(x):
2     r = x[::-1]
3     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

3.3.2 Exception Handling

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

3.3.3 Aliasing

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

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

3.3.4 Readability and Writability

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

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

3.4 Cost

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

1. Training programmers in a new language
   - Function of Simplicity and Orthogonality
   - Function of programmer experience
2. Writing software
   - Function of Writability of the language
3. Compilation time
   - Time to compile a program
   - Resources required to compile a program in a language
4. Run time
   - Performance during runtime
   - Dependent on the effort made to optimize the input source code
5. Financial cost of special software
   - The cost of using the Compiler for a language for instance.
   - Languages with free Compilers or interpreters tend to be accepted more quickly than languages with a financial cost

6. Cost of limited reliability
    - Maintenance time
        - Corrections made to errors in the program
        - Modifications to add new functionality
    - Insurance cost, in special cases
        - Airplanes
        - Nuclear power plants
        - X-Ray machines

4 Backus-Naur Form and Context-Free Grammars

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

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

4.1 Context-Free Grammars

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

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

Grammar	Rule Patterns	Type
Regular	$\langle \text{X} \rangle \to a\langle \text{Y} \rangle$ or $\langle \text{X} \rangle \to a$ or $\langle \text{X} \rangle \to \epsilon$	3
Context-Free	$\langle \text{X} \rangle \to \gamma$	2
Context-Sensitive	$\alpha\langle \text{X} \rangle\beta \to \alpha\gamma\beta$	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 21 (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 22 (Backus-Naur Form)**.** The *Backus-Naur form* of a Context-Free Grammar is an extension of the Canonical Form. This form is less formal than the Canonical Form, but is allows for condensation of multiple productions that have the same nonterminal on the left-hand side to the same production. This is done with the | symbol.

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

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

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

These include:

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

The Extended Backus-Naur Form is:

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

4.3 Use Today

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

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

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

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

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

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

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

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

The are denoted as shown below:

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

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

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

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

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

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

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

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

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

4.3.1 Multiple Productions on Single Line

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

For example,

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

can be combined to

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

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.

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

4.3.3 Grammars and Derivations

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

Example 4.1: Left-Most Derivation.

Perform a Leftmost Derivation of the sentence

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

With the grammar

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

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

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

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

Example 4.2: Right-Most Derivation.

Perform a Rightmost Derivation of the sentence

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

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

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

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

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 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 37 (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 38 (Evaluation Relation)**.** The *evaluation relation* states the relation between a program p and the program's result v:

$$p \Downarrow v \tag{5.1}$$

This is read as "p evaluates to v".

Remark 38.1. 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 $\texttt{e}_1$ `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 39 (Metavariable)**.** A *metavariable* is a variable in the Metalanguage. In Equation (5.1), 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.

$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \mathit{nat} \\ &\quad | \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\quad | \langle \text{expr} \rangle \texttt{ div } \langle \text{expr} \rangle \end{aligned} \tag{5.2}$$

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

$$\langle \text{W} \rangle \rightarrow \texttt{ett} | \texttt{två} | \texttt{tre}$$

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:

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

5.1 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,

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

with the Semantics below:

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

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.

5.2 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 (5.2). We can write this condition as:

$$\frac{n \in \mathbb{N}}{n \Downarrow \texttt{asNat}(n)} \tag{5.3}$$

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 (5.3) as:

$$\frac{n \in \mathbb{N}}{n \Downarrow n} \tag{5.4}$$

5.3 Recursion

If we want to construct the addition in Equation (5.2), then we can write:

$$\frac{e_1 \Downarrow n_1 \;\; e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2} \tag{5.5}$$

5.4 Completeness

Defn 40 (Complete)**.** A semantic specification is *complete* when all possible cases have been defined, in some way.

For example:

$$\frac{e_1 \Downarrow n_1 \;\; e_2 \Downarrow n_2}{e_1 \;\texttt{div}\; e_2 \Downarrow \left\lfloor \frac{n_1}{n_2} \right\rfloor} \tag{5.6}$$

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 5.1: 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.

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

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

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 $\max(1+2)(1+1)$.

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

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

5.5 Language with Variables

For this section, we are going to use Equation (5.7) as our Context-Free Grammar.

$$\begin{aligned} \langle \text{expr} \rangle &\to nat \\ &\mid id \\ &\mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\mid \texttt{let}\ id = \langle \text{expr} \rangle \texttt{in} \langle \text{expr} \rangle \end{aligned} \tag{5.7}$$

5.5.1 Environments

Defn 41 (Environment)**.** An *environment* is a mathematical object that tells us the bindings of variables. We will use the following notation:

$$E_n = \{\mathtt{v}_1 \mapsto m_1, \mathtt{v}_2 \mapsto m_2\} \tag{5.8}$$

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 41.1 (Empty Environment). The *empty environment* is written

$$E_\emptyset = \{\} \tag{5.9}$$

To retrieve an value from an Environment, where the Environment being used is Equation (5.8), we write:

$$E(\mathtt{a}) \Rightarrow 1 \tag{5.10}$$

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

$$E_2[\mathtt{c} \mapsto 42] = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 7, \mathtt{c} \mapsto 42\} \tag{5.11a}$$

$$E_2[\mathtt{a} \mapsto 0] = \{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 7\} \tag{5.11b}$$

We define this as:

$$E[x \mapsto v](y) = \begin{cases} v & \iff x = y \\ E(y) & \text{otherwise} \end{cases} \tag{5.12}$$

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

5.5.2 Defining Semantics with Environments

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

$$E \vdash p \Downarrow n \tag{5.13}$$

Which means p can be drawn from the environment E, and should evaluate to a number, n (n is defined elsewhere). If p is **not** defined in the Environment, but is possible to evaluate, for example $p = 4$, then nothing happens with the Environment.

Essentially, using $E \vdash p$ means that p has access to the Metavariables present in the Environment, but p is not required to be drawn from E.

Example 5.2: Natural Semantics with Environments. Lecture 3

Given the Context-Free Grammar,

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

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

We will start by using the rules that we defined in Example 5.1. 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.

5.6 Typing with Natural Semantics

Defn 42 (Typing Assertion)**.** A *typing assertion* asserts the type of an expression, operator, and/or operand. They are denoted as

$$expr : type \tag{5.14}$$

Equation (5.14) states that an expression *expr* has the type *type*.

Remark 42.1 (Typing Assertions of Literals). It is axiomatic for literals to have their corresponding type. For example, $3 : int$, $3.14 : float$, etc.

Using the basic Typing Assertions, one can apply these same assertions to operators, like `+`, `-`, etc. This is called a Typing Rule.

Defn 43 (Typing Rule)**.** Expressions have *typing rules* applied to them, which defines the expression in terms of its constiuent expressions. These are extensions of Typing Assertions, as these are usually recursively derived from the constiuent expressions' types.

For example, if we define `+` to require its two operands be int types, then it makes sense for the output of the operation to also be an int.

6 Names

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

- Variables
- Subprograms
- Formal Parameters
- Other program constructs

6.1 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?

6.2 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.

6.3 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 44 (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 45 (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 45.1 (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`

6.4 Variables

Defn 46 (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

6.4.1 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.

6.4.2 Address

Defn 47 (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 47.1 (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.

6.4.3 Type

Defn 48 (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.

6.4.4 Value

Defn 49 (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 49.1 (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.

6.5 Binding

Defn 50 (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 51 (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.

6.5.1 Binding of Attributes to Variables

Defn 52 (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-bit integers.

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

Defn 53 (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.

6.5.2 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

6.5.2.1 Static Type Binding

Defn 54 (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 55 (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 56 (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 56.1 (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.

6.5.2.2 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 57 (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 57.1 (Benefits of Dynamic Binding). The primary benefit of having Dynamic Binding is the programming flexibility it provides.

Remark 57.2 (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.

6.5.3 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 58 (Allocation)**.** *Allocation* is the act of binding a Value to a Memory cell for a Variable.

Defn 59 (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 60 (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 Storage Bindings and Lifetime of scalar Variables into 4 categories, according to their Lifetimes.

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

6.5.3.1 Static Variables

Defn 61 (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.

Remark 61.1 (`static` as a Modifier). The Keyword `static` can be used on variables to tell the compiler/runtime system that the variable should be placed in the `static` portion of memory. This is similar to making a Global Variable, in that they are both stored in the static portion of memory. However, the use of the `static` Keyword is used when the variable should have a limited scope. For example, consider the code below.

```
1 int f(int x) { // Parameter is locally scoped and stack-dynamic
2         int y; // Locally scoped variable, stack-dynamic
3         int* a = // The pointer, *a, is a locally scoped variable and is stack-dynamic
4                 malloc(sizeof(int)); // The anonymous heap variable is locally scoped and heap-dynamic
5         static int c; // This is a locally-scoped variable allocated in static memory
6         return 0;
7 }
8 int x; // Global-scope variable, allocated in static memory
```

The pros and cons of Static Variables are:

- Pros
  - Efficiency. All Memory addressing is done with absolute addresses, making things very fast.
  - No cost to allocate and deallocate the Memory during run-time.
  - Programs can be history sensitive.
- Cons
  - Reduced flexibility. If there is a language that only has Static Variables, then recursive subprograms are impossible.
  - Memory cannot be shared between inactive and active subprograms.

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.

6.5.3.2 Stack-Dynamic Variables

Defn 62 (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 62.1. These are the variables that are most commonly used, and are usually function-local variables

Remark 62.2. 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.

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.

Defn 63 (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.

6.5.3.3 Explicit Heap-Dynamic Variables

Defn 64 (Explicit Heap-Dynamic Variable)**.** *Explicit heap-dynamic variables* are nameless (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, meaning whenever a heap value is allocated, 2 variables are created at a time. There is the Stack-Dynamic Variable Pointer/Reference and the heap-allocated *anonymous* (meaning the object itself has no Name) object. **The only way to reference the heap-object is through the Pointer/Reference Stack-Dynamic Variable.**
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`

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).

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.

6.5.3.4 Implicit Heap-Dynamic Variables

Defn 65 (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

6.6 Scope

Defn 66 (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 main(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 67 (Visible)**.** A Variable is *visible* in a statements if it can be referenced in that statement.

Remark 67.1 (In-Scope). Sometimes, a Variable that is Visible is called *in-Scope*.

Defn 68 (Local Variable)**.** A Variable is a *local variable* in a program unit or block if it is declared there. Variables defined within subprograms are also local variables. The Scope is usually the body of the subprogram in which they are defined.

Remark 68.1 (Storage Binding). If the Local Variable is a Stack-Dynamic Variable, it is bound to storage when the subprogram begins and unbound when that execution terminates.

Defn 69 (Nonlocal Variable)**.** The nonlocal variables of a program are visible with that particular program unit or block, but are not declared there.

Remark 69.1 (Global Variables). Global variables are a special case of Nonlocal Variables. These are discussed in Section 6.6.5.

6.6.1 Static Scope

Defn 70 (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)

In a brace ({})-delimited programming language, like C, if 2 things with the same name are declared at the same nested scoping level, then they are siblings. In the code example below, `sub1()` and `sub2()` are functions that are static siblings.

Remark 70.1 (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 71 (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 72 (Static Ancestor)**.** The *static ancestor*s are all the Static Parents to that particular program unit/block. The static ancestor for functions would be the Scope that contains both functions.

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()`. Even though `sub1()` calls `sub2()`, because they share the same nested Scope, the `x` in `sub2()` refers to the `var x = 3;`. This is because both `sub1()` and `sub2()` are in the same Scope, namely the function `big()`'s scope. 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.

6.6.2 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`.

6.6.3 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 74 (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 75 (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`.

6.6.3.1 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.

6.6.4 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.

6.6.5 Global Scope

Defn 76 (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.

6.7 Referencing Environments

Defn 77 (Referencing Environment)**.** The *referencing environment* of a statement is the collection of all Variables that are visible in the statement.

6.7.1 Referencing Environments in Languages with Static Scope

In a language that uses Static Scopeing, the referencing environment includes all Local Variables in its Scope and all Variables in the Static Ancestor scopes that are Visible. This also includes all function definitions and Global Variables up to that point.

The code block below and Table 6.1 illustrate how a Referencing Environment behave in a language that uses Static Scoping.

```
1  g = 3  # A global variable
2  def sub1():
3      a = 5  # Create a local variable
4      b = 7  # Create another local variable
5      ...    # EXECUTION POINT 1
6      def sub2():
7          global g  # The global variable g is assignable here now
8          c = 9     # Create a new local variable
9          ...       # EXECUTION POINT 2
10         def sub3():
11             nonlocal c  # Makes the nonlocal variable "c" visible here
12             g = 11
13             ...         # EXECUTION POINT 3
```

Execution Point	Referencing Environment
1	Locals `a` and `b` of `sub1`, the global `g`, for reference and not assignment
2	Local `c` of `sub2`, the global `g` for reference and assignment (`global` Reserved Word)
3	Local `g` of `sub3`, the nonloca `c` of `sub2` (`nonlocal` Reserved Word)

Table 6.1: Referencing Environment of a Statically Scoped Program

6.7.2 Referencing Environments in Languages with Dynamic Scope

In a language that uses Dynamic Scopeing, the referencing environment includes all the Local Variables, the Variables of all other Active subprograms, and the subprogram names. This means that some Variables in Active subprograms can be hidden from the referencing environment.

The code block below and Table 6.2 illustrate how a Referencing Environment behave in a language that uses Dynamic Scoping.

```
1  void sub1() {
2    int a, b;
3    ...       // EXECUTION POINT 1
```

```
}

void sub2() {
  int b, c;
  ...       // EXECUTION POINT 2
  sub1();
}

int main() {
  int c, d;
  ...       // EXECUTION POINT 3
  sub2();
  return 0;
}
```

Execution Point	Referencing Environment
1	`a` and `b` of `sub1`, `c` of `sub2`, `d` of `main` `c` of `main` is hidden by `sub2` and `b` of `sub2` hidden by `sub1`
2	`b` and `c` of `sub2`, `d` of `main` `c` of `main` is hidden by `sub2`
3	`c` and `d` of `main`

Table 6.2: Referencing Environment of a Dynamically Scoped Program

7 Data Types

7.1 Type Systems

Defn 78 (Type System)**.** *Type Systems* are systems that relate the concepts of *Data Types*, *Expressions*, and *values*. They describe rules for ensuring these concepts are correct.

There are some properties that type systems should have.

(i) The type system should allow us to predict the output of computations. Meaning it $e \Downarrow v$, then we can assign types to both e and to the result v.

Remark 78.1 (Expressions). In this portion of the document, an Expression refers to

$$e \Downarrow v$$

Defn 79 (Data Type)**.** A *data type* defines a subset of data values. It also specifies the set of operations possible 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

$$v : \tau \tag{7.1}$$

where v is a value and τ is a type.

Remark 79.1 (Has the Type). To say "v has the type τ"

$$v \in \tau \tag{7.2}$$

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 80 (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 81 (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 82 (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:

(i) *Type Preservation*: The predictions of the type system agree with the evaluation rules.

(ii) *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.

(iii) *Termination*: We want the type system to be *decidable*, that is, we want an automatic mechanism that performs type checking.

7.1.1 Strong and Weak Typing

Defn 83 (Strong Type Checking)**.** *Strong type checking* is a property of a programming language if and only if

(i) All values have a Data Type

(ii) **Any** Type Errors and prevents the operations that caused the Type Error from taking place, **BEFORE EXECUTION**

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

- Haskell
- Python
- Ruby
- JavaScript (This is somewhat debatable)

Remark 83.1 (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 84 (Weak Type Checking)**.** *Weak type checking* is the absence of Strong Type Checking. Meaning, that an operation that causes a Type Error does **NOT** always get prevented. In some languages, like JavaScript, the operation would return some default value and continue processing.

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

- C
- C++

7.1.2 When to Type Check

Defn 85 (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*.

For example, the following code **WILL NOT EVEN COMPILE** until the type error is fixed.

```
object StaticTypeCheck {
        def main(args: Array[String]):Unit = {
                var x = 0;
                print("Hello, World!");
                print(x[7]); // Static Type error. File won't even compile!
        }
}
```

Languages that perform static type checking are called *statically typed languages*. Such languages are:

- Haskell
- Standard ML
- C, C++, C#, Objective-C, Java, Kotlin, etc.

Remark 85.1 (Undecidability of Static Type Checking). Not all Data Types can be checked statically. For example, the subscript used in an array access or division by a variable that may be zero must be checked dynamically.

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

Defn 86 (Dynamic Type Checking)**.** Any Type Checking that is performed at runtime is called *dynamic type checking*.

For example, the following code **will** run, but will throw a type error **DURING RUNTIME**.

```
1 var x = 0;
2 print("Hello, World!");
3 print(x[7]); // Type Error here. Only found during run time of the program
```

Languages that perform dynamic type checking are called *dynamically typed languages*. Such languages are:

- Python
- JavaScript

7.1.3 Type Checking

The Type System only allows an Expression **if and only if** it can assign a Data Type to the Expression.

Defn 87 (Well-Typed)**.** An Expression is *well-typed* if and only if we can show $e : \tau$ for some type τ.

We can use rules similar to operational semantics to derive the type system, much like how we derived what computation meant. For example,

$$\frac{e_1 : Nat \quad e_2 : Nat}{e_1 + e_2 : Nat}(type\text{-}add)$$

and now the typing of values needs to be defined, because otherwise the $(type\text{-}add)$ doesn't know what types it is taking in, only what they are supposed to be for the addition to be defined.

$$\frac{v \in \mathbb{N}}{v : Nat}$$

So far, we have seen semantic type rules for operations that take in one type and return the same type. The same general formulation holds true for operations that take in one type and return another.

$$\frac{e_1 : Nat \quad e_2 : Nat}{e_1 = e_2 : Bool}$$

The above equality operator works for the natural numbers, but it won't work with booleans, so we need to make another semantic type rule about that relationship.

$$\frac{e_1 : Bool \quad e_2 : Bool}{e_1 = e_2 : Bool}$$

7.1.4 Dynamic Type Checking

Whenever a typing rule depends on the evaluation relation $\Downarrow$, we must defer type checking to runtime. Depending on the language, there might be a lot or a little amount of checking. Some languages actually defer all type checking to runtime, these are *dynamically typed* languages, like Python and JavaScript.

7.1.5 Overloading

Overload Operators are a form of Ad-Hoc Polymorphism. They rely on the typing rules of a language to determine what type of operation to perform. For example,

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 : Nat \quad v_2 : Nat \quad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}(add)$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 : String \quad v_2 : String \quad v = v_1 + + v_2}{e_1 + e_2 \Downarrow v}(concat)$$

7.2 Primitive Data Types

Defn 88 (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.

7.2.1 Numeric Types

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

Defn 89 (Numeric Data Type)**.** *Numeric data type*s are Data Types that handle numbers.

7.2.1.1 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 90 (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 90.1. Using Twos Complement is similar to adding by a negative number to an integer instead of performing subtraction.

7.2.1.2 Floating-Point

Defn 91 (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 92 (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 93 (Floating-Point Range)**.** *Range* is a combination of the range of fractions and the range of the exponents.

7.2.1.3 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`.

7.2.1.4 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 94 (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 94.1. These numbers are usually stored 2 per byte, because each only takes 4 bits.

