EDAP05: Concepts of Programming Languages - Skills Sheet

Karl Hallsby

Last Edited: January 14, 2020

Contents

1 Language Critique

The book provides a convenient common framework for arguing about the benefits and disadvantages of language features and programming languages in general. Table 1.1 and, more generally, Section 1.3 in the book describe a number of axes along which we can try to understand the effect of language features and the qualities of programming languages.

1.1

You should be able to recognise and describe all the characteristics listed in Table 1.1, as well as the three basic criteria listed therein.

- Readability — How easy is it to read the program's source code and understand what's happening?
  - Simplicity — How "simple" is the language?
    - Is it like brainfuck and hard to read and write (leading to reliability issues), or like Python that is quite literate (in terms of how similar it can be to English)?
  - Orthogonality — How do the number of features present in a language overlap?
    - Operator overloading is one instance where we sacrifice orthogonality for a simpler to read and write (and more reliable) language.
    - For example, the addition operator is typically overloaded for integers and floating-point numbers, because in math we use the same operator for both, but have different meanings and required operations in computers.
  - Data Types — How closely do data types that we use in the program match our needs? Do these overlap with things that may be present in the real world (in the case of OOP languages)?
  - Syntax Design — Is the syntax, i.e. the way the language must be read and written easy or difficult?
    - If it is difficult, is there a reason?
    - This may influence the reliability of the language if it is hard to understand how the program is working.
- Writability — How easy is it to write a program's source code and understand what's happening? This does not just refer to the writer, but also the reader (who may be the same persion or someone else) later.
  - Support for Abstraction — Can we take repeated details of our program and make them more general than before?
    - For program operations (subprograms), instead of having to embed common operations everywhere in a program that may need to perform this calculation, can we make it a subprogram that can be called repeatedly and operate as before?
    - For data, how naturally can we represent more complicated things? For example, a binary tree in C is a `struct` with 2 pointers and a holding value. In older languages (FORTRAN), it was 3 arrays in parallel.
  - Expressivity — Using the features built into the language, how easy is it to **naturally** express the computation that we want to perform?
    - How much of the language's code do we need to describe some computation?
- Reliability — How sure are we (the writer) that our code is actually doing what we want, without error? If there are errors, how can we ensure that they are handled?
  - Type Checking — Using the data types from earlier, what kind of checking can we perform to ensure our program operates the way we intend it to?
    - Think of SML's type checking, or Rust's type checking for a strongly-typed and statically checked language.
    - C/C++ is weakly-typed because of pointers (which can be overwritten in any way, without regards to the type checker), although it is statically checked.
    - Python is strongly-typed, but dynamically checked. This is the reason any variable can be any type at any point of execution, but the operations that can be performed on them are limited.
    - JavaScript is weakly-typed and dynamically checked. Being weakly-typed means the language allows errors that could be caught to instead "fly under the radar", until the thing is used again. Being dynamically checked means that these are only found during runtime of the language.
  - Exception Handling — If an error happens during program execution (a file is not present, network connection lost, etc.) how well can we handle that? How easy is it to read and write a program where these can occur?
  - Restricting Aliasing — Can we restrict the number of things that point to a single memory cell? This is important for type checking and resource deallocation.

1.2

For some language feature, you should be able to recognise the characteristics and criteria (from Table 1.1 (Section 1.1)) affected by that feature's presence or absence.

1.3

For some language feature, you should be able to recognise how the feature's presence or absence affects the language, based on the three criteria from Table 1.1 (Section 1.1).

1.4

For two related language features, you should be able to compare them, based on the three criteria from Table 1.1 (Section 1.1).

1.5

For some language feature, you should be able to recognise and explain how it can affect the Cost considerations for training, compile time, execution time, and cost of poor reliability as outlined in Section 1.3.7.

- Training Time — How long does it take programmers to learn this language?
- Compile Time — How long does it take the compiler to compile the program?
  - A slower compiler may catch more errors or generate more optimized machine code
  - A faster compiler allows for quicker development and faster programmer feedback.
- Execution Time — When the program is running, how much time does it take to run?
  - This can be measured in many ways: CPU time, CPU cycles, human time, etc.
- Poor Reliability — If the program runs, but does so unreliably, it must be fixed. What is the cost of fixing the reliability issues with this/these program(s)?

2 Language Implementation

You should be familiar with the following concepts:

2.1

Language implementation via **pure interpretation**, **compilation**, and **hybrid implementation**.

- Pure Interpretation
  - Python
  - The program runs on an interpretation layer between the hardware and itself. (VM usually)
  - Quicker development with faster feedback on issues
  - No compilation (May be possible to compile down to VM-specific code, but not machine code.)
  - Slower execution time (Must interpret each command, break down to possible CPU actions, and write the machine code on-the-fly.)
- Compilation
  - C
  - Program runs ON the computer's hardware
  - Slower development, must compile for some issues to be found (Type Error). Must run for other issues to be found (Null pointer).
  - Compile down to machine code for quick runtime.
  - Fast execution. Human-readable code already in machine code, just need to load into memory and begin execution.
- Hybrid Implementation
  - Java
  - Program may start by being interpreted, but parts in use will start to be compiled while running.
  - Runs some parts through an interpreter, and some pieces are compiled.
  - Theoretically, same kind of fast development as interpreted languages, with the more frequently-used code being compiled for speed.
  - Hypothetically, Fast execution for the parts that are compiled.

2.2

Language implementation with the help of a **just-in-time compiler**.

- A Just-In-Time (JIT) compiler starts by having the code run in an interpreted mode.
- After some time, the most frequently used portions of code are compiled, to improve performance, and the interpreted parts are replaced by the compiled parts.
- Java uses this.

2.3

The concept of language **run-time system**.

- These are the systems in place that were put in by the compiler/interpreter to help the program run.
- These can include: Error handlers, resource deallocators (Garbage collectors), and type coercers.

3 Syntax

Syntax describes the possible structure (or form) of programs of a given programming language. Backus-Naur Form (BNF) grammars have emerged as the standard mechanism for describing language syntax. BNF grammars used to describe languages when communicating with language adopters and compiler implementors. There are also many tools (particularly the yacc and antlr families of programs) for automatically generating parsers, programs that recognise whether an input program matches a grammar and, if it does, execute user-defined actions upon encountering certain language constructs.

3.1

You should be able to determine whether a given property of a language is part of the syntax, of the static semantics, or of the dynamic semantics.

- Syntax — This is done with the context-free grammar and parsing the input symbols
- Static Semantics — This is done during the implementation of the compiler/interpreter. These define some of the operations possible and how they should behave.
  - For example, the `+` operator must be defined to mean addition when given integers and concatenation when