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 7.1: Binary Coded Decimal

7.2.2 Boolean Types

Defn 95 (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 95.1 (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.

7.2.3 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.

7.3 Character String Types

Defn 96 (Character String Type)**.** A *character string type* is one in which the values consist of sequences of characters. In most programming languages, these are simply refered to as `string`s.

7.3.1 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?

7.3.2 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 97 (Regular Expression)**.** A *regular expression*, sometimes called a *regex* is a way to define a sequence of characters to form strings.

7.3.3 String Length Options

Defn 98 (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 99 (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 100 (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.

7.3.4 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.

7.3.5 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

7.4 User-Defined Ordinal Types

Defn 101 (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 101.1. There are 2 User-Defined Ordinal Types that are supported by most programming languages:

- Enumeration Types
- Subrange Types

7.4.1 Enumeration Types

Defn 102 (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 102.1. 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?

7.4.1.1 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.

7.4.1.2 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.

7.4.2 Subrange Types

Defn 103 (Subrange Type)**.** A *subrange type* is a contiguous sequence of an Ordinal Type. For example, this is a subrange: `12..14`.

7.4.2.1 Ada's Design

Ada included Subrange Types in Subtypes.

Defn 104 (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.

7.4.2.2 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.

7.4.3 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.

7.5 List Types

Lists were first supported in LISP.

Defn 105 (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 105.1. 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 with immutable nodes. 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))
```

The empty list `()` is also denoted as `nil`. `nil` also serves as the `false` value of the language, and everything else is `true`.

Defn 106 (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)]
```

7.6 Arrays

Defn 107 (Array)**.** An *array* is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element. The individual elements of an array are of the same Data Type. References to individual array elements are specified using subscript expressions

Defn 108 (Length)**.** The *length* of an Array is defined to be the number of storage elements present in it. This means that a list that is declared to have 10 elements, but none assigned, has a length of 10.

Remark 108.1 (The Empty Array). *The empty array* is defined to have a length of 0.

Defn 109 (Sparse)**.** If an array is *sparse*, it means that not all elements present in an array are filled with user-input data. For example, if you have an array with Length 10 in JavaScript, you can add an element to position 50, to create an array that now has Length 51, with only 11 elements.

7.6.1 Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are jagged and/or rectangular multidimensional arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What king of slices are allowed, if any?

7.6.2 Arrays and Indices

Specific elements in an array are referenced by means of the name of the aggregate and a dynamic selector, known as *subscript*s or *indices*. If all of the subscripts used are constants, the selector is static; otherwise, it is dynamic. Arrays are sometimes called *finite mapping*s, because they map Memory cells to values, with a finite length.

Most languages use brackets, `[` and `]`, to denote the array indices. However, some languages use parentheses.

The type of the subscripts are usually integers, but Ada also allows any Ordinal Type to be used as a subscript. Some languages check the bounds of the array accesses through subscripts, though some don't. Some languages allow the index to be a negative integer, in which case, it is the index of that element starting from the end as 0.

7.6.3 Subscript Bindings

The binding of the subscript type to an Array variable is usually static, but the value ranges are sometimes dynamic. Some languages have an implicit lower bound on the subscript range, usually 0. However, some languages allow for negative subscripting, which starts their indexing from the end of the Array

7.6.4 Array Categories

There are 5 categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated.

1. Static Array
2. Fixed Stack-Dynamic Array
3. Stack-Dynamic Array
4. Fixed Heap-Dynamic Array
5. Heap-Dynamic Array

7.6.4.1 Static Arrays

Defn 110 (Static Array)**.** A *static array* is one in which the subscript ranges are statically bound and the storage allocation is static. Meaning the entire array, other than the values it contains are created before runtime.

The advantages and disadvantages of this are:

- Advantages
  - Efficiency, there is no dynamic allocation or deallocation overhead required.
- Disadvantages
  - Flexibility, the storage for the array is fixed for the entire execution of the program.

7.6.4.2 Fixed Stack-Dynamic Arrays

Defn 111 (Fixed Stack-Dynamic Array)**.** In a *fixed stack-dynamic array*, the subscript ranges are statically bound, but the storage allocation is done at declaration elaboration time during runtime.

The advantages and disadvantages of this are:

- Advantages
  - If 2 subprograms both have large arrays, they can use the same space, so long as only a single one is running at a time.
- Disadvantages
  - The time required to allocate and deallocate the array.

7.6.4.3 Stack-Dynamic Array

Defn 112 (Stack-Dynamic Array)**.** In a *stack-dynamic array*, the subscript ranges and storage allocation are done during declaration elaboration time during program execution. However, once the subscript ranges are bound and the storage allocatec, they are both fixed throughout the lifetime of the array.

The advantages and disadvantages of this are:

- Advantages
  - Flexibility, the size of the array doesn't need to be known until just before the array is used.
- Disadvantages
  - The time required to allocate and deallocate the array.
  - Determine the subscript ranges and bind them.

7.6.4.4 Fixed Heap-Dynamic Array

Defn 113 (Fixed Heap-Dynamic Array)**.** A *fixed heap-dynamic array* is similar to a Fixed Stack-Dynamic Array, in that subscript ranges and storage binding are done on demand at runtime, and are fixed throughout the array's lifetime. However, the storage is allocated from the heap instead of the stack.

The advantages and disadvantages of this are:

- Advantages
  - Flexibility, the array can be any size to fit any problem.
- Disadvantages
  - The allocation and deallocation time is much longer on the heap compared to the stack.

7.6.4.5 Heap-Dynamic Array

Defn 114 (Heap-Dynamic Array)**.** A *heap-dynamic array* is one in which the binding of subscript ranges and storage allocation is done during program execution, **and can change any number of times during the array's lifetime**.

The advantages and disadvantages of this are:

- Advantages
  - Flexibility, arrays can grow and shrink during program execution to fit the needs of the problem.
- Disadvantages
  - Allocation and deallocation take longer on the heap.
  - Allocation and deallocation may happen several times during a program's execution.

7.6.5 Array Initialization

Some languages allow the programmer to initialize the Array with values when the storage is allocated.

```
1 Integer, Dimension (3) :: List = (/0, 5, 5/)
```

However, C/C++, Java, and C# do not allow programmer-specification length of an Array during declaration/initialization.

```
1 int list [] = {4, 5, 7, 83};
```

In C and C++, a Character String Type or string, is a character array with the last element being a null terminator ASCII 00.

Ada allows for initialization to specific indices in an array during storage allocation.

```
1 List : array (1..5) of Integer := (1, 4, 5, 7 ,9);
2 Bunch : array (1..5) of Integer := (1 => 17, 3 => 34, others => 0);
```

Because of this feature, the `others` clause initializes all other elements with a "default" value.

7.6.6 Array Operations

Defn 115 (Array Operation)**.** An *array operation* is one that operates on an Array as a whole. The most common operations are:

- Assignment
- Concatenation
- Comparison for Equality
- Slices

Some programming languages offer built-in support for these common operations, but some don't. There is also no standard set of symbols used to perform these operations.

7.6.7 Rectangular and Jagged Arrays

Defn 116 (Rectangular Array)**.** A *rectangular Array* is an multidimensional Array in which all of the rows have the same number of elements, and all the columns have the same number of elements. Rectangular Arrays model rectangular tables and square matrices exactly.

Defn 117 (Jagged Array)**.** A *jagged Array* is a multidimensional Array in which the lengths of the rows do not need to be the same. This applies to all Arrays in higher dimensions as well. These are made possible with having an array, where each element is itself, an Array.

7.6.8 Slices

Defn 118 (Slice)**.** A *slice* of an Array is some substructure of that Array. A slice is **not** a new Data Type, rather, it a mechanism for addressing a portion of an Array as a single unit.

Each language has a different way to Slice an Array, and the language's documentation should be looked at to determine how they are done.

7.6.9 Evaluation

Arrays are supported by nearly all programming languages, with APL even being built around doing Array processing. The greatest progress has occurred with using ordinal types as subscript types, Slices, and dynamic Arrays.

7.6.10 Implementation of Array Types

Implementing Arrays requires considerable compile-time effort. The code to access the elements must be generated at compile-time. At runtime, the code must be executed to produce element addresses. There is no way to precompute the address to be accessed by a reference such as `list[k]`. The only time any computation can be done on an Array at compile-time is if the array is statically bound.

To access an element in an Array:

$$\text{address}(\texttt{list[k]}) = \text{address}(\texttt{list}) + (k * \texttt{elementSize}) \tag{7.3}$$

The compile-time descriptor for a single-dimensioned Array is:

- Array name
- Element Data Type
- Index Data Type
- Index Lower Bound
- Index Upper Bound
- Array's starting address

If a language uses dynamic range checking on the index, then description must be included with the program during its execution.

7.6.10.1 Multidimensional Arrays If true multidimensional Arrays are being implemented, not just an array of arrays, things get more complicated. This is because of the multidimensional nature of the Array and the one dimensional nature of Memory. There are 2 ways to organize multidimensional Arrays in Memory.

1. Row Major Order
2. Column Major Order

Defn 119 (Row Major Order)**.** In *row major order*, the elements of the Array that have their first subscript to be the lower bound value of the subscript are stored first, folled by the elements of the second value of the first subscript.

For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

would be stored as

$$1, 2, 3, 4, 5, 6, 7, 8, 9$$

Defn 120 (Column Major Order)**.** In *column major order*, the elements of an array that have their last subscript to be the lower bound value of the subscript are stored first, followed by the elements of the second value, and so on.

For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

would be stored as

$$1, 4, 7, 2, 5, 8, 3, 6, 9$$

The compile-time descriptor for a multidimensional array includes:

- Multidimensional Array name
- Element Data Type
- Index Data Type
- Number of dimensions
- Index Range
- $\vdots$
- Index range $n - 1$
- Starting address of the array

7.7 Associative Arrays

Defn 121 (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 121.1 (Alternative Names)*.* There are many alternative names for Associative Arrays.

- Hashtable
- Hashmap
- Associative Array
- Dictionary

Remark 121.2 (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.

7.7.1 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.

7.7.2 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.

7.8 Record Types

Defn 122 (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 122.1 (Clarification)*.* An important clarification here is that the Record, defined in Definition 122 is **NOT** a record in a database **in any way**.

Remark 122.2*.* 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 122.3 (Record vs. Object)*.* A Record and an object are quite similar. However, the differences between them depend on the language.

Defn 123 (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.

7.8.1 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.

7.8.2 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 124 (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 125 (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 126 (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.

7.8.3 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.

7.8.4 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.

7.9 Tuple Types

Defn 127 (Tuple)**.** A *tuple* is a Type that is similar to a Record, except the elements are **not** named. Instead, they are integer-indexed, like Arrays.

Remark 127.1. Tuples can be used to return multiple values from a function in languages that do not natively support that.

Remark 127.2. Python natively supports these. These behave similarly to Lists, but are immutable.

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

7.10 Union Types

Defn 128 (Union)**.** A *union* is a Data Type whose Variables may store different Type Values at different times during program execution. **However, ONLY ONE value is stored at a time.**

This means that in a C-program, which uses Free Unions will only hold one thing at a time, and **based on the field that you call in the union, the program will interpret the bit pattern differently**. This site gives a good exposition of this point: Greater Exposition.

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.

7.10.1 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?

7.10.2 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 129 (Free Union)**.** A *free union* is a Union that have ***NO*** Type Checking enforced on their use. For example, this C snippet:

```
union flexType {
    int intEl;
    float floatEl;
};
union flexType el1;
float x;
// Some more code here...
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 130 (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.

7.10.3 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 131 (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.

```
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;
        -- More code here --
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
        Figure_1 := (Filled => True,
                                Color => Blue,
                                Form => Rectangle,
                                Side_1 => 12,
                                Side_2 => 3);
```

7.10.4 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.

7.10.5 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.

7.11 Pointer and Reference Types

Pointers and References do are not structured types, like an Array. Nor are they scalar Values, because they are used to reference some other Variable, rather than storing its own data. These are called:

- Reference Types (Not directly related to References)
- Value types

Both of these add Writability to the language.

Defn 132 (Pointer)**.** A *pointer* type is one in which the variables have a range of values that consists of Memory addresses and a special value, `nil`/`null`. The `nil`/`null` is not a valid address and is used to indicate that a pointer cannot currently reference a Memory cell.

Pointers have 2 distinct uses:

1. Provide some of the power of indirect addressing
2. Pointers provide a way to manage dynamic storage, the Heap.

Defn 133 (Heap-Dynamic Variable)**.** Variables dynamically allocated from the Heap are called *heap-dynamic variables*. They often do not have identifiers associated with them, thus can only be referenced by Pointer or Reference type variables. These Variables are technically Anonymous Variables.

Defn 134 (Anonymous Variable)**.** An *anonymous Variable* is one that does not have a name/identifier associated with it.

7.11.1 Design Issues

- What are the scope and lifetime of a Pointer?
- What is the lifetime of a Heap-Dynamic Variable (The value a Pointer points to)?
- Are Pointers restricted as to the type of value to which they can point? (Can they only point to integers?)
- Are Pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support Pointers, References, or both?

7.11.2 Pointer Operations

There are 2 main operations that most languages support,

1. Assignment: Set a Pointer Variable Value to some useful address.
2. Dereferencing

If a Pointer shows up in an expression, there are 2 ways to interpret it:

1. Use the address stored by the Pointer as an operand
2. Use the value that the address being stored by the Pointer as an operand. Perform a Dereferencing on the value stored in the Pointer.

Defn 135 (Dereferencing)**.** *Dereferencing* a Pointer takes a reference through one level of redirection. This can be implicit or explicit.

In C/C++, an explicit dereferencing is done with the Prefix Unary `*` operator.

Remark 135.1 (Pointer Dereferencing a Record). When a Pointer points to a Record, the syntax of a reference to the Fields in the Record varies by language.

If a language supports using Pointers to manage the Heap, there must be an explicit allocation operation. Some languages use a subprogram (`malloc` in C). Object-Oriented languages often use the `new` Reserved Word/Keyword. If a language does not support implicit deallocation of Memory, a `delete`/`free` operator/subprogram must be provided to reclaim the unused Memory.

7.11.3 Pointer Problems

Due to the problems that Pointers can introduce, some languages only support References and utilize implicit deallocation.

7.11.3.1 Dangling Pointers

Defn 136 (Dangling Pointer)**.** A *dangling Pointer*, or a *dangling Reference* is a Pointer/Reference that contains the address of a Heap-Dynamic Variable that has already be deallocated. These are dangerous for several reasons:

- That Memory might be in-use by some other Heap-Dynamic Variable.
- Type Checking might be invalid, if the types of the things stored were different.
- There is potentially no relationship between the value stored in Memory and where the Pointer points to.
- Possible that location is being temporarily used by the storage management system.

7.11.3.2 Lost Heap-Dynamic Variables

Defn 137 (Lost Heap-Dynamic Variable)**.** A *lost Heap-Dynamic Variable* is an allocated Heap-Dynamic Variable that is no longer accessible to the user program.

This is sometimes refered to as *garbage*.

Defn 138 (Memory Leak)**.** A *memory leak* is a case of when a Lost Heap-Dynamic Variable does not get collected by the runtime system, or the programmer, causing some Memory to be in-use, inaccessible, and potentially program-threatening.

7.11.4 Pointers in Ada

Ada's Pointers are called *access* types. The Dangling Pointer problem is partly solved here, by allowing for implicit deallocation of a Heap-Dynamic Variable at the end of its Pointer scope. However, few compilers implement this, so this is mostly theory. This is mostly handled by the fact that Heap-Dynamic Variables can only be access by Variables on one type. When the end of scope for that type declaration is reached, no Pointers can be left pointing at the Heap-Dynamic Variable. However, Ada does have support for an explicit deallocator, `Unchecked _ Deallocation`.

The Lost Heap-Dynamic Variable problem was not solved in Ada.

7.11.5 Pointers in C/C++

C/C++ do not offer a solution for the Dangling Pointer or Lost Heap-Dynamic Variable problems. They are more like the way assembly languages handle addresses. This means they are extremely flexible, but must be used with great care.

Because Pointers in C/C++ are stored as a value, they can have some basic arithmetic done on them. This is how Array on the Heap are constructed. C/C++'s Pointers can point to functions, which is used to pass functions as parameters to other functions.

In C/C++, the `*` operator denotes a dereferencing operation, and the `&` operation denotes the operator for producing the address of a Variable.

```
1 int *ptr; // Declare a pointer that will point to a thing of int type
2 int count, init; // Declare some integer variables
3 ...
4 ptr = &init; // Put the memory address of "init" into ptr
5 count = *ptr; // Dereference the address in "ptr" and take the value there and assign to count's storage
```

Remark (`void*` Pointer Type)*.* The `void*` type on a Pointer means that it will accept a Pointer of any Data Type.

7.11.6 Reference Types

Defn 139 (Reference)**.** A *reference* type Variable is similar to a Pointer, in that it points to something. The difference comes down to what it points to. A Pointer points to an address in Memory. A reference refers to an object or value in Memory. A reference **can only refer to one thing**, i.e. the Memory address it refers to cannot change.

In C/C++, they are declared with the `&` operator.

Remark 139.1 (Memory Address Arithmetic)*.* Therefore, address arithmetic cannot be done on references.

In C/C++, there is a special Reference type used for Formal Parameters in function definitions. It is a constant Pointer that is always implicitly dereferenced. These allow for the Actual Parameters to operate as In/Out Mode parameters. This can also be achieved with Pointers, but each use requires the Pointer be explicitly dereferenced.

7.11.7 Evaluation

Pointers are like the `goto` statement. They widen the range of Memory cells that can be referenced by a Variable. They may be dangerous, but they are essential for the lowest levels of programming: device drivers, operating systems, kernels, etc. However, it does not make sense for all applications, which is why some other languages explicitly disallow Pointers from being used at all.

7.11.8 Implementation

Pointers are usually used in Heap management.

7.11.8.1 Representations of Pointers and References In most computers today, Pointers and References are single values store in Memory cells. However, this differs in earlier computers.

7.11.8.2 Solutions to the Dangling-Pointer Problem There are several solutions, including:

1. Tombstones
   - Pointers and References point to a *tombstone*, which points to the actual value.
   - When a Heap-Dynamic Variable is deallocted, the tombstone is set to `nil`/`null`, indicating the Heap-Dynamic Variable no longer exists.
   - The problem with this is that it is expensive in time and space.
     - The tombstones are never deallocted, so they will take up space until execution ends.
     - Every access to anything on the Heap requires an extra redirection, which may be another clock cycle.
2. Locks-and-Keys Approach
   - The Pointer/Reference has a (`key`, `address`) pair.
   - The Heap-Dynamic Variable has a header with a `lock`.
   - During execution, if a Pointer is dereferenced, its `key` is compared to the `lock`, and if they match, the execution happens.
     - If they don't match, then a runtime error is raised.
   - When the Heap-Dynamic Variable is deallocated, the `lock` value is set to an illegal lock value.
   - If a Pointer is dereferenced to that particular `lock`, a runtime error is raised.

The best thing to do is not have programmers handle Heap-Dynamic Variables themselves, and have the runtime system do handle them implicitly.

7.11.8.3 Heap Management In this class, there are 2 ways to manage the Heap.

1. Reference Counters
2. Mark-Sweep

Defn 140 (Reference Counter)**.** In a *reference counter* garbage collection algorithm, every Heap-Dynamic Variable has a count attached to it, with the number of Pointers pointing to it. So long as the count does not get smaller than 1, the Heap-Dynamic Variable is reachable by the program. If the count reaches 0, then the Heap-Dynamic Variable is deallocated.

Remark 140.1 (Problems). • Space: A little extra space is required to keep track of the count
- Time: A little bit of time is required to update the counter everytime a Pointer is created/destroyed.
- In a circular structure, the Pointer count will never reach zero, so an unreachable structure might not be deallocated.

Remark 140.2 (Advantages). • It is an incremental algorithm, so there is no time when a large amount of time is required to process the Heap.

Defn 141 (Mark-Sweep)**.** The *mark-sweep* garbage collection algorithm starts by assuming that everything is unreachable from the program, and marks it as such. First, it "unmarks" the registers, Call Stack, Global Variables as reachable. Then, it follows all pointers present in the Call Stack to the Heap. If the Heap-Dynamic Variable is reachable, then it any Pointers it may have are followed. Anything on the Heap that is still reachable after the mark phase is saved.

In the sweep phase, anything that is still marked as unreachable is deallocated.

Remark 141.1 (Problems). • Space: The heavy recursion while inside the Heap requries a lot of Call Stack space.
- Time: Since this is run when the Heap is nearly full, it takes a lot of time to handle.
  - This leads to the main program being executed to be "paused".

7.12 Type Equivalence

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

Defn 142 (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 143 (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 144 (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 144.1. 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 145 (Primitive Equivalence)**.** *Primitive equivalence* directly compares two values by comparing their bit patterns in memory.

Defn 146 (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 147 (Reference Equivalence)**.** *Reference Equivalence* checks whether two pointers/references point to the same address in Memory.

Defn 148 (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 149 (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.

```
type Celsius is new Float;
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 150 (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.

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

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

8 Advanced Data Types

Static Type Checking is favorable because it allows us to catch Type Errors early. Overall, this contributes to the robustness and Reliability of a language. We attempt to assign a Data Type to every value and Expression. This is usually done with static typing rules as automation and explicit use specification.

However, static typing cannot solve all our issues in programming. For instance, if we wanted to exclude division by zero, we might require that the divisor be non-zero. However, this limits our ability to do computation.

There are 3 main trade-offs that we need to make with Data Types:

1. **Precision**: How accurately can we make types capture the behaviour of a value, Expression, Subprograms, or other program concept? This concept ties into the question of Increasing Reliability.
2. **Automation**: How much type-checking can we do while still being certain that the type-checking mechanism will eventually (and, ideally, quickly) finish? This concept ties into the question of Reducing Compile-Time Cost and more generally Reducing Development Cost.
3. **User-Friendliness**: At what point does writing and reading types become too unwieldy to be practical for users? This concept ties into several of the Readability and Writability criteria.

Defn 151 (Polymorphism)**.** *Polymorphism* is the idea that part of a program can have multiple forms, i.e. a single program handles multiple cases of input types.

There are 3 types of polymorphism that are discussed here:

1. Parametric Polymorphism
2. Ad-Hoc Polymorphism
3. Subtype Polymorphism

Defn 152 (Conservative)**.** If a Type System cannot precisely express the possible values a Data Type can take, and forces us to describe the values more generally, this is a *conservative* Type System.

8.1 Parametric Polymorphism

Defn 153 (Parametric Polymorphism)**.** *Parametric polymorphism* can be summarized with the following phrase; "This value has a Data Type, but you don't need to know what it is".

Remark 153.1 (Generics). Outside of the Functional Programming Language world, Parametric Polymorphism is usually refered to as *generics*. Subprograms that make use of these generics (Type Parameters) are called *generic subprograms*.

An example of this is a function that creates a list of fixed size where all elements are initialized with a programmer-specified initial value. The code shown below is *invalid* Scala code, but illustrates the concept.

```
def makeIntArray(len : Int, initialVal : Int) = {
  val result = new Array(len); // Create an array with 'len' entries
  for(i <- 0 to (len-1)) {
    result.update(initialVal);
  }
  return result;
}

def makeFloatArray(len : Int, initialVal : Float) = {
```

```
  val result = new Array(len); // Create an array with 'len' entries
  for(i <- 0 to (len-1)) {
    result.update(initialVal);
  }
  return result;
}
```

This is even more apparent if we write a couple of identity functions. These are functions that take their parameter and just return it, and do nothing else. Again, written in Scala, they are:

```
def idInt(x : Int) : Int = x
// idInt takes a parameter with type Int and returns an value of type Int

def idStr(x : Str) : Str = x
// idStr takes a parameter with type Str and returns an value of type Str
```

The way to solve this is with Type Parameters.

Defn 154 (Type Parameter)**.** The *type parameter* language mechanism abstracts over types that are already present in the program, so we can use these types without knowing their exact form. In many ways, type parameters mirror traditional subprogram Parameters.

There are 2 forms of type parameters:

1. Formal Type Parameters
2. Actual Type Parameters

Using the definition of a Type Parameter system, we can rewrite the identity functions, in Scala, as:

```
def id[T](x : T) : T = x
// id takes a parameter with a type parameter T and returns an value of the same type provided, T

/* In Scala, these parametric polymorphic functions are called with
 * functionName[ActualTypeParameter](ActualParameters)
 */
val three = id[Int](3); // The variable three has the Int value 3
val hello = id[Str]("Hello"); // The variable hello has the Str value "Hello"
```

For those familiar with Java, this Type Parameter system has the syntax

```
public class C {
        // public static returnType methodName <formalTypeParameter> (formalParameters)
        public static T id <T> (T x) {
                return x;
        }

        int three = C.id<Integer>(3);
        String hello = C.id<String>("Hello");
}
```

Remark. The code examples of Parametric Polymorphism above make it seem like it is only possible to write these Subprogram Definitions with just one Formal Type Parameter. However, just like normal subprogram Parameters, you can pass as many Actual Type Parameters as you would like.

Take this use of a HashMap in Java as an example.

```
/* The definition of Java's HashMap function is shown below (this comes from Java 8's documentation.
 * java.util.HashMap<K,V>
 */

import java.util.HashMap;
```

```
6
7  public class C {
8      private HashMap<String, Integer> strIntHashMap;
9      private HashMap<String, Double> strIntHashMap;
10
11     public C() {
12         // Create a HashMap that uses Strings as its keys and is storing Integers as its values.
13         HashMap<String, Integer> strIntHashMap = new HashMap<String, Integer>();
14         HashMap<String, Double> strDblHashMap = new HashMap<String, Double>();
15     }
16 }
```

Defn 155 (Formal Type Parameter)**.** A *formal type parameter* are the Type Parameters present in the Subprogram Header. These are type variables that can be used freely throughout the body in all the places where the Data Type can vary, but should act the same.

Remark 155.1 (Type Parameter). Formal Type Parameters are sometimes colloquially referred to as *type parameter*s. In this case, they are related to the definition of Type Parameters (Definition 154), but they are only they "placeholder" types.

Defn 156 (Actual Type Parameter)**.** The *actual type parameter* is the actual Data Type provided to a function that uses/requires a Formal Type Parameter.

Remark 156.1 (Type Variable). Actual Type Parameters are sometimes colloquially referred to as *type variable*s. These share a relationship with normal Variables, but they do not necessarily have all the same properties as normal Variables. However, these functions can "vary" depending on what the programmer specifies.

8.2 Ad-Hoc Polymorphism

Defn 157 (Ad-Hoc Polymorphism)**.** *Ad-Hoc polymorphism* can be summarized by the following phrase; "This value has a Data Type, and you don't need to know what it is, but here are a few things you can do with it".

Depending on the language, this can be achieved with Typeclasses.

To motivate this discussion, we need to refer back to the previous section on Parametric Polymorphism and the code block below.

```
1  fn max2<T> (x : T, y : T) -> T {
2          if x > y {
3                  return x;
4          } else {
5                  return y;
6          }
7  }
```

However, Rust will complain about this code, and suggest that we must add a Type Constraint to bound the Type Parameter `T` by a Rust trait. This is because the Data Type `T` could be **ANY** type present in the programming language/project. For example, finding the maximum of a string is an ambiguous operation. What does it mean to find the "maximum string"?

The trait that is used to bound `T` is `std::cmp::PartialOrd`. This means that **ANY** type `T` that we feed into the `max2` function **MUST** be bounded by `std::cmp::PartialOrd`, meaning the definition of relational operators (`<`, `>`, etc.) is defined. Knowing this, we rewrite our `max2` code as shown below, and attempt to use it on something that does not have its type bounded.

```
1  fn max2<T> (x : T, y : T) -> T where T: std::cmp::PartialOrd {
2          if x > y {
3                  return x;
4          } else {
5                  return y;
6          }
7  }
8
9  // max2 compiles successfully!
10
11 // Now assume we have a record constructed with some fields in it.
```

```
struct myRecord { ... }

// We perform some operations.
let r1 : myRecord;
let r2 : myRecord;
let r3 : myRecord = max2::<myRecord>(r1, r2); // COMPILATION FAILS HERE
/* This is because r1 and r2 are NOT bounded by PartialOrd, so max2 doesn't know how to compare them.
 */
```

Defn 158 (Type Constraint)**.** A *type constraint* is a method of constraining a Data Type to **ONLY** Data Types that have certain characteristics about them, functions defined, etc. This is used to bound the possible types that can be used by certain things to ensure that all the operations specifed in a portion of a program are defined for any possible Data Type that can be fed in.

Defn 159 (Typeclass)**.** A *typeclass* is a form of a Type Constraint. It was first introduced by the Haskell programming language and has since been introduced to other programming languages as well. The name typeclass is language-agnostic, meaning other languages might call these something else.

For example, the Rust Trait `std::cmp::PartialOrd` requires that the implementing type define the meaning of the greater than, $>$, operator.

Remark 159.1 (Rust Traits). The Rust programming language calls Typeclasses *trait*s. The traits used in Rust **are not the same** as the ones in Scala. They just so happen to have the same name.

Remark 159.2 (Typeclass/Class Confusion). **BE CAREFUL!** Typeclasses and Classes are **NOT** related to each other and are fundamentally different.

8.2.1 User-Defined Typeclasses

Most programming languages that support Typeclasses allow for the programmer to specify their own Typeclasses for use in their programs to bound their Data Types. In this section, we are going to construct another `max` function, but this one will use a greater-than operator that we define. By creating a new operator for ourselves, we need to bound the possible inputs to the operator to ensure that it works for all the Data Types we expect to put into it.

8.3 Subtype Polymorphism

Defn 160 (Subtype Polymorphism)**.** *Subtype polymorphism* can be summarized with the following phrase; "This value has a Data Type, and while you don't know what exact type it is, this Data Type is a more restricted (or general) version of another Data Type that you *do* know, so therefore there are some things that you can do with it".

Defn 161 (Subtype)**.** A Data Type T is a *subtype* of type U, denoted $T <: U$ or $U :> T$, if any value $v : T$ can be used in any context that requires a value of type U.

Some examples of subtypes are:

- $[2\texttt{T}05] <: [2\texttt{T}06]$
- $[2\texttt{T}05] <: [1\texttt{T}05]$
- $[2\texttt{T}05] <: [1\texttt{T}06]$
- $[2\texttt{T}05] <: \texttt{INTEGER}$

This is because the supertype contains all possible values that the subtype can take, and more. These are all based on subset relations, meaning that typing inherits a few properties.

(i) Subtyping is *reflexive*. Each type T is a subtype and supertype of itself, i.e. $\texttt{T} <: \texttt{T}$.
(ii) Subtyping is *transitive*. Meaning if we know $\texttt{T} <: \texttt{U}$ and $\texttt{U} <: \texttt{V}$, then $\texttt{T} <: \texttt{V}$ is true.

8.3.1 Typing Conversions

Defn 162 (Widening Conversion)**.** Whenever we use a value of a Subtype in a place that expects a supertype, the language must perform a *widening conversion*.

Remark 162.1 (Implicit). Most languages make this an implicit operation, because no information is lost, making it a type-safe operation.

Defn 163 (Narrowing Conversion)**.** Some languages also allow us to translate a supertype to a subtype, using a *narrowing conversion*.

Remark 163.1 (Explicit). The languages that support Narrowing Conversions make it an explicit command. This instruction may fail, or may lead to information loss, meaning it is not type-safe. So the language designers tend to make it an explicit operation.

8.3.2 Subtyping and Records

Take the following 2 types, written in C, as the example for this section.

```c
struct R1 {
        int x;
};

struct R2 {
        int x;
        float y;
};
```

Any function that uses R1 would also work on any record of type R2. For these, don't think about the size of the actual record, but the possible information that can be stored and the potential operations they can have.

- Both records can have their `x` field accessed and modified, and both of these are they same type themselves, `int`.
- R2 can have its `y` field accessed and modified.
  - Thus, we can do everything that R1 can do with R2, making R2 a Subtype of R1

$$\texttt{R2} <: \texttt{R1}$$

8.3.3 Subtyping and Subprograms

In a given subprogram, with Formal Parameter inputs and some return values, there are a few ways we can use subtyping to change the operation of the subprogram.

If we need a subprogram with type $T_2 \to U_2$, we can instead provide a subroutine of type $T_1 \to U_1$ if T_1 is **less constrained** than T_2 and if U_1 is **more constrained** than U_2. Meaning, for a subprogram, we can put **supertypes of the parameter's type** into the subprogram and it will still work. Similarly, if we return **subtypes of the return type**, the subprogram will still work.

In other words, if we think of the function in the middle as a funnel whose opening we can widen and whose end we can narrow.

This relationship can be represented by Equation (8.1), called the *Arrow Rule*.

$$\frac{T_1 :> T_2 \;\; U_1 <: U_2}{T_1 \to U_1 <: T_2 \to U_2} \tag{8.1}$$

Remark. However, this **does not** hold true when the data is mutable!!

8.3.4 Subtyping in Languages

Defn 164 (Structural Subtyping)**.** A type constructor in a language uses *structural subtyping* if two different types T, U constructed from this type constructor can be in a subtyping relation without being explicitly declared to be in a subtyping relation.

Defn 165 (Nominal Subtyping)**.** A language uses *nominal subtyping* for a type constructor if two different types T, U constructed from this type constructor can be in a subtyping relation, but only if they are explicitly declared to be in this relation.

Remark. The advantages and disadvantages of structural and nominal subtyping are analogous to those of structural and nominal type equivalence.

8.3.5 Variance of Types

Using a `Box` Scala trait as the basis of our discussion, the code is shown below.

```scala
trait Box[T] {
  def put(v : T)
  def get() : T
}
```

Defn 166 (Covariance)**.** Let τ be a type constructor with formal type parameters $\tau_1, \ldots, \tau_k$, such that $T = \tau[\tau_1, \ldots, \tau_k]$ is a type. Let $i \in \{1, \ldots, k\}$.

If for all $\tau_i' <: \tau_i$ we can always substitute a value of type $\tau[\tau_1', \ldots, \tau_i', \ldots, \tau_k']$ in a context that expects a value of type $\tau[\tau_1, \ldots, \tau_i, \ldots, \tau_k]$ without violating type preservation then τ_i is *covariant* in T.

Using the Box example from above, we can create an *covariant* Scala trait by writing something like this,

```
trait CovariantBox[T] {
  def get() : T
}
```

When we vary the type parameter T towards a subtype, the type of ReadBox[T] also varies towards that of a subtype; we say that T is **covariant**.

Namely, if $A <: B$ and $C[\alpha]$, and $C[A] <: C[B]$, then α is covariant.

Defn 167 (Contravariance)**.** Let τ be a type constructor with formal type parameters $\tau_1, \ldots, \tau_k$, such that $T = \tau[\tau_1, \ldots, \tau_k]$ is a type. Let $i \in \{1, \ldots, k\}$.

If for all $\tau_i' :> \tau_i$ we can always substitute a value of type $\tau[\tau_1', \ldots, \tau_i', \ldots, \tau_k']$ in a context that expects a value of type $\tau[\tau_1, \ldots, \tau_i, \ldots, \tau_k]$ without violating type preservation then τ_i is *contravariant* in T.

Using the Box example from above, we can create an *contravariant* Scala trait by writing something like this,

```
trait ContravariantBox[T] {
  def put(v : T)
}
```

When we vary the type parameter T towards a subtype, the type of WriteBox[T] varies in the opposite direction, towards that of a supertype; we say that T is **contravariant**.

Namely, if $A <: B$ and $C[\alpha]$, and $C[A] :> C[B]$, then α is contravariant.

Defn 168 (Invariance)**.** Let τ be a type constructor with formal type parameters $\tau_1, \ldots, \tau_k$, such that $T = \tau[\tau_1, \ldots, \tau_k]$ is a type. Let $i \in \{1, \ldots, k\}$.

If τ_i is neither covariant nor contravariant in T, then τ_i is *invariant* in T.

Using the Box example from above, we can create an *invariant* method by writing something like this,

```
def invariantBox(box : Box[B], b : B) {
  val v : B = box.get()
  box.put(b)
}
```

- `A` :> `B`: For example, consider `A = INTEGER` and `B = [1 TO 10]`. In this case, `box.put(b)` is safe, as `Box[A]` can store any number. However, `box.get()` might now return the number 99, which does not fit into the variable `b`. **Thus, this option is not statically type-safe.**
- `A` <: `B`: For example, consider `A = [1 TO 10]` and `B = INTEGER`. In this case, `box.get()` works, but `box.put(b)` does not: `b` might be 99, which we cannot pass to an operation that only accepts `A` as parameter. **Thus, this option is not statically type-safe either.**

Namely, if $A <: B$ and $C[\alpha]$, and $C[A] \neq C[B]$, then α is invariant.

Defn 169 (Bivariance)**.** Let τ be a type constructor with formal type parameters $\tau_1, \ldots, \tau_k$, such that $T = \tau[\tau_1, \ldots, \tau_k]$ is a type. Let $i \in \{1, \ldots, k\}$.

If for all $\tau_i' <: \tau_i$ we can always substitute a value of type $\tau[\tau_1', \ldots, \tau_i', \ldots, \tau_k']$ in a context that expects a value of type $\tau[\tau_1, \ldots, \tau_i, \ldots, \tau_k]$ without violating type preservation **AND**, if for all $\tau_i' :> \tau_i$ we can always substitute a value of type $\tau'[\tau_1', \ldots, \tau_i', \ldots, \tau_k']$ in a context that expects a value of type $\tau[\tau_1, \ldots, \tau_i, \ldots, \tau_k]$ without violating type preservation then τ_i is *bivariant* in T.

Remark 169.1. If both the input and output allow for the type to be both broadened and narrowed, it is not terribly interesting to study. Thus, we will not be studying them in much detail in this class.

Namely, if $A <: B$ and $C[\alpha]$, and $C[A] = C[B]$, then α is bivariant.

8.3.5.1 Definition-Site Variance

Defn 170 (Definition-Site Variance)**.** *Declaration-site variance* means that we decide about the type variable's variance when we define our Abstract Data Types and Classes.

In Scala and C#, type parameters are always invariant unless they are declared to be covariant or contravariant. The `+` and `-` operators are needed to specify Covariance and Contravariance.

```
class ReadBox[+T] (v : T) { // covariant
  def get() : T = v;
}

class WriteBox[-T] { // contravariant
  def put(v : T) = {};
}

class Box[T] (v : T) { // invariant
  def get() : T = v;
  def put(v : T) = {};
}

class B extends A; // B <: A
class C extends B; // C <: B <: A -> C <: A

// Use the covariance to read from a subtyped box.
def read(rb : ReadBox[B]) {
  val b : B = rb.get;
}
read(new ReadBox(new C()));

// Use the contravariance to read from a supertyped box
def write(wb : WriteBox[B]) {
  wb.put(new B());
}
write(new WriteBox[A]);
```

8.3.5.2 Use-Site Variance

Defn 171 (Use-Site Variance)**.** *Use-site variance* means that we decide about the type variable's variance when we declare an instance of our Abstract Data Types and Classes.

In Java, this is written with either an `extends` or `super` Keyword. Assuming that `B` <: `A`:

```
Box<? extends A> covariantBox = new Box<B>();
Box<? super B> contravariantBox = new Box<A>();
Box<?> bivariantBox = new Box<A>();
Box<A> invariantBox = new Box<A>();

class B extends A {}
class C extends B {}

// Use the covariance to read from a subtyped box.
public static void read(ReadBox<? extends B> rb) {
    B b = rb.get();
}
read(new ReadBox<C>);

// Use the contravariance to read from a supertyped box
public static void write(WriteBox<? super B> wb) {
    wb.put(new B());
```

```
}
write(new WriteBox<A>);
```

Java will now prevent us from calling any method in `covariantBox` that has the type parameter in a contravariant position, so we can only call `covariantBox.get()`. Analogously, for `contravariantBox`, we can only call the `put()` operation. The `invariantBox` prohibits calls to either `put()` or `get()`; but if we had an operation that does not contain the type parameter of the `Box` type anywhere (e.g., a to-string operation of $type() \to String$), then we could still call that method.

Use-Site Variance requires us to write more complex types, but allows us to re-use the same type definition (and the same implementations) for covariant, contravariant, bivariant, and invariant uses.

This is shown in the code below.

```
// Given a regular parametric polymorphic class C (This is also the bivariant case)
class C<T> {
    T get();
    void set(T v);
    boolean isSet();
}

// Here's what it would look like with covariance
class C<? extends T> {
    T get();
    void set(T v); // The set method would be disallowed
    boolean isSet();
}

// Here's what it would look like with contravariance
class C<? super T> {
    T get(); // The get method would be disallowed
    void set(T v);
    boolean isSet();
}

// Here's what it would look like with invariance
class C<?> {
    T get(); // The get method would be disallowed
    void set(T v); // The set method would be disallowed
    boolean isSet(); // Is still allowed because there is no reliance on T
}
```

9 Abstract Data Types and Encapsulation Constructs

9.1 The Concept of Abstraction

Defn 172 (Abstraction)**.** An *abstraction* is a view or representation of an entity that includes only the most significant attributes. In a general sense, abstraction allows one to collect instances of entities into groups in which their common attributes need not be considered, and their unique attributes separate entities which may be from the same group.

There is:

- Process Abstraction, which is discussed elsewhere, with subprograms.
- Data Abstraction.

9.2 Introduction to Data Abstraction

An *Abstract Data Type* is a data structure, in the form of a record, but includes subprograms that manipulate its data. It is an enclosure that only includes the data representations of one specific Data Type, and the subprograms provide operations for that type. This allows unnecessary details of the type to be hidden from units outside the enclosure.

Defn 173 (Object)**.** An instance of an Abstract Data Type is called an *object*.

Abstract Data Types are used to combat program complexity by grouping things together similarly to how we would group them in the real-world as humans.

9.2.1 Floating-Point as an Abstract Data Type

Technically, all Data Types are Abstract Data Types. These are implementations of information hiding, as the programmer does not usually (C/C++ are semi-counter-examples) have direct access to manipulate the bits that make up the number. They only have the operations presented to them by the language designer and implementer.

Overall, this improved program Reliability and portability.

9.2.2 User-Defined Abstract Data Types

Defn 174 (Abstract Data Type)**.** An *abstract data type* has 2 components:

1. The enclosure that "displays" what operations are possible on this Data Type is called the Interface.
2. The code that implements the functionality specified by the interface is called the Implementation.

An abstract data type is a Data Type that satisfies the following conditions:

- The representation of objects of the type is hidden from the program units that use that type, so they only direct operations possible on those objects are those provided in the abstract data type's definition. This improves:
  - Increases Reliability
  - Clients (Units using an abstract data type) cannot manipulate the underlying representation of objects directly
  - Objects can be changed only through the provided operations
  - Reduces the range of code and the number of Variables the programmer must be aware of when reading/writing a program
  - Reduces the likelihood of naming conflicts
- The declarations of the type and the protocols of the operations on objects of the abstract data type, which provide the type's interface, are contained in a single syntactic unit. This benefits the language by:
  - Organizing the program into logical units that can be compiled separately.
- The type's interface does not depend on the underlying representation of the objects, or the implementation of the operations.
  - For example, if a stack is implemented with a linked list, then needs to be changed to an array-like structure, the underlying representation can be changed without affecting any clients that are using the subprograms and Variables.
  - Accessing and modifying data in an abstract data type is done with *getters* and *setters* that allow clients indirect access to the hidden data. There are 3 reasons why this is an improvement:
    1. Read-only access can be provided, by having a getter method, but no corresponding setter method.
    2. Constraints can be included in setters. The setter can enforce the range that a data value can take.
    3. The actual implementation of the data memvers can be changed without affecting the clients, if getters and setters are the only access.
- Also, other program units are allowed to create Variables of the defined abstract data type.

Defn 175 (Interface)**.** An *interface* is the programmer-usable "contract" that can be used for an Abstract Data Type. It ensures that all programmers who use this Abstract Data Type have a common set of operations that behave in a defined manner.

An example of an interface is C and C++'s header files (`*.h` for C, and `*.hpp` for C++).

Remark 175.1. The Interface, usually, does not contain any code. The code that implements the Interface is in the Implementation file. However, in Java, an Abstract Data Type requires that the method have a Subprogram Definition at the same time as its Subprogram Declaration.

Remark 175.2 (Reliance on Specification). An Abstract Data Type's Interface must have a specification to ensure that the Abstract Data Type has the expected operations. This is further discussed in Section 9.3.1.

Defn 176 (Implementation)**.** An *implementation* is the language/project designer's implementation of the "contract" specified by the Interface. For any given problem, there may be a single Interface, but there may be many different possible implementations.

For example, an integer vector, `IntVector` that is created with default values in every element would have a single Interface, but we could *implement* the `IntVector` with several different data structures. We could use:

- An array, for quick random accesses
- A linked list, for efficient memory usage
- A binary tree for relatively efficient lookups and appending of values, ith efficient memory usage.

Remark 176.1 (Reliance on Specification). An Abstract Data Type's Implementation must have a specification to ensure that the Abstract Data Type operates as expected. This is further discussed in Section 9.3.1.

9.3 Design Issues for Abstract Data Types

- The Abstract Data Type name must be externally visible, to allow for object creation
- The Abstract Data Type representation must be hidden.
- There are few built-in operations by a language for Abstract Data Type operations
  - If there are some defined, they are the most basic ones: assignment, comparison.
  - Overloading of subprograms should be allowed
- The form of the container for the interface to the Abstract Data Type.
- Whether Abstract Data Type can be parameterized.
- What access controls are provided, and how are such controls specified?
- Is the specification of the Abstract Data Type physically separate from its implementation?

9.3.1 Specification of Abstract Datatypes

Defn 177 (Specification)**.** We need a *specification* of the Abstract Data Type to ensure that the it is implemented with the correct functionality and that it operates as expected. We also need to specify the actions that are required for the Abstract Data Type in the Interface.

For example, if we have an Interface of an `IntVector`, that supports the following operations:

- create : `int` $\to$ `IntVector`
- length : `IntVector` $\to$ `int`
- append : `(IntVector, int)` $\to$ `()`
- get : `(IntVector, int)` $\to$ `int`

In a code example, we can write our Interface as a Rust Trait.

```
trait IntVector {
        fn create(i32) -> Self;
        fn length(Self) -> i32;
        fn append(Self, i32);  // No return value from an append operation
        fn get(Self, i32) -> i32;
}
```

If we have no specification, we could implement `IntVector` in Python like so;

```
IntVector = bool

def create(len : int) -> IntVector:
    return False

def length(v : IntVector) -> int:
    return 0

def append(v : IntVector, value : int):
    pass  # Return nothing

def get (v : IntVector, offset : int) -> int:
    return 0
```

However, this is a completely useless Implementation. We need a Specification to ensure that the Implementation makes sense.

- If $\ell = \texttt{length}(v)$, and we call $\texttt{append}(v, x)$ once, then afterwards $\texttt{length}(v) = \ell + 1$.
- $\texttt{get}(\texttt{create}(\ell), i) = 0$ if $i \in \{0, 1, \ldots, \ell - 1\}$, otherwise, there is an error.

Using this Specification, we can make an Implementation that behaves the way we expect it to, which will allow the programmer to use the Interface safely, without knowing the actual implementation details.

9.4 Generic Abstract Datatypes

Just like in the Parametric Polymorphism, we can generalize Abstract Data Types. For our `IntVector` example, we might want it to be able to store **any** type of thing. So, we parameterize the Vector type, as shown below.

```
trait Vector<T> {
        fn create() -> Self; // The Vector now ALWAYS starts out empty
        fn length(Self) -> i32;
        fn append(Self, T);  // No return value from an append operation
        fn get(Self, i32) -> T;
}
```

Now, the generic type T is used throughout the entire Abstract Data Type, and any subsequent instance of this Abstract Data Type **must** specify its type too.

Just like in Section 8.2, we can use Typeclasses to constrain our generic type(s). For instance, if we wanted to make a `PriorityQueue`, we would only need to add a single Typeclass to our `Vector` code to change it.

```
trait Vector<T> where T : std::cmp::PartialOrd {
        fn create() -> Self; // The Vector now ALWAYS starts out empty
        fn push(Self, T); // No return type
        fn getTop(Self) -> T;
}
```

Now, the compiler will check if all the Data Types that are fed into the `PriorityQueue` implement the Typeclass `std::cmp::PartialOrd`.

9.5 Language Examples

All of the examples in this section are for the same structure.

9.5.1 Abstract Data Types in Ada

Ada provides an encapsulation construct that can define a single Abstract Data Type, including the ability to hide its representation. These encapsulation constructs are a more general approach than a pure Abstract Data Type. Thus, they fulfill more roles, but can still be used to create an Abstract Data Type..

9.5.1.1 Encapsulation

Defn 178 (Package)**.** Ada's encapsulating constructs are called *package*s. A package can have 2 parts, both of which can *also* be called a package. Not all packages have both of these parts.

1. Package Specification
2. Body Package

A Package Specification and its Body Package can be compiled separately, though the Package Specification must be compiled first. Client code can also be compiled before teh Body Package is compiled or written.

Defn 179 (Package Specification)**.** Ada's *package specification* provides the interface of the encapsulation, and sometimes more.

Defn 180 (Body Package)**.** Ada's *body package* provides the implementation of most, or all, of the entities named in the associated package specification. The Reserved Word, `body`, is used in the header to identify it as a body package.

9.5.1.2 Information Hiding The designer of an Ada Package that defined a Abstract Data Type can choose the make the entire type entirely visible to clients, or to only provide the interface information.[1]

There are 2 methods to hide the representation:

1. Include 2 sections in the Package Specification, one where the entities are visible to the clients are defined, and one that hides its contents.
    - The declaration appears in the visible part of the specification, providing the name of the type and the fact its representation is hidden.

[1] Though, if the Package does not hide its contents, it is not truly an Abstract Data Type.

- The representation is part of the `private` part, which is introduced with that Reserved Word.

2. Hide the representation by defining the Abstract Data Type as a Pointer and provide the pointed-to structure's definition in the Body Package, whose contents are hidden from clients.

These 2 methods are so wildly different because of compilation issues that can arise.

Types that are declared to be private are called Private Type.

Defn 181 (Private Type)**.** Types that are declared to be private are called *private types*. They are declared to be private with the `private` Reserved Word.

An alternative to Private Types are Limited Private Types.

Defn 182 (Limited Private Type)**.** *Limited private types* are an alternative to Private Types. These are more restricted, in that the are described in the private section of a Package Specification. They have *no* built-in operations at all. These are useful when the usual predefined operations of assignment and comparison are not useful.

They are declared with the `limited private` Reserved Word.

These have built-in operations for assignment and comparisons for equality/inequality. Any other operations must be defined in the Package Specification.

9.5.1.3 Example

```
package Stack_Pack is
   -- the visible entities, or public interfaces
private
end Stack_Pack;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
   function Empty(Stk : in Stack_Type) return Boolean is
   begin
   end Empty;

   procedure Push(Stk : in out Stack_Type; Element : in Integer) is
   begin
   end Push;

   procedure Pop(Stk : in out Stack_Type) is
   begin
   end Pop;

   function Top(Stk : in Stack_Type) return Integer is
   begin
   end Top;
end Stack_Pack;
```

9.5.1.4 Evaluation Ada was the first **commercial** language that supported Abstract Data Types[2]. Though, their design may seem complicated and repetitious, it is very powerful, and very reliable.

9.5.2 Abstract Data Types in C++

9.5.2.1 Encapsulation TODO!!

9.5.2.2 Information Hiding TODO!!

9.5.2.3 Constructors and Destructors TODO!!

[2]There were academic languages (CLU) that supported Abstract Data Types before Ada.

9.5.2.4 Example

```
// Interface and implementation of a simple stack

#include <iostream.h>

// Interface Section

// Implementation Section
```

If a header file is used to help modularize the program, the same program can be written this way.

```
// Header file for the Stack class

#include <iostream.h>

class Stack {
private:
        int *stackPtr;
        int maxLen;
        int topSub;
public:
        Stack(); // Constructor
        ~Stack(); // Destructor
        void push(int);
        void pop();
        int top();
        int empty();
}
```

```
// Implementation file for the Stack class

#include <iostream.h>
#include "Class_Example-Stack-Header-CPP.hpp"
using std::cout;

Stack::Stack() { // Define Constructor Instructions
        stackPtr = new int[100];
        maxLen = 99;
        topSub = -1;
}

Stack::~Stack() {
        delete [] stackPtr;
}

void Stack::push(int number) {}

void Stack::pop() {}

int Stack::top() {}

int Stack::empty() {
        return (topSub == -1);
}
```

9.5.2.5 Evaluation C++ supports Abstract Data Types with its class construct. This offers similar power as Ada's Packages. Both provide effective methods for encapsulation and information hiding of Abstract Data Types.

The largest difference is that classes are Data Types themselves, whereas Ada Packages are more general encapsulations. Also, Ada's Packages were designed for more than Data Abstraction.

9.5.3 Abstract Data Types in Objective-C

Objective-C is similar to C++, in that it was designed as an extension to C to support object-oriented programming. However, Objective-C uses the Smalltalk syntax for its method calls.

9.5.3.1 Encapsulation The development of the Abstract Data Type's interface and actual code are separated in Objective-C.

Defn 183 (Interface)**.** In Objective-C, the interface portion of an Abstract Data Type class is defined in a container called an *interface*. It has the following general syntax:

```
@interface class-name: parent-class {
  instance-variable-declarations
}
  method-prototypes
  (+ | -) (return-type) method-name [: (formal-parameters) ];
@end
```

- The `+` indicates the method is a class method (Shared among all instances of this class).
- The `-` indicates the method is an instance method.
- The brackets around the Formal Parameters means the colon and parenthesized formal parameter list is optional.
- If the list is present, then the delimiter is the colon, `:`.

Defn 184 (Implementation)**.** The implementation of the Abstract Data Type class defined with an Interface is a container that implements (obviously) the actual code. This is generally called the *implementation*. These implementations have the following general syntax:

```
@implementation class-name
  method-definitions
@end
```

Defn 185 (Initializers)**.** Objective-C does not have constructors. Instead, it has *initializers*. They only provide initial values, and can be given any name. Since they can have any name, they **must** be called explicitly.

Method calls are done with brackets. If there are Formal Parameters in the called function, the Actual Parameters are passed with a colon, like in the function definition. For example,

```
Adder *myAdder = [[Adder alloc]init]; // Object call to initializer. This one takes no parameters
[myAdder add1: 7]; // Object call to function with parameters, passing 7
```

An Object is creaetd by using the `alloc` Reserved Word.,
All class instances, Objects, are Heap-Dynamic Variable, and are referenced through References.

9.5.3.2 Information Hiding Objective-C uses a setup similar to C++, with `@private` and `@public` directives to specify access levels. However, the default access level is actually a *protected* level of access, rather than *private* as in C++. There is also no way to restrict access to a method.

9.5.3.3 Example

```
// Interface and implementation of a simple stack

#import <Foundation/Foundation.h>

// Interface Section
@interface
Stack: NSObject {
```

```
        int stackArray [100];
        int stackPtr;
        int maxLen;
        int topSub;
}

-(void) push: (int number);
-(void) pop;
-(int) top;
-(int) empty;
@end

// Implementation Section
@implementation Stack
-(Stack*) initWith {
        maxLen = 100;
        topSub = -1;
        stackPtr = stackArray;
        return self;
}

-(void) push: (int) number {}

-(void) pop {}

-(int) top {}

-(int) empty {}

int main(int argc, char* argv[]) {}
@end
```

9.5.3.4 Evaluation

Objective-C's classes are also Data Types. However, the support for Abstract Data Types is adequate. The complete dissimilarity between the C syntax and Smalltalk syntax is also quite confusing.

One deficiency is the lack of ability to restrict access to methods. Another small deficiency is the requirement that constructors be called explicitly.

9.5.4 Abstract Data Types in Java

In Java, the method body **must** appear with its corresponding method header, meaning the an Abstract Data Type is both declared and defined as a single syntactic unit, at the same time. All Objects are allocated from the Heap and accessed through References.

There is also no need to write destructor methods in Java, because of the implicit garbage collection it uses.

9.5.4.1 Example

```
class StackClass {
    private int [] stackRef;
    private int maxLen;
    private int topIndex;

    public StackClass() { // A Constructor
        stackRef = new int [100];
        maxLen = 99;
        topIndex = -1;
    }

    public void push(int number) {}
    public void pop() {}
```

```
    public int top() {}
    public boolean empty() {
        return (topIndex == -1);
    }
}
```

9.5.4.2 Evaluation

Java is different in mostly cosmetic ways to C++. Overall, they offer the same level of support for the design of Abstract Data Types.

9.5.5 Abstract Data Types in C#

All Objects are Heap-Dynamic Variables. Default constructors are defined for all classes, and assign default values to all Abstract Data Type Variables present. The programmer can define as many constructors as desired. Destructors can be defined, but because C# uses implicit garbage collection on nearly everything, this is usually not a big deal.

9.5.5.1 Encapsulation TODO!!

9.5.5.2 Information Hiding TODO!!

9.5.6 Abstract Data Types in Ruby

Ruby provides similar support for Abstract Data Types and classes as Java and C++.

9.5.6.1 Encapsulation

In Ruby, a class definition is a compound statement.

Classes are also dynamic, in that members can be added at any time. The removal of methods is allowed.

9.5.6.2 Information Hiding

Access controls for methods are dynamic, so access violations are only detected during execution. The default method access level is `public`, but it can also be `protected` or `private`.

All data members of an Abstract Data Type are private, and that **cannot** be changed.

9.5.6.3 Example

```ruby
# Defines a stack of maximum length 100, implemented with an array
class StackClass

  # Constructor
  def initialize
    @stackRef = Array.new(100)
    @maxLen = 100
    @topIndex = -1
  end

  # Push method
  def push(number)
  end

  # Pop method
  def pop
  end

  # Top method
  def top
  end

  # Empty method
  def empty
    @topIndex == -1
  end
end # End of StackClass
```

9.5.6.4 Evaluation Everything is an Object in Ruby, and arrays are arrays of references to Objects. Ruby also has dynamic-length arrays be default, so this stack implementation is the most flexible of them all. It can store an arbitrary number of items, and can store any type of item, including items of different Data Types.

10 Expressions

Defn 186 (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 186.1 (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 187 (Operand)**.** An *operand* is a:

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

10.1 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.

10.1.1 Arity

Defn 188 (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 189 (Unary)**.** A *unary* operation requires a single Operand. For example, negation, `-x`

Defn 190 (Binary)**.** A *binary* operation requires 2 Operands. For example, addition, `y + z`

Remark 190.1. These are usually use Infix notation.

Defn 191 (Ternary)**.** A *ternary* operation requires 3 Operands. For example, `w = if x ? y : z`.

Remark 191.1. As far as I know, the only ternary operator is a single-line if statement.

10.1.2 Fixity

Defn 192 (Fixity)**.** *Fixity* is the position of an operation relative to an Operand. There are 3 possible positions:

1. Prefix
2. Infix
3. Suffix

Defn 193 (Prefix)**.** *Prefix* notation has the operators before the Operands that it operates on. For example:

- `-x`
- `+ x y`

Defn 194 (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 195 (Suffix)**.** *Postfix* or *suffix* notation has the operator after the Operands that it operates on. For example:

- `x!`
- `x y +`

10.1.3 Operator Evaluation Order

10.1.3.1 Precedence

Defn 196 (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 196.1. 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 10.1: Precedence of Arithmetic Operators

10.1.3.2 Associativity

Defn 197 (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 197.1. 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 10.2: 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.

10.1.3.3 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

$$(A + B) * C$$

10.1.3.4 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.

10.1.3.5 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[3].

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

10.1.3.6 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;
```

10.1.4 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?

10.1.4.1 Side Effects

Defn 198 (Side Effect)**.** A *side effect* of a function is when the function changes either one of its parameters or a Global Variable.

Defn 199 (Pure)**.** A function can be called *pure* when the function **does not** change its parameters or a Global Variable.

Remark 199.1. These are used in both mathematics and Functional Programming Languages.

Remark 199.2. This is **incredibly** important for multi-core/multi-thread processing, where the order that the data is accessed and potentially written to can change the program's state. This means that 2 threads could be using the same information with 2 different contexts, which is problematic.

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.

10.1.4.2 Referential Transparency and Side Effects

Defn 200 (Referential Transparency)**.** A program has *referential transparency* (is referentially transparent) 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 200.1. A function/program that has Referential Transparency should also be a Pure program/function.

Remark 200.2. 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.

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

Referential Transparency and Side Effects are closely related:

- If an expression e is referentially transparent, it may or may not have side effects, but if it has side effects, then these are not visible outside of e.
- If an expression e is not referentially transparent, it has side effects.
- If an expression e has side effects, it may or may not be be referentially transparent, depending on whether the side effects are visible outside of e.
- If an expression e has no side effects, it is referentially transparent.

10.2 Data Type Conversions

Defn 201 (Data Type Conversion)**.** A *Data Type conversion* is the act of converting one Data Type from one to another. There are 2 effects of this:

1. Narrowing Conversion
2. Widening Conversion

Conversions can be:

- Implicit
- Explicit

Defn 202 (Narrowing Conversion)**.** A *narrowing conversion* converts a Value of one Data Type to another that cannot store approximations of all the Values in the original Data Type. For example, converting a `double` to a `float` in Java.

Remark 202.1 (Safety). Narrowing Conversions are not always safe. The magnitude of the converted value may be changed in during the process of conversion.

For example, in Java, converting the `double 1.3E25` to an `int` will result in a very different number.

Defn 203 (Widening Conversion)**.** A *widening conversion* converts a Value of one Data Type to another that **can store approximations of ALL** the Values in the original Data Type. For example, converting an `int` to a `float` in Java.

Remark 203.1 (Safety). Widening Conversions are almost always safe, because the magnitude of the original value is maintained. However, sometimes they result in reduced accuracy. For example, converting an `int` to a `float` will drastically increase the range of potential values, but may decrease precision.

10.2.1 Coercion in Expressions

Operators that allow operands in an operation to have different Data Types are called Mixed-Mode Operations.

Defn 204 (Mixed-Mode Operation)**.** *Mixed-mode operation*s are operations that allow the operands to have different Data Types. However, there must be an implicit Data Type Conversion present, because computers can only handle single-type operations.

Remark 204.1 (Relation to Overloaded Operations). We will handle operations as if there are distinct operations for every Data Type input. However, if the language has overloaded operators and uses Static Type Checking, then the compiler/interpreter chooses the correct operation. If the operands are of different Data Types and the operation is legal, then the compiler/interpreter must choose one to be coerced, supply the code for the coercion, then perform the operation.

The advantages and disadvantages of this implicit Data Type Conversion are:

- Advantages
  - Much greater flexibility in operators/operands and operations.
- Disadvantages
  - Reliability problems may arise

The real question is whether the compiler/interpreter should handle these issues, or if the programmer should.

In languages that support `byte` and `short`, or their equivalents, and operations are performed on them, they undergo implicit Widening Conversion to `int`s.

10.2.2 Explicit Data Type Conversion

Most languages have facilities to explicitly perform a Data Type Conversion, both widening and narrowing. Sometimes warnings are presented to the programmer.

In C-based languages, an explicit Data Type Conversion is called a *type cast*. The specific syntax of these casts depends on the language. Refer to the language specification to find the exact details for explicit Data Type Conversions.

10.2.3 Errors in Expressions

Any number of errors can occur during an expression's evaluation.

- If the language requires type checking, type errors can occur
- Limitations of computer arithmetic (Limited bit representation)
  - *Overflow*: If the result was too large. In a signed integer system, the first bit is reserved as a sign bit, and if addition flips this first bit, the number becomes a large negative number.
  - *Underflow*: If the result was too small. In a signed integer system, the first bit is reserved as a sign bit, and if this bit is flipped, the number becomes a large positive number.
- Limitations of arithmetic
  - Cannot divide by zero

Overflow, underflow, division by zero, and some other errors are run-time errors, which are sometimes called *exceptions*.

10.3 Relational and Boolean Expressions

10.3.1 Relational Expressions

Defn 205 (Relational Operator)**.** A *relational operator* is an operator that compares the values of its two Operands. These are ***ALWAYS*** Binary operators.

Defn 206 (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.`[4] or `<>`
   - ML and F# use `<>`
3. `>=`, Greater than or Equal to
4. `<=`, Less than or Equal to
5. `>`, Greater than
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

10.3.2 Boolean Expressions

Defn 207 (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.

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`.

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

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

Table 10.3: Precedence Table with All Expression Types

10.4 Short-Circuit Evaluation

Defn 208 (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 10.4 to 10.5.

	0	1
0	0	0
1	0	1

Table 10.4: `AND` Truth Table

	0	1
0	0	1
1	1	1

Table 10.5: `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.

11 Assignment Statements

Assignments are one of the central constructs in Imperative Programming Languages. Assignments allow progarmmers to dynamically change the bindings of Bindings to Variables.

11.1 Simple Assignments

Nearly all languages use the `=` symbol as the assignment operator. In these languages, the use of `==` as the equivalence Relational Operator is common.

2 notable exceptions to this assignment symbol are:

1. ALGOL 60, with `:=`
2. Ada, with `:=`

They chose this set of symbols so as to be able to use the `=` as the equivalence Relational Operator. The destination of the Value has varied widely in many languages. Additionally, the appearance of an assignment as a stand-alone statement is also dependent on language.

11.2 Conditional Targets

Some languages allow for assignments to take place in conditional Blocks. These assignments bind the Value to a *conditional target*.

In Perl for example,

```
($flag ? $count1 : $count2) = 0;
```

which is equivalent to

```
if ($flag) {
    $count1 = 0;
} else {
    $count2 = 0;
}
```

11.3 Compound Assignment Operators

Defn 209 (Compound Assignment Operator)**.** A *compound assignment operator* is a shorthand method of specifying a commonly needed form of assignment. The overwritting of a Variable's Value with some new value that is dependent on the Variable's previous Value.

For example:

```
sum += value;
# This is equivalent to
sum = sum + value;
```

Remark 209.1 (Supported Operators). If a language supports an addition Compound Assignment Operator, then it is likely that is supports it for all the other binary arithmetic operations. They are likely to be denoted `-=`, `*=`, `/=`, and `%=` for subtraction, multiplication, division, and modulo, respectively.

11.4 Unary Assignment Operators

Many C-based langagues support 2 special unary arithmetic operators, that are abbreviated assignment statements. These are:

1. Increment by 1, usually written `++`
   - This is equivalent to `i = i + 1`, where `i` is a defined, in-scope, and alive integer Variable
2. Decrement by 1, usually written `--`
   - This is equivalent to `i = i - 1`, where `i` is a defined, in-scope, and alive integer Variable

Both of these can be Prefix and Suffix operators. The different fixities mean different things.

- Prefix, `sum = ++count`
  - First increment `count`
  - Then assign the result to `sum`
- Suffix, `sum = count++`
  - First assign the value of `count` to `sum`
  - Increment the value stored in `count` by 1
  - Note, the value in `sum` will be the value that was in `count` **BEFORE** the increment occurred.

Remark (Multiple Unary Operators Associativity). When there are multiple unary operators on a single Variable at a single time, they are **RIHT-TO-LEFT ASSOCIATIVE**. Thus,

```
- count ++
# is handled like
- (count++)
# rather than
(- count) ++
```

11.5 Assignment as an Expression

In C-based langauges, the assignment statement also produces a result that is the same as the value assigned. This means that the assignment statement can be treated as an Expression, and an Operand as well, and used as such. For example, this is a valid while-loop in C.

```
while ((ch = getchar()) != EOF) { ... }
```

This statement gets a character from the standard input, which is usually the keyboard. If the letter input to the program is **not** the End-Of-File character, then the statement(s) in the while-loop's body execute. The character was **ALSO** assigned to the character Variable, `ch`.

11.5.1 Side Effects

Allowing this action in programs means that Expressions can be more difficult to read and udnerstand. Instead of thinking of these kinds of Expressions as Expressions, they must be though of as a list of instructions with a strange order of execution. For example,

```
a = b + (c = d / b) - 1;
// This gets translated to
c = d / b;
a = b + c - 1;
```

In C programs, this is a big cause of problems. Because both `if (x = y)` and `if (x == y)` are valid, but the second is performing a Relational Operator, while the first is performing an assignment. Java and C# have avoided this problem by **only** allowing boolean expressions in their `if` statements.

12 Statement-Level Control Structures

Defn 210 (Control Statement)**.** *Control statements* provide the ability to:

- Select alternative control flow paths of statement execution
- Cause repeated execution of statements or sequences of statements

Defn 211 (Control Structure)**.** A *control structure* is a Control Statement and the collection of statements whose execution it controls.

12.1 Selection Statements

Defn 212 (Selection Statement)**.** A *selection statement* provides a means of choosing between 2 or more execution paths in a program.

There are 2 general categories of these:

1. 2-Way Selection
2. N-Way Selection

12.1.1 2-Way Selection

There are several ways to syntactically design a 2-Way selection statement, but they all follow the same basic steps.

```
if control_expression
   then clause
   else clause
```

12.1.1.1 Design Issues

- What is the form and type of the expression that controls the selection?
- How are the `then` and `else` clauses specified?
- how should the meaning of nested selectors be specified?

12.1.1.2 The Control Expression

The control expressions are specified in parentheses if a `then` Reserved Word/Keyword is not used to introduce the `then`-clause.

```
if(x==y) // The control expression is inside the parentheses, it is x==y
..
```

If there is a `then` Reserved Word/Keyword, then the parentheses are optional, but are usually left out.

```
if x==y then # The control expression has no parentheses, it is x==y
...
```

In languages that did not have a Boolean data type, arithmetic expressions were used as control expressions. However, many modern languages allow either arithmetic and Boolean expressions, or **ONLY** Boolean expressions.

12.1.1.3 Clause Form

In many modern languages, the `then` and `else` clauses can be either single statements or compound statements. Many languages use the curly braces { and } to delimit compound statements. However, some use a set Reserved Words (for example `end if` in Ada) to delimit these statement sequences. Python is weird and uses indentation.

The variety of methods of writing the `then` and `else` clauses have implications for nesting these selectors.

12.1.1.4 Nesting Selectors

The problem here is what is called the "Dangling Else" problem. For example, the following code snippet is valid in a C-based language.

```
if(sum == 0)
  if(count == 0)
    result = 0;
  else
    result = 1;
```

However, to which `if` does the `else` belong to? There is no way to use syntax to determine which `if` to match the `else` to, rather the static semantics handle this. Generally, the static semantics match the current `else` with the nearest previous unpaired `then` clause. This is an issue that can be solved with some braces.

To force the `else` clause above to match with the `if(sum == 0)` clause,

```
if(sum == 0) {
  if(count == 0) {
    result = 0;
  }
} else {
  result = 1;
}
```

However, if the `else` clause was meant to go with the inner `if-then` statement,

```
if(sum == 0) {
  if(count == 0) {
    result = 0;
  } else {
    result = 1;
  }
}
```

One way to completely avoid this type of issue is to use `then` and `end` Reserved Words/Keywords. This also leads to more regularity in the writing of the language. Using the code snippets from earlier:

```
if sum == 0 then
  if count == 0 then
    result = 0
```

```
  end
else
  result = 1
end
```

This matches the first C-like example.

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

This matches the second C-like example.

Remark. Some languages completely avoid this issue by enforcing that there always be an `else` clause, even if it is to be empty.

12.1.1.5 Selector Expressions

In ML, F#, and LISP, the selector is an expression that can result in a value. In F# for example,

```
let y =  // Assign either x to y if x > 0, otherwise, assign 2 * x to y
    if x > 0 then
        x
    else
        2 * x;;
```

Note: This might be improperly formatted. This was done to increase regularity between all `if-then-else` statements.

12.1.2 N-Way Selection

The N-Way selection statement allows selection from one of any number of statements/statement groups. Although these can be constructed from 2-Way selectors and `goto`s, it is more readable with a dedicated statement.

12.1.2.1 Design Issues

- What is the form and type of the expression that controls the selections?
- How are the selectable segments specified?
- Is the execution flow through the structure restricted to just a single selectable statement?
- How are the case values specified?
- How should unrepresented selector expression values be handled, if at all?

12.1.2.2 Examples of Multiple Selectors

C and its derivative langauges use the `switch` statement. Its general form is

```
switch (expression) {
  case constant_expression1: statement1;
  ...
  case constant_expressionN: statementN;
  [default: statementN+1]
}
```

The control and constant expressions are some discrete type, including:

- Integers
- Characters
- Enumeration Types

The selectable statements can be:

- Statement sequences
- Compound Statements
- Blocks

The optional `default` segment is for options that do not fit with previously defined `case`s. Also, there is **no** implicit branching to the end of the whole statement at the end of each `case`. To branch during a `case`, a `break` is used. `break` is a restricted `goto`.

Ruby has another syntax for this same construct, called a case expression. Additionally, because it is an expression, we can also use it to assign values conditionally.

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

12.2 Iterative Statements

Defn 213 (Iterative Statement)**.** An *iterative statement* is one that causes a statement or collection of statements to be executed zero ore more times. These are frequently called *loops*.

There are 2 questions that need to be answered to design iterative statements.

1. How is the iteration controlled?
2. Where should the control mechanism appear in the loop statement, i.e. when is it executed in relation to the code in the statement body?

Remark 213.1 (Iteration Statement). An *iteration statement* is the combination of the Body and the control mechanism.

Counter-Controlled Loops and Counter-Controlled Loops, and a combination of the 2, are the main ways to control an Iterative Statement. The 2 main choices of when to execute the control mechanism of the loop are:

1. At the beginning of the loop, before the body. Pretest
2. At the end of the loop, after the body. Posttest

Defn 214 (Body)**.** The *body* of an Iterative Statement is the collection of statements whose execution is controlled by the iteration station statement.

Defn 215 (Pretest)**.** A *pretest* Iterative Statement means the test for loop completion, the control mechanism, is evaluated before the loop Body executes.

Defn 216 (Posttest)**.** A *posttest* Iterative Statement means the test for loop completion, the control mechanism, is evaluated after the loop Body executes.

12.2.1 Counter-Controlled Loops

Defn 217 (Counter Iteration Statement)**.** A *counter iteration statement* has:

- A *loop variable*, where the count value is maintained
- The *loop parameters*. These include
  - A means of specifying the *initial* and *terminal* values of the loop variable
  - The *step size*, the difference between sequential loop variable values

Logically Controlled Loops are more **general** than Counter-Controlled Loops, they are not necessarily more commonly used. The greater complexity of Counter-Controlled Loops means that their design is more demanding.

Remark (Machine Instructions). Sometimes there are machine-level instructions for performing Counter-Controlled Loops. However, these are rare, because a language can outlive a processor's ISA.

12.2.1.1 Design Issues

- What are the Data Type, and Scope of the loop variable?
- Should it be legal for the loop variable or loop parameters to be changed in the loop, and if so, does it affect the loop control?
- Should the loop parameters be evaluated only once, or once for each iteration?

12.2.1.2 The `for` Statement of Ada

The Ada `for` statement can be generalized to

```
for variable in [reverse] discrete_range loop
    -- Body statements
end loop;
```

- The `discrete_range` is a Subrange Type of an integer or Enumeration Type, such as `1..10` or `Monday..Friday`.
- The `reverse` Reserved Word indicates the values of the `discrete_range` are assigned in reverse order.
- The `variable` is only in scope of the loop
  - It is implicitly declared at the beginning of the `for` statement
  - and implicitly undeclared after loop termination
  - It also shadows/masks variables with the same name as it
- The `body` cannot assign the loop variable a new value

The operational semantics are given below.

```
  [define for_var (its type is that of the discrete range)]
  [evaluate discrete range]
loop:
  if [there are no elements left in the discrete range] goto out
  for_var = [next element of discrete range]
  [loop body]
  goto loop
out:
  [undefine for_var]
```

12.2.1.3 The `for` Statement of the C-Based Languages

The C `for` statement can be generalized to

```
for (expression_1; expression_2; expression_3) {
    // Body statements
}
```

- `expression_1` is usually an assignment statement to create the loop variable, which can produce results
  - This is only evaluated once, before the loop begins
  - In early versions of C, these could **not** be definitions (`int count = 0;`)
- `expression_2` is the loop control, it makes comparisons against the loop variable
  - This is evaluated before every iteration of the loop body
- `expression_3` manipulates the loop variable
  - This is evaluated after every iteration of the loop body
- All 3 of these expressions are optional.
  - Having no `expression_2` means the comparison is always true, which may result in an infinite loop
  - If `expression_1` is absent, no loop variable is initialized, but that doesn't stop other Variables declared elsewhere from being part of the control.

Because C expressions can be used as statements, expression evaluations are shown as statements in this operational semantics.

```
  expression_1
loop:
  if expression_2 = 0 goto out
  [loop body]
  expression_3
  goto loop
out:
  ...
```

C's design choices for its `for` loop are as follows:

- No explicit loop variables or loop parameters
- All involved variables can be changed in the loop body
- Expressions are evaluated in the order shown before.
  - There can even be multiple expressions present in each of the `expressions`.
- You can branch into a C `for` loop body

12.2.1.4 The `for` Statement of Python

The Python `for` statement can be generalized to

```
for loop_variable in object:
    # Body statements
[else:
    # Else clause]
```

- The `loop_variable` is assigned the one of the values in the object, which is often a range, for each iteration.
- The `else` clause is executed if the loop terminates **normally**

12.2.2 Logically Controlled Loops

Collections of statements must be executed repeatedly, but should be controlled by a Boolean expression rather than a counter. Logically controlled loops are convenient for this. They are also more general than Counter-Controlled Loops.

12.2.2.1 Design Issues

- Should the control be Pretest or Posttest?
- Should the logically controlled loop be a special form of a counting loop or a separate statement?

12.2.2.2 Examples C-based programming languages have both Pretest and Posttest Logically Controlled Loops. These loops have these respective forms:

```
while (control_expression) {
    // Loop body
}
```

```
do
    // Loop body
while (control_expression);
```

The Pretest (`while`) executes as long as the `control_expression` evaluates to true. The Posttest (`do-while`) executes as long as the `control_expression` evaluates to true. However, the `do-while` loop will cause the loop body to be executed at least once.

The operational semantics are given below, with the `while` coming first, then the `do-while`.

```
loop:
  if control-expression is falst go to outlive
  [loop body]
  goto loop
out:
  ...

loop:
  [loop body]
  if control_expression is true goto loop
```

In C, it is possible to branch into these statement's bodies. Java's `while` and `do-while` statements are similar, but they require the `control_expression` be `boolean` type.

12.2.3 Iteration Based on Data Structures

A general data-based iteration statement uses a user-defined data structure and user-defined iteration to go through the elements in the structure. An example of this is in Python, shown below

```
for count in range(0, 9, 2)
    # Body
```

This would set the value of `count` to `0, 2, 4, 6, 8` before each iteration.

The user-defined iterator function must be history sensitive. It should also terminate when it fails to find more elements.

User-defined Iteration Statements are more commonly used and more important in object-oriented programming, because of the heavy use of abstract data types for data structures. In Java, this is done by having a class implement the `Iterable` interface. It is called the `foreach` loop, and is written as

```
for (String myElement : myList) {
    ...
}
```

13 Subprograms

This is a way to perform Process Abstraction. This generally improves Readability and Reliability.

13.1 General Characteristics

- Each subprogram has a single entry point
- The calling program unit is suspended during the execution of the called subprogram, which implies there is only one subprogram in execution at any given time
- Control always returns to the caller whenthe subprogram terminates

Alternatives to these generalizations result in coroutines and concurrent units.

13.2 Basic Definitions

Defn 218 (Subprogram Definition)**.** A *subprogram definition* describes the interface to and the actions of the subprogram abstraction

Defn 219 (Subprogram Call)**.** A *subprogram call* is the explicit request that a specific subprogram be executed.

Remark 219.1 (Call)*.* This is generally shortened to just a *call*.

Defn 220 (Active)**.** A subprogram is said to be *active* if it has been called, but not yet completed its execution.

Defn 221 (Subprogram Header)**.** A *subprogram header* is the first part of a Subprogram Definition. This serves several purposes:

1. Specifies that the following syntactic unit is a Subprogram Definition of some kind.
2. Provides a name for the subprogram, if it's not an anonymous subprogram.
3. Optionally specify a list of parameters.

Defn 222 (Subprogram Body)**.** The *subprogram body* defines the actions that the subprogram takes when there is a Subprogram Call. The body may be delimited with curly-braces, { and }. It may be whitespace delimited, Python. It may also have an `end` statement that ends the execution of that block.

Python is unique in that its Subprogram Definitions can be executed in control-statement blocks. For example,

```
if conditional_expression :
    def func(...):
        ...
else:
    def func(...):
        ...
```

This means that there are 2 possible Subprogram Definitions possible during runtime, and which one is currently valid depends on the result of the `conditional_expression`.

Defn 223 (Parameter Profile)**.** The *parameter profile* of a subprogram contains the number, order, and types of its Formal Parameters.

Defn 224 (Protocol)**.** The *protocol* of a subprogram is its Parameter Profile, and if its a function, its return type.

Defn 225 (Subprogram Declaration)**.** A *subprogram declaration* is the act of providing type and name information, but not giving any Subprogram Body. This is needed in languages that do not allow forward references to subprograms.

Remark 225.1 (Prototype)*.* In C/C++, if a subprogram needs to be declared, it is called a *prototype*. These are generally specified in *header* files, with a file extension of `.h`.

13.3 Parameters

Subprograms typically want access to nonlocal data to perform their computations. There are 2 ways to gain access to this nonlocal data:

1. Direct access to nonlocal Variables (Global Variables)
2. Parameter passing

Data that is passed to the subprogram as a parameter is acccessed through names local to the subprogram. Parameter passing is more flexible, because if direct access is used, new storage needs to be allocated for computation results. Direct access also leads to issues with Variables being visible to places they shouldn't be. Pure Functional Programming Languages avoid this by having all their data being immutable.

Defn 226 (Formal Parameter)**.** A *formal parameter* are the parameters present in the Subprogram Header. These are sometimes thought of as "dummy variables" because they aren't normal Variables. They are only bound to storage when the subprogram is called, and that storage is often through some other program Variables.

Remark 226.1 (Parameter)*.* Sometimes Formal Parameters are just called *parameter*s, usually when Actual Parameters are called Arguments.

Defn 227 (Actual Parameter)**.** An *actual parameter* is the parameter that is bound to the Formal Parameter of a subprogram.

Remark 227.1 (Argument)*.* Sometimes Actual Parameters are called *argument*s, usually when Formal Parameters are called Parameters.

The binding of Actual Parameters to Formal Parameters is usually done by position. So, the first Actual Parameter is bound to the first Formal Parameter. However, this is only a good method when the number of parameters is small.

When the Formal Parameter list gets long, it is hard to get all of the Actual Parameters in the right order. One solution is to use Keyword Parameters.

Defn 228 (Keyword Parameter)**.** *Keyword parameter*s have the name of the Formal Parameter usable by the Actual Parameter to bind the Value.

For example, if `sumer` has the Formal Parameters `length`, `list`, and `sum`:

```
1 sumer(length = my_length, list = my_array, sum = my_sum)
```

Advantages and Disadvantages of keyword parameters:

- Advantages
  - No need to remember Formal Parameter order.
- Disadvantages
  - Need to remember the name of the Formal Parameters.

Remark 228.1 (End of Actual Parameter)*.* In an Actual Parameter list, all parameters after a Keyword Parameter **must** be keyworded, because the list may not be well-formed enough for parameters to line up by position.

Languages that support default values on Formal Parameters handle them differently. In Python, regular Formal Parameters and ones with default values can be in any order. However, in C++, which does not support Keyword Parameters, Formal Parameters with default values must be at the end of the Subprogram Header. This is illustrated in the next 2 code blocks, which are in Python and C++, respectively.

```
def compute_pay(income, exemptions = 1, tax_rate)
```

```
float compute_pay(float income, float tax_rate, int exemptions = 1)
```

Other languages have more varied and interesting ways to pass Actual Parameters to subprograms. Look at the language specification for more details.

13.4 Local Referencing Environments

Issues related to Variables defined within subprograms.

13.4.1 Local Variables

The definition of Local Variables is given in Definition 68. These can be either Static Variable or Stack-Dynamic Variable.

If Local Variables are Stack-Dynamic Variable, they are bound to storage when the subprogram begins and unbound when that execution terminates. The advantages and disadvantages of Stack-Dynamic Variable are:

- Advantages
  - Allows for recursive subprograms
  - Inactive subprograms can share Memory with the active subprogram
- Disadvantages
  - The cost of the time requried to allocate, initialize, and deallocate these variables
  - The indirect Memory accesses to the data
  - When all Variables are Stack-Dynamic Variable, subprograms cannot be history sensitive.

However, the advantages and disadvantages of Static Variable are:

- Advantages
  - No runtime overhead to allocate/deallocate the storage
  - Direct Memory access (Absolute addressing)
- Disadvantages
  - Inability to support recursion
  - Cannot share Memory with inactive subprograms.

Most contemporary programming languages make their Local Variables Stack-Dynamic Variable by defautlt. However, this can usually be overridden with a `static` keyword.

13.4.2 Nested Subprograms

The idea was to create a hierarchy of logic and Scopes. The motivation was that if a subprogram is only used within one other subprogram, why not place it there, and hide it from the rest of the program?

Static Scoping is usually used in languages that allow nested subprograms. Language support of this featuer depends **heavily** on the language. You will have to chek the language specification to find out if the programming language supports them.

13.5 Parameter-Passing Methods

How are Actual Parameters passed to the subprograms?

13.5.1 Semantic Models of Parameter Passing

Formal Parameters are characterized by 1 of 3 distinct Semantics models:

1. The Formal Parameters receive data from the corresponding Actual Parameter. This is called **In Mode**.
2. The Formal Parameters can transmit the computed data back to the Actual Parameter. This is called **Out Mode**.
3. They can do both 1 and 2. This is called **In/Out Mode**.

Defn 229 (In Mode)**.** Formal Parameters can receive data from the corresponding Actual Parameter. This is called *in mode*.

This is generally used when passing Actual Parameters to a subprogram.

Defn 230 (Out Mode)**.** Formal Parameters transmit the computed data back to the corresponding Actual Parameters. This is called *out mode*.

This is similar to a return value, but acts *only* on the actual parameters.

Defn 231 (In/Out Mode)**.** The Formal Parameters can both transmit and receive data from the corresponding Actual Parameters. This is called *in/out mode*.

This is generally used when a Subprogram Call takes in Actual Parameters and returns values in those same parameters.

There are 2 conceptual models of how data transfers take place in parameter transmission:

1. An actual value is copied (to the caller, to the callee, or both).
2. Or an access path is transmitted. This is usually a pointer/reference.

13.5.2 Implementation Models of Parameter Passing

A great variety of implementations for parameter passing have been put together. Here, we list some of them, and discuss their respective advantages and disadvantages.

Remark (Call-by-...). All of these models can have the "Pass-by" replaced with "Call-by", which means the same thing.

13.5.2.1 Pass-by-Value

Defn 232 (Pass-by-Value)**.** When a parameter is *passed-by-value*, the value of the Actual Parameter is used to initialize the corresponding Formal Parameter, which then acts as a Local Variable in the subprogram. This implements In Mode Semantics.

Passing a parameter by value is typically done by copying the Actual Parameter's Value for the Formal Parameter. This means we don't have to make the Memory cell read-only, because making cells read-only can be difficult.

The advantages and disadvantages are:

- Advantages
  - Fast to copy scalars in both linkage cost and access time
- Disadvantages
  - Additional Memory is required for the Formal Parameter's new value.
  - The Actual Parameter must be copied to the storage area for the corresponding Formal Parameter
  - Difficult to implement by transmitting access paths
  - This copying can be expensive if the Actual Parameter is large, like an array with many elements.

13.5.2.2 Pass-by-Result

Defn 233 (Pass-by-Result)**.** *Pass-by-result* is an implementation for Out Mode parameters. When a parameter is passed-by-result, no Value is transmitted to the subprogram. The corresponding Formal Parameter acts as a Local Variable, but before control is transferred back to the caller, the Formal Parameter's result is transmitted back to the caller's Actual Parameter.

The advantages and disadvantages of pass-by-result are fairly similar to Pass-by-Value, but with some extra disadvantages.

- Advantages
  - Fast to copy scalars
- Disadvantages
  - If the values are returned by copying the value into the Actual Parameter, extra storage and copy operations are required.
  - Difficult to transmit an access path.
    - Need to ensure the initial value of the Actual Parameter is **not** used in the subprogram.
  - There can be an Actual Parameter collision.

```
1 void Fixer(out int x, out int y) {
2   x = 17;
3   y = 35;
4 }
5 ...
6 f.Fixer(out a, out a); // What gets assigned first? Is a=17 or a=35? Who knows?
```

  - A similar issue arises when the implementor can choose between 2 different times to evaluate the addresses of the Actual Parameters. This is illustrated below.

```
void DoIt(out int x, int index) {
  x = 17;
  index = 42;
}
...
sub = 21;
f.doIt(out list[sub], out sub); // What gets assigned to list[sub], because sub is unknown: sub=21? s
```

13.5.2.3 Pass-by-Value-Result

Defn 234 (Pass-by-Value-Result)**.** *Pass-by-value-result* is an implementation of the In/Out Mode model in which actual values are copied. It is a combination of Pass-by-Value and Pass-by-Result. The Value of the Actual Parameter is used to initialize the Formal Parameter, which then acts as a Local Variable. At subprogram termination, the value of the Formal Parameter's Local Variable is copied back to the Formal Parameter, then copied back to the Actual Parameter.

Remark 234.1 (Pass-by-Copy). Pass-by-Value-Result is sometimes called *pass-by-copy*, because the Actual Parameter is copied to the Formal Parameter at the subprogram's start, and then copied back at the subprogram's termination.

The advantages and disadvantages of pass-by-value-result are shared with both Pass-by-Value and Pass-by-Result.

13.5.2.4 Pass-by-Reference

Defn 235 (Pass-by-Reference)**.** *Pass-by-reference* is a second implementation model for In/Out Mode. In this case, rather than copying the actual data Values back and forth, pass-by-reference transmits an access path, usually an address/pointer/reference, to the called subprogram. This allows the subprogram to access the **SAME** value as the calling program.

The advantages and disadvantages of this implementation are:

- Advantages
  - The passing process is efficient, in terms of time and space.
    - No duplicate space is required
    - There is also no copying required
- Disadvantages
  - Access to the Formal Parameters will be slower than Pass-by-Value, because of the indirect addressing required.
  - There may be inadvertent and erroneous changes to the Actual Parameter's Value.
  - Aliasing can occur.
    - This harms Readability and Reliability.

```
void func(int &first, int &second);
func(total, total); // When using first or second, they both point to the same ``total'' variable
fun(list[i], list[j]); // Assuming i==j is true, list[i] and list[j] both point to the same value
```

  - Program verification is more difficult.

13.5.2.5 Pass-by-Name

Defn 236 (Pass-by-Name)**.** *Pass-by-name* is an In/Out Mode parameter transmission method that does not correspond to a single implementation model. When Actual Parameter are passed by name, every occurrence of the actual parameter is textually substituted for the corresponding Formal Parameter. A pass-by-name Formal Parameter is bound to an access method at the time of the program call, but the actual binding to a Value or an Address is delayed until the Formal Parameter is assigned or referenced.

Implementing this requires a subprogram be passed to the called subprogram to evaluate the Address or Value of the Formal Parameter.

The advantages and disadvantages of this are:

- Advantages
  - Formal Parameters are lazily evaluated, so they will only be evaluated once they are needed.
  - However, the Formal Parameter must be evaluated **EACH TIME**, making this a very inefficient method.
  - Can add Expressivity.
- Disadvantages

- Pass-by-name parameters are difficult to implement
- Pass-by-name parameters are inefficient
- Add significant complexity to the program
  - Reduced Readability
  - Reduced Reliability

Remark 236.1 (Languages using Pass-by-Name). There are no widely-used high-level languages that use Pass-by-Name. However, it is used at compile time by macros is assembly languages and for generic parameters of generic subprograms.

13.5.2.6 Pass-by-Need

Defn 237 (Pass-by-Need)**.** This is a fairly niche way to pass Actual Parameters around to a subprogram's Formal Parameters. It is only used by the Haskell language.

This is **VERY** similar to the Pass-by-Name implementation, but instead of evaluating the Actual Parameter each time the Formal Parameter appears, the Actual Parameter is evaluated *AT MOST ONCE*. Then, every subsequent occurrence of the Actual Parameter has it value substituted by Referential Transparency, or if its a function call, the result of the Pure function.

The advantages and disadvantages of pass-by-need are:

- Advantages
  - More time-efficient than Pass-by-Name
- Disadvantages
  - Less flexity in the presence of updating Variables.
    - Haskell avoids this problem because its "variables" are immutable, which makes this a moot point.

13.6 Parameters That Are Subprograms

Being able to use Subprograms as Actual Parameters in a program can help simplify the programming of certain problems. There are 2 major complications of this desire:

1. The Type Checking of the parameters of the activations of the subprogram that was passed as a parameter.
2. In languages that allow Nested Subprograms, there is a question of what Referencing Environment should be used. There are 3 possible solutions:
   (a) *Shallow Binding*
   (b) *Deep Binding*
   (c) *Ad-Hoc Binding*

Defn 238 (Shallow Binding)**.** In *shallow binding*, the Referencing Environment of the call statement that enacts the subprogram is used.

The Referencing Environment where the call statement to the passed subprogram occurs is Referencing Environment when the subprogram is executed.

Remark 238.1 (Shallow Binding Program Output). In the program below, the output of `sub2` is `4`, because shallow binding binds the Referencing Environment of `sub2` to `sub4`'s Referencing Environment, where `sub2` was executed.

Defn 239 (Deep Binding)**.** In *deep binding*, the Referencing Environment of the definition of the passed subprogram is used.

The Referencing Environment of the definition of the subprogram is used as the Referencing Environment.

Remark 239.1 (Deep Binding Program Output). In the program below, the output of `sub2` is `1`, because deep binding binds the Referencing Environment of `sub2` to `sub1`'s Referencing Environment, where `sub2` was defined.

Defn 240 (Ad-Hoc Binding)**.** In *ad-hod binding*, the Referencing Environment of the call statement that *passed* the program as an Actual Parameter is used.

The Referencing Environment where the call statement passes the subprogram is used as the Referencing Environment for the subprogram.

Remark 240.1 (Ad-Hoc Binding Program Output). In the program below, the output of `sub2` is `3`, because ad-hoc binding binds the Referencing Environment of `sub2` to `sub3`'s Referencing Environment, where `sub2` was originally passed as a subprogram.

This code block is written with JavaScript's syntax, but depending on the binding used, the output of `sub2` will be different.

```
function sub1() {
  var x;
  function sub2() {
    alert(x); // Creates a box with the value of x
  };
  function sub3() {
    var x;
    x = 3;
    sub4(sub2);
  };
  function sub4(subx) {
    var x;
    x = 4;
    subx();
  };
  x = 1;
  sub3();
};
```

13.7 Closures

Defn 241 (Closure)**.** A *closure* is a subprogram and the Referencing Environment where it was defined. The Referencing Environment is needed if the subprogram can be called from any arbitrary place in the program.

Remark 241.1 (Static Scoped, No Nested Subprograms). If a statically-scoped programming language does not support nested subprograms, then it usually doesn't support Closures either. All of the Variables in the Referencing Environment of a subprogram are accessible, regardless of the place in the program where the subprogram is called.

When subprograms can be nested, the subprogram can use its Local Variables, the Global Variables, and any Variables declared in parent subprograms. This isn't an issue when the nested subprogram can only be called where the enclosing scopes are active and Visible.

However, if the nested subprogram can be called from elsewhere. This can happen if the subprogram is passed as a parameter or assigned to a variable. This also means that the subprogram could be called after its parent subprograms have terminated, meaning some of the Variables defined there are no longer available. To prevent this, we have to have special Variables which are said to have *Unlimited Extent*.

Defn 242 (Unlimited Extent)**.** A Variable whose lifetime is that of the whole program, and are required by nested subprograms said to have *unlimited extent*.

These variables are usually heap-dynamic, rather than Stack-Dynamic Variables.

The code snippet below is an example of a Closure and its Variables having Unlimited Extent.

```
static Func<int, int> makeAdder(int x) {
  return delegate(int y) { return x + y;};
}
...
Func<int, int> Add10 = makeAdder(10); // Makes a function that adds 10 to a passed integer parameter
Func<int, int> Add5 = makeAdder(5); // Makes a function that adds 5 to a passed integer parameter
Console.WriteLine("Add 10 to 20: {0}", Add10(20)); // Prints "Add 10 to 20: 30"
Console.WriteLine("{0}", Add5(20)); // Prints 25
```

14 Implementing Subprograms

14.1 General Semantics of Calls and Returns

Defn 243 (Subprogram Linkage)**.** The `call` and `return` operations are together called *subprogram linkage*.

There are several steps that must occur for a subprogram's `call` to work.

1. Include the implementation of that language's parameter-passing method.

2. If local variables are not static, there must be space allocation for the Local Variables declared in the subprogram, and bind them to storage.
3. Must save the execution status of the CPU, everything required to jump back to the point where the `all` occurs. This includes:
   - CPU status bits
   - The Environment Pointer (Dynamic Link)
   - Register values
4. Arrange the transfer of control to the code of the subprogram, and ensure control can be returned to the proper place when execution is done.
5. If the language supports nested subprograms, there needs to be a mechanism to provide access to the parent subprogram's variables. (Static Link)

The required actions for a subprogram to return its execution to the parent subprogram are:

1. If the parameters are Out Mode or In/Out Mode, the local values of the associated Formal Parameters must be copied to the Actual Parameters.
2. All storage used for Local Variables must be deallocated.
3. Restore the execution status of the calling parent subprogram.
4. Return control to the calling parent subprogram.

14.2 Implementing "Simple" Subprograms

In this case, "simple" means subprograms that cannot be nested and all local variables are static.

There must be storage for:

- Status information for the caller program.
- Parameters
- The return Memory address
- Return values for functions
- Temporaries used by the code of the subprogram(s)

Some of the semantic actions in the next 2 sections (Sections 14.2.1 to 14.2.2) can be occur at 2 different times during Subprogram Linkage. These are called the *prologue* and *epilogue* of the Subprogram Linkage.

14.2.1 Semantics of the Subprogram `Call`

The following steps must be followed:

1. Save the execution status of the current program unit.
2. Compute and pass the parameters.
3. Pass the return address to be called at the end of subprogram execution.
4. Transfer control to the called subprogram.

The last 3 actions must be done by the caller program. The first action could be done by the caller program or the called subprogram.

14.2.2 Semantics of the Subprogram `Return`

These are the steps that must be followed to `return` from a subprogram:

1. If parameters are Pass-by-Value-Result or In/Out Mode, the current values of those Formal Parameters are moved/made available to the corresponding Actual Parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. The execution status of the caller program is restored.
4. Control is transferred back to the caller program.

The first, third, and fourth steps could be handled by the called subprogram.

14.2.3 Activation Records for "Simple" Subprograms

Defn 244 (Activation Record)**.** An *activation record* for a "simple" subprogram is the noncode portion, because the data it describes is only relevant during the activation/execution of the subprogram. A concrete example of an activation record is called an *activation record instance*.

In this "simple" subprogram setup, activation records are of fixed size. There can also only be one active version of a given subprogram at a time (since recursion isn't supported with just Static Variables).

15 Object-Oriented Programming

15.1 Introduction

Defn 245 (Object-Oriented Programming)**.** *Object-oriented programming* (*OOP*) is a programming paradigm designed around the manipulation of Objects that accurately simulate real-life situations. These languages must provide 3 key language features:

1. Abstract Data Type
2. Inheritance
3. Dynamic binding of method calls to methods.

The use of an object-oriented language allows for Abstract Data Types to be reused. This allows for programmers to reuse large portions of code and only having to change the parts that are necessary. This increases productivity and increases Reliability.

15.2 Inheritance

The use of Inheritance allows for Abstract Data Types to be reused between pieces of software. This, in turn, means code is reused, but some of it must be specialized. Additionally, some of the underlying problem has categories of objects that are related, both as siblings (similar to each other) and as parents and children (having a descendant relationship).

Defn 246 (Class)**.** A *class* is the Abstract Data Type of Object-Oriented Programming languages. They are the building blocks of the programming paradigm.

Classes have 2 kinds of Variables:

1. Class Variable
2. Instance Variable

They also have 2 kinds of Methods:

1. Class Method
2. Instance Method

Defn 247 (Object)**.** An *object* in an Object-Oriented Programming setting is an instance of a Class.

Defn 248 (Subclass)**.** A Class that is derived from another Class is called a *subclass* or *derived class*.

Defn 249 (Superclass)**.** A Class from which other Classes are derived is called a *superclass*, or *parent class*.

A Subclass can differ from its Superclass in several ways.

1. The Superclass can define some of its Variables or Methods to have private access, which means they won't be visible to the Subclass.
2. The Subclass can add Variables and/or Methods to those inherited from the Superclass.
3. The Subclass can modify the behavior of one or more of its inherited Methods. A modified Method has the same name, and often the same protocol, as the original.

Defn 250 (Method)**.** A *method* is a subprogram that defines the operations of/on Objects of a Class.

Defn 251 (Message)**.** The calls to Methods are sometimes called *message*s. Passing a message is different than calling a traditional subprogram. A message is sent to an Object that is a request to execute a Method. These Methods can operate on themselves, or on other Objects.

Defn 252 (Message Interface)**.** Thus, the entire collection of Methods is called the *message interface*, or *message protocol* of the Object.

Defn 253 (Override)**.** A new Subclass's Method can *override* a Superclass's Method. This is then called an *overridden method*.

Overriding a Method is useful because it allows us to provide an operation in the Subclass that is similar to the one in the Superclass, but specialized for Objects of the Subclass.

Defn 254 (Instance Method)**.** An *instance Method* is a Method that **only** operates on instances of the Class.

Defn 255 (Instance Variable)**.** An *instance Variable* is a unique set of Variables defined by the initializing Class, and stores that Object's state.

Defn 256 (Class Variable)**.** *Class Variable*s belong to and operate on the Class itself, rather than just the Objects of a Class. This means there is only **one** copy for the whole Class, rather than a copy for every instance of this Class.

For example, a counter of the number of instantiated Objects of that Class would require a class variable.

Defn 257 (Class Method)**.** A *Class Method* performs operations on the Class as a whole, rather than the individual Objects of that Class. This means there is only **one** copy for the whole Class, rather than a copy for every instance of this Class.

Defn 258 (Inheritance)**.** *Inheritance* is the act of making a Superclass and deriving (inheriting) a Subclass from it. This creates a new Class as the original, but is slightly different.

Remark 258.1. The ability to reuse code through Inheritance means that dependencies between these Abstract Data Types can form. This technically defeats the purpose of Abstract Data Types' independence of each other. However, increasing the reusability without introducing dependencies may not be possible. In addition, these dependencies may make solving the issue in the problem domain easier.

Defn 259 (Single Inheritance)**.** In *single Inheritance*, a Class can **only** have 1 parent Superclass. However, any Superclass can have an infinite number of derived, child, Subclasses.

When a relationship between Classes that use single Inheritance is visualized, a derivation tree is made.

Defn 260 (Multiple Inheritance)**.** In *multiple Inheritance*, a Class can have multiple parent Superclasses. Any Superclass can also have an infinite number of derived, child, Subclasses.

When a relationship between Classes that uses multiple Inheritance it can be visualized as a derivation graph.

Remark 260.1 (Diamond Dependency)*.* In a language that supports Multiple Inheritance, a problem can arise when a single class inherits from 2 Classes, which both inherit from one superclass. In the class derivation graph, this ends up creating a diamond shape, hence the name *diamond dependency*. If any superclasses have methods that are overridden, then without an explicit statement to tell the compiler which method to use, it is impossible to decide which to use. The approach C++ takes is that the programmer must specify which superclass' method to use.

15.2.1 Top Superclass

To have a class hierarchy in a programming language, there must be a "top" Superclass somewhere in the hierarchy, one class from which all subsequent classes derive from.

In Java, there is a single top class `java.lang.Object`.

However, C++ takes the stance of not having any implicit single top Superclass. Instead, it can have multiple top Superclasses, and also allows any number of top-level classes without a Superclass.

15.2.2 Interfaces in Java/JVM Languages

In Java, one way to create an Abstract Data Type and Typeclass-like system is with its interfaces.

These provide a mechanism for defining new types that describe families of operations that are not classes. In Java, these types are called interfaces, and traits in Scala. Interfaces/traits may be sub-types of other interfaces/traits, and classes may be subtypes of interfaces/traits, but since there are no implementations to inherit from an interface/trait, these two do not act as classes.

15.2.2.1 Abstract Data Types Using Interfaces For any Class that *implements* one of these interfaces/traits, it must provide the implementation details. For example we will use the `IntVector` example from Section 9.3, written in Scala and Java.

```
trait IntVector {
  def length : Int
  def append(v : Int)
  def get(v : Int) : Int
}
```

```
interface IntVector {
    public int length;
    public void append(int v);
    public int get(int v);
}
```

15.2.2.2 Parametric Polymorphism Using Interfaces

We can also make this more general, like before, with Formal Type Parameters.

```
trait Vector[T] {
  def length : Int
  def append(v : T)
  def get(v : Int) : T
}
```

```
interface Vector<T> {
    public int length;
    public void append(T v);
    public T get(T v);
}
```

15.2.2.3 Bounded Parametric Polymorphism Using Interfaces

If we wanted to define a bounded interface, using interfaces, to implement the `PriorityQueue` from Section 9.4, we coudl write the scala code below.

```
/* Requires that any subtype provide an operation that llows for comparison
 * against any object.
 */
trait Cmp {
  def isGreaterThan(v : Any) : Boolean
}
```

```
/* T <: Cmp ensure that T MUST be a subtype of Cmp (implements Cmp)
 */
trait PriorityQueue[T <: Cmp] {
  def push(v : T)
  def getTop() : T
}
```

15.2.2.4 F-Bounded Parametric Polymorphism Using Interfaces

The previous examples are *too* restrictive, in that they require that the object T be able to compare itself against ***any other*** object, when we just want to compare this type against inself, i.e. Integers against Integers. We can do this by boudning the subtype that `Cmp` compares against. If we rewrite the `Cmp` trait into the `GT` trait, using *F-Bounded Polymorphism*, we end up with the code below. More precisely, bounding a type parameter by a type that contains the type parameter itself is called *F-bounded polymorphism*.

```
trait GT[T] {
  def isGreaterThan(v : T) : Boolean
}
```

```
/* By bounding T to be a subtype of GT[T], we only allow T's which have the
 * isGreaterThan operation specified.
 */
trait GTPriorityQueue[T <: GT[T]] {
  def push(v : T)
  def getTop() : T
}
```

Thus, anything that wishes to implement the `GTPriorityQueue` must be specified similarly as what is shown below.

```
class UserRequest extends GT[UserRequest] {
  ...
  def isGreaterThan(v : UserRequest) = ...
}
```

15.3 Dynamic Binding

Defn 261 (Dynamic Dispatch)**.** *Dynamic dispatch* is a way to dynamically bind Messages to Method definitions.

An example of Dynamic Dispatch is shown below.

```
class E {
    int f (B b) { ... }
    int f (A a) { ... }
    int g (A a) { ... }
}

class F extends E {
    @Override
    int g (A a) { ... }
}

public class C {
    public static void main(String[] args) {
        A aa = new A() ; // static type : A, dynamic type : A
        A ab = new B() ; // static type : A, dynamic type : B
        E ee = new E() ; // static type : E, dynamic type : E
        E ef = new F() ; // static type : E, dynamic type : F
        ee.f(aa); // => E.f(A)
        ee.f(ab); // => E.f(A)
        ee.g(aa); // => E.g(A)
        ef.g(aa); // => F.g(A)
    }
}
```

Defn 262 (Polymorphic)**.** The reference to a Subclass is *polymorphic* if it could also point to something in the Superclass. Thus, during runtime, the correct reference must be resolved through polymorphism and executed through Dynamic Dispatch.

For example, in Java

```
public class A {
    public draw() { ... }
}
public class B extends A {
    @Override
    public draw() { ... }
}
public static void main(String[] args) {
    A myA = new A();
    B myB = new B();
    myA.draw(); // Draws the thing defined by instructions in class A's draw() method
    myB.draw(); // Draws the thing defined by instructions in class B's draw() method
    // We need to determine if we should call B's draw() or A's draw in this case
```

In some cases, a Class sit so high in the hierarchy that it is never instantiated to an Object. In that case, its functionality can be stripped out, and the Class can turn into an Abstract Class, with Abstract Methods.

Defn 263 (Abstract Method)**.** An *abstract Method* (pure virtual Method in C++) is a Method that is missing its body. However, because the Method definition is still present, all Subclasses **are required** to implement the body/functionality of the abstract Method, turning it into a concrete Method.

Defn 264 (Abstract Class)**.** An *abstract Class* (abstract base Class in C++) is a Class that contains one or more Abstract Methods. **These usually cannot be instantiated**.

Any Subclass of an abstract class must provide implementations of all the inherited Abstract Methods.

15.4 Design Issues for Object-Oriented Languages

15.4.1 Exclusivity of Objects

There are 3 main schools of thought/implementation when it comes to the implementation of Objects in an Object-Oriented Programming language.

1. **EVERYTHING** is an Object. Basic scalar types, collections, programs, everything is an Object. In the purest version of this model, all Classes are treated the same way.
   - The benefits of this are the uniformity and elegance of the language. It is incredibly regular, where everything looks the same and behaves similarly, without exception.
   - The disadvantages of this is mainly the cost of executing everything as an Object. There are Messages and the method calls which are slower than simpler, primitive, operations.
2. Keep the original, traditional collection of types, but add the object typing model on top.
   - The benefits of this is that the language retains its speed of execution on the primitive types.
   - The drawbacks of this is the confusing syntax and semantics that can be output by this.
3. Use an imperative type structure for the Primitive Data Types, but all structured types as objects.
   - The benefit of this is the execution speed of operations is relatively similar
   - The downsides are the complications that can appear in the language, and the need for *wrapper Classes* that allow these nonObject types and Objects to interact.

15.4.2 Are Subclasses Subtypes?

This question boils down to the question "Does an 'is-a' relationship hold between a Subclass and its Superclass?" Or, can we replace every occurrence of the Superclass with the Subclass, and everything would still work?

Most programming languages significantly restrict the ways in which a Subclass can be constructed, to ensure that this holds true. The characteristics of Subclasses that are Subtypes are:

- The Methods of the Subclass that Override the Superclass's are type compatible with their corresponding Methods. This means that if we were to replace any call with its overridden version, the program would still be type-safe.
- Every overriding Method must have the same number of Formal Parameters as the Overriden Method.
- The Data Types of the Formal Parameters are also compatible.
- The return types of the Method are also compatible.

15.4.3 Single and Multiple Inheritance

It might make sense for a Class to inherit from more than 1 Superclass, if it has components of each. However, this leads to 2 issues:

1. Complexity
   - If the multiple Superclass defined something with the same name, it must be explicitly resolved.
   - What if a Subclass inherit from multiple Superclasses that themselves inherit from the same Superclass? This forms the *diamond inheritance problem*.
2. Efficiency
   - There needs to be an extra heap access to access the second Superclass of an Object.

Multiple Inheritance can lead to complex program organizations. Many who have attempted to use Multiple Inheritance have struggled to design Classes that benefit from having multiple Superclasses. This also leads to greater dependency and complexity between Classes.

15.4.4 Allocation and Deallocation of Objects

The first question is, where are Objects allocated from?

- Allocated from anywhere, like Abstract Data Types. Means they could be
  - Allocated from the call stack
  - Allocated from the heap
  - Allocated in static memory
- Heap-Dynamic Variable only.
  - Uniform design among Objects

  - Simpler to access objects through the Pointers/References to the Object.
  - Is allocation implicit or explicit?
- if Objects are stack-dynamic, then the extra information that a Subclass has might be truncated in Memory when copied to a location that held a Superclass. This is called *object slicing*.

Is deallocation implicit or explicit? If its implicit, then what is the storage reclamation solution? If its explicit, then how do we handle the Dangling Pointer problem?

15.4.5 Dynamic and Static Binding

Dynamic binding of Messages to Methods is essential. But the question is, are all of these bindings implicitly dynamic? The alternative is that the user must specify if the binding is to be dynamic or static.

The benefit of static bindings is that they are much faster.

15.4.6 Nested Classes

The primary motivation of nesting Classes is to hide information. If a Class is only needed by one other Class, why make the single-use Class public? Is nesting of these Classes inside Methods allowed?

The biggest issue with this is visibility. Which of the facilities of the nesting Class are visible in the nested Class? Which of the facilities of the nested Class are visible in the nesting Class?

15.4.7 Initialization of Objects

Are Objects initialized with values when the are first created? If so, how are these Objects initialized? What values are they initialized with?

This requires us to ask if these Objects must be initialized manually, or through an implicit mechanism. If we are instantiating an Object of a Subclass, then we also need to instantiate the necessary materials for the Superclass.

15.5 Subtyping in Object-Oriented Languages

This is a continuation from Section 8.3.

From earlier, the 3 things that Object-Oriented Programming must support are:

1. Dynamic method binding or *Dynamic Dispatch*
2. Inheritance
3. Support for Abstract Data Types

The subtyping rules for methods follow the subtyping rules for subroutines, so the following overriding between classes C and D is perfectly safe:

```
1 class C {
2     A g (B b) { ... } // type B -> A
3 }
4
5 class D extends C {
6     @Override
7     B g (A a) { ... } // type A -> B
8 }
```

One complication here is the implicit `self` or `this` reference (or pointer, in C++). This self-reference allows methods to access their own state, so its type is always fixed to be a subtype of the type of the declaring class.

This self-reference is technically a 'hidden parameter' to each method, but since a method can only be called on an object whose dynamic type is a subtype of the class in which that method was defined, it is safe for us to use this more precise type information.

15.6 Abstract Data Types in Object-Oriented Languages

16 Functional Programming

Recall the definition of a Functional Programming Language.

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.

Because of its close relationship to mathematics, it is much easier to develop mathematical techniques for reasoning about the behavior of programs developed in functional languages. These techniques are important tools for helping us to ensure that programs work properly without having to resort to tedious testing and debugging that can only show the presence of errors, never their absence. Moreover, they provide important tools for documenting the reasoning that went into the formulation of a program, making the code easier to understand and maintain.

But, in short, these types of languages rely on the *evaluation of expressions*, rather than the execution of commands.
In this section, the functional language used is Standard ML.

16.1 Side Effects in Functional Languages

In a strict definition of Functional Programming Languages, Side Effects are not allowed. However, Side Effects include:

- Writing a file to disk
- Sending packets over the network
- Writing to standard output

However, if a functional language did not include these, the language is severely hampered in its usability.[5] To get "around" this issue, there are 3 ways functional languages can handle these non-pure actions.

1. The language was *imperative first*. These are languages that:
   - Were originally Imperative Programming Languages, like C, C++, Java
   - Can be adapted to behave like functional languages if they are written correctly.
2. The language was *functional first*. These are languages that:
   - Are by default true Functional Programming Languages, like Haskell, F#, the ML-family
   - Allow Side Effects to be contained within the functions themselves, and not outside them.
   - Has the compiler track these functions with Side Effects so they are handled correctly.
3. The language is *monadic*. There is only a single language that supports this model right now, Haskell. A monad is:
   - An abstraction or generalization over ordered things
   - These things may have side-effects, which are tagged.
   - Monads allow for an imperative-like state to be introduced to a functional environment. However, the state exists *ONLY* for the monad's function(s) and its/their children.
   - Monads may also allow for state to be present throughout a program, but permission to access the state must be passed to functions, like an Actual Parameter.

16.1.1 Building Side Effects into a Language

Since inference rules describe Semantics in a "pure" mathematical way, side effects become additional evaluation results. This means our evaluation relation now relates the tuple ($expr$, previous state) with the tuple (val, state afterwards).

For example, to model printing, you could write

$$\frac{(e, P) \Downarrow (v, P')}{(\texttt{PRINT}\, e, P) \Downarrow (v, P' + [v])} \tag{16.1}$$

e, The expression to be evaluated and printed.
P, The previous state of the system, in this case, the list of all previously printed numbers.
v, The value the expression e evaluates to.
P', The state of the system after printing, in this case, the list of all printed numbers up to this one.
$P' + [v]$, The list of printed numbers up to this point, with the value v appended.

Remark. In Equation (16.1), `PRINT`x would also return the evaluation result of x, so `PRINT(PRINT7)` would print the number 7 twice.

[5] Some functional languages are like this, but they are not widely used, so they are not discussed.

16.1.2 Modelling Side Effect Updating of Variables

In most Imperative Programming Languages, variables can be updated, i.e. have their value changed from under them. This is different than in Functional Programming Languages, which technically create a new variable with the same name and assign the new value.

Modelling updateable variables is a bit more tricky but still possible. The challenge here is that our previous notion of Environments is no longer powerful enough, because a subexpression may change global state, as seen in the `MYSTERY` code below.

```
VAR x : INTEGER; // x is a Global Variable
PROCEDURE P() : INTEGER =
BEGIN
   x := 0;
   RETURN 0
END
// What is below is the "main" program.
BEGIN
   x := 1;
   PRINT x + P() + x
END
```

There are two models for modelling this. The simpler one turns the environment into a parameter of the evaluation relation, so that you get

$$(e, E) \Downarrow (v, E') \tag{16.2}$$

However, the model in Equation (16.2) is too weak for real languages like Java or Scala, since those lanugages permit Aliasing.

The second model can support Aliasing. To do this, we usually use a two-step look-up process, where the first lookup (with the Environment E) determines a variable's memory address a. The second step uses a so-called store to translate the address to a value. This gives rules like the following:

$$\frac{E[x] = a \;\; S[a] = v}{E \vdash (x, S) \Downarrow (v, S)} (\textit{Read Variable}) \tag{16.3a}$$

$$\frac{E \vdash (x, S) \Downarrow (v, S') \;\; E[x] = a}{E \vdash (x := e, S) \Downarrow (v, S'[a \mapsto v])} (\textit{Update Variable}) \tag{16.3b}$$

$$\frac{E \vdash (S, s_1) \Downarrow (v_1, S') \;\; E \vdash (S', s_2) \Downarrow (v_2, S'')}{E \vdash (s_1; s_2, S) \Downarrow (v_2, S'')} (\textit{Sequence}) \tag{16.3c}$$

With this kind of model it becomes easy to model Pass-by-Reference versus Pass-by-Value Semantics, or to describe the `&` and `*` operators from C and C++:

$$\frac{E[x] = a}{E \vdash (\&x, S) \Downarrow (a, s)} (\textit{Get Address}) \tag{16.4}$$

$$\frac{E \vdash (e, S) \Downarrow (a, S') \;\; v = S'[a]}{E \vdash (*e, S) \Downarrow (v, S)} (\textit{Dereference Address}) \tag{16.5}$$

16.2 Standard ML

Defn 265 (Standard ML)**.** *Standard ML*, *SML*, *Standard Meta Language* is a general-purpose, modular, Functional Programming Language with compile-time type checking and type inference. It uses Pass-by-Value for its evaluation of Actual Parameters to functions.

Computation in Standard ML consists of sequential evaluation of expressions. Each expression has 3 characteristics:

1. It may or may not have a *Data Type*.
2. It may or may not have a *Value*.
3. It may or may not have an *effect*. These include:
   - Raising an exception
   - Modifying memory
   - Performing I/O
   - Sending a message on a network

Every expression is **required** to have at least one Data Type, making them *well-typed*. If an expression does **not** have a Data Type, it is called *ill-typed*, and cannot be evaluated. The *type checker* is in charge of checking the types of all expressions.

Well-typed expressions can be evaluated to determine their value. This evaluation can be interrupted by an exception and/or a run-time fault.

Defn 266 (Soundness)**.** The *soundness* of the Data Type system ensures the accuracy of predictions made by the type checker.

16.2.1 Type Checking

A type in Standard ML requires 3 things:

1. A *name* for the type, for later uses.
2. The *values* that the type can have.
3. The *operations* that may be performed with this type.

There are several basic Data Types in Standard ML.

1. `int`, corresponding to integer numbers
2. `real`, corresponding to floating-point numbers
3. `char`, denoted `#''c''`
4. `string`, denoted `''string''`
5. `bool`, either `true` or `false`

16.2.2 `If`-Statements in Standard ML

The `if`-statement syntax is shown below.

```
if conditionalExp then exp1 else exp2
```

There are a few things to note. The `conditionalExp` must evaluate to an expression of a `bool` type. The expressions `exp1` and `exp2` **MUST** have the same type. However, even if both branches have the same type, one branch can still raise raise an exception during runtime.

Remark. This is a *conditional expression*. This means that the result from the conditional expression could be assigned to a variable as a value.

16.2.3 Declarations in Standard ML

By default, both variables and types have global scope. However, Standard ML has introduced methods to limit the scope of things. These are described more fully in Paragraph 16.2.3.3.

16.2.3.1 Variables Variables in Standard ML are slightly different than Variables in Imperative Programming Languages. In Standard ML, variables maintain the same value throughout their life. If a variable is rebound, then a whole new memory cell is created for the new value and the old discarded. For example,

```
val a : int = 5;
val a = a + 1; (* This creates a whole NEW "a" to use *)
(* Create 2 new values at once *)
val a : int = 6 and b : real = 3.14;
```

It is important to note that variables are *bound-by-value* in Standard ML. This means that the right-hand side is evaluated first, meaning that the rebinding of `a` in the previous example is expanded to `val a : int = 5 + 1`.

16.2.3.2 Types Aliased Data Types can be created in Standard ML. For example,

```
type float = real; (* Creates a new type "float" that has a range the same as the base "real" type *)
type count = int and average = real; (* Introduces 2 new types at once *)

(* There is one case that does not work, because 2 new types are being introduced AT ONCE. *)
type float = real and average = float;
```

16.2.3.3 Limiting Scope TODO!!

16.2.4 Functions in Standard ML

Functions and variables share the same expression namespace! This means that things like this can happen:

```
fun g(y : int) = y * y;
val g = g(7); (* Uses the function g above, then overwrites it *)
```

16.2.5 Pattern Matching in Standard ML

Defn 267 (Pattern Matching)**.** *Pattern matching* is an N-Way Selection construct. **IT IS AN EXPRESSION, MEANING EACH CASE MUST RETURN THE SAME TYPE**. In addition, when pattern matching, the the value(s) that match a particular case can be bound to names. This allows for use use of the matched value to be used for some computation.

```
val emptyList : string list = []; (* emptyList : string list *)
val singletonList = ["1"]; (* val singletonList : string list*)
val multiList = ["1", "2", "3"]; (* val multiList : string list*)

(* checkList is a function with a pattern matching expression. *)
fun checkList toCheck = case toCheck of
                            [] => print "empty list\n"
                          | [a] => print ("list with one element: " ^ a ^ "\n")
        (* The a is matched and then a is bound with the value of the single element in the list *)
                          | _ => print "more than one element in list\n";
(* val checkList = fn : string list -> unit *)

checkList emptyList;
(* Prints "empty list\n"*)

checkList singletonList;
(* Prints "list with one element: 1\n *)

checkList multiList;
(* Prints "more than one element in list\n*)
```

16.2.6 Data Type Inferencing

Defn 268 (Data Type Inferencing)**. TODO!!**

16.2.7 Algebraic Datatypes

Defn 269 (Algebraic Datatype)**.** An *algebraic datatype* can be viewed as a generalization of datatypes already present. Additionally, these can behave like Enumeration Types or like Unions. These can then have Pattern Matching performed on them.

A simple Algebraic Datatype can be visualized as a cardinal direction. For example,

```
datatype direction = North
           | South
           | East
           | West;
```

We can apply pattern matching to Algebraic Datatypes as well.

```
fun dirstr North = "North"
  | dirstr South = "South"
  | dirstr East =  "East"
```

```
  | dirstr West = "West";

(* direction -> string *)
```

Another example, slightly more complicated is the construction of a list type, that we will call `lst`.

```
datatype 'a lst = NIL
                | CONS of 'a * 'a lst;

fun len (NIL) = 0
  | len (CONS(e, tail)) = 1 + len(tail);

fun tostr (NIL) = NIL
  | tostr (CONS (i, tail)) = CONS(Int.toString i, tostr(tail));

fun inclist (NIL) = NIL
  | inclist (CONS (i, tail)) = CONS(i+1, inclist(tail));

fun genlst (0, _) = NIL
  | genlst (k, init) = CONS(init, genlist (k-1, init));
```

Now say we wanted to simplify the total number of functions possible in our `lst` code. Many of our functions have the same construction and the same recursion. We can simplify them down to a `map` function, which takes a list and a function as Formal Parameters and applies the function to each element of the given list, returning a new list in the process. The code for this `map` function is shown below.

```
fun map (NIL, f) = NIL
  | map (CONS(i, tail), f) = CONS(f(i), map(tail, f));
(* The type specification of map is => fn : 'a lst * ('a -> 'b) -> 'b lst *)
(* NOTE that this map is different than the one built into SML *)
(* The type specification of the built-in map is => ('a -> 'b) -> 'a list -> 'b list *)
```

16.3 Continuations

Defn 270 (Continuation)**.** *Continuations* allow us to abstract in the opposite direction of Variables. They allow us to abstract over the context in which the values get used.

They describe the required input to an expression (variables, values, other expressions, etc.), any computations required by the expression (adding a value, computing another expression, etc.) the command to be executed on a line (the instruction, `print`, etc.) and then the next statement.

For example, in the code snippet of `Mystery` below, there is a continuation on each line where a Variable is used in place of a value.

```
VAR x : INTEGER;
VAR y : INTEGER
BEGIN
  ...
  PRINT x + 1;
  PRINT y
END
```

So, there are continuations on lines 5 and 6. The one on line 6 is "take an integer value, `PRINT` that value, and then `END` the program". The continuation on line 5 is "take an integer value, add 1 to it, `PRINT` the result, then `PRINT` y, and finally `END` the program".

```
open SMLofNJ.Cont;

datatype looper =  INIT of looper cont
                 | LOOP of (string list * looper cont)
```

```
5
6  fun loop (input_list : string list) =
7      let
8          val (h::list, cont) = case callcc (fn c => INIT c) of
9                                    INIT c         => (input_list, c)
10                                 | LOOP (rest, c) => (rest, c)
11         val _ = print "  ---\n"
12         val (h::list, cont') = case callcc (fn c => INIT c) of
13                                    INIT c         => (h::list, c)
14                               | LOOP (rest, c) => (rest, c)
15         val _ = print h
16         val _ = print "\n"
17     in case list of
18              []  => (print "Done\n")
19                     (* bug was in the line below: " throw cont " instead of " throw cont' " *)
20           |  [_] => throw cont' (LOOP (list, cont'))
21         |  _     => throw cont (LOOP (list, cont))
22     end
```

17 Logic Programming

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 Trigonometry

B.1 Trigonometric Formulas

$$\sin(\alpha) \pm \sin(\beta) = 2\sin\left(\frac{\alpha \pm \beta}{2}\right)\cos\left(\frac{\alpha \mp \beta}{2}\right) \tag{B.1}$$

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

B.2 Euler Equivalents of Trigonometric Functions

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

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

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

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

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

B.3 Angle Sum and Difference Identities

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

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

B.4 Double-Angle Formulae

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

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

B.5 Half-Angle Formulae

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

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

B.6 Exponent Reduction Formulae

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

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

B.7 Product-to-Sum Identities

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

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

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

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

B.8 Sum-to-Product Identities

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

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

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

B.9 Pythagorean Theorem for Trig

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

B.10 Rectangular to Polar

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

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

B.11 Polar to Rectangular

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

C Calculus

C.1 L'Hopital's Rule

L'Hopital's Rule can be used to simplify and solve expressions regarding limits that yield irreconcialable results.

Lemma C.0.1 (L'Hopital's Rule)**.** *If the equation*

$$\lim_{x\to a}\frac{f(x)}{g(x)}=\begin{cases}\frac{0}{0}\\ \frac{\infty}{\infty}\end{cases}$$

then Equation (C.1) *holds.*

$$\lim_{x\to a}\frac{f(x)}{g(x)}=\lim_{x\to a}\frac{f'(x)}{g'(x)} \tag{C.1}$$

C.2 Fundamental Theorems of Calculus

Defn C.2.1 (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

$$\int_a^b f\left(x\right)dx=F\left(b\right)-F\left(a\right) \tag{C.2}$$

Defn C.2.2 (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

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

then

$$\begin{aligned}\frac{d}{dx}\int_a^x f\left(t\right)dt&=f\left(x\right)\\ F'\left(x\right)&=f\left(x\right)\end{aligned} \tag{C.3}$$

Defn C.2.3 (argmax)**.** The arguments to the *argmax* function are to be maximized by using their derivatives. You must take the derivative of the function, find critical points, then determine if that critical point is a global maxima. This is denoted as

$$\operatorname*{argmax}_x$$

C.3 Rules of Calculus

C.3.1 Chain Rule

Defn C.3.1 (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,

$$\begin{aligned}f'(x)&=g'(x)\cdot h(x)+g(x)\cdot h'(x)\\ \frac{df(x)}{dx}&=\frac{dg(x)}{dx}\cdot g(x)+g(x)\cdot\frac{dh(x)}{dx}\end{aligned} \tag{C.4}$$

C.4 Useful Integrals

$$\int\cos(x)\,dx=\sin(x) \tag{C.5}$$

$$\int\sin(x)\,dx=-\cos(x) \tag{C.6}$$

$$\int x\cos(x)\,dx=\cos(x)+x\sin(x) \tag{C.7}$$

Equation (C.7) simplified with Integration by Parts.

$$\int x\sin(x)\,dx = \sin(x) - x\cos(x) \tag{C.8}$$

Equation (C.8) simplified with Integration by Parts.

$$\int x^2\cos(x)\,dx = 2x\cos(x) + (x^2-2)\sin(x) \tag{C.9}$$

Equation (C.9) simplified by using Integration by Parts twice.

$$\int x^2\sin(x)\,dx = 2x\sin(x) - (x^2-2)\cos(x) \tag{C.10}$$

Equation (C.10) simplified by using Integration by Parts twice.

$$\int e^{\alpha x}\cos(\beta x)\,dx = \frac{e^{\alpha x}\big(\alpha\cos(\beta x) + \beta\sin(\beta x)\big)}{\alpha^2+\beta^2} + C \tag{C.11}$$

$$\int e^{\alpha x}\sin(\beta x)\,dx = \frac{e^{\alpha x}\big(\alpha\sin(\beta x) - \beta\cos(\beta x)\big)}{\alpha^2+\beta^2} + C \tag{C.12}$$

$$\int e^{\alpha x}\,dx = \frac{e^{\alpha x}}{\alpha} \tag{C.13}$$

$$\int xe^{\alpha x}\,dx = e^{\alpha x}\left(\frac{x}{\alpha} - \frac{1}{\alpha^2}\right) \tag{C.14}$$

Equation (C.14) simplified with Integration by Parts.

$$\int \frac{dx}{\alpha+\beta x} = \int \frac{1}{\alpha+\beta x}\,dx = \frac{1}{\beta}\ln(\alpha+\beta x) \tag{C.15}$$

$$\int \frac{dx}{\alpha^2+\beta^2x^2} = \int \frac{1}{\alpha^2+\beta^2x^2}\,dx = \frac{1}{\alpha\beta}\arctan\left(\frac{\beta x}{\alpha}\right) \tag{C.16}$$

$$\int \alpha^x\,dx = \frac{\alpha^x}{\ln(\alpha)} \tag{C.17}$$

$$\frac{d}{dx}\alpha^x = \frac{d\alpha^x}{dx} = \alpha^x\ln(x) \tag{C.18}$$

C.5 Leibnitz's Rule

Lemma C.0.2 (Leibnitz's Rule)**.** *Given*

$$g(t) = \int_{a(t)}^{b(t)} f(x,t)\,dx$$

with $a(t)$ and $b(t)$ differentiable in t and $\frac{\partial f(x,t)}{\partial t}$ continuous in both t and x, then

$$\frac{d}{dt}g(t) = \frac{dg(t)}{dt} = \int_{a(t)}^{b(t)} \frac{\partial f(x,t)}{\partial t}\,dx + f\big[b(t),t\big]\,\frac{db(t)}{dt} - f\big[a(t),t\big]\,\frac{da(t)}{dt} \tag{C.19}$$

D Complex Numbers

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

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

where

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

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.

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

D.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.

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

Defn D.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.

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

D.1.1 Complex Conjugates of Exponentials

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

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

D.1.2 Complex Conjugates of Sinusoids

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

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

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

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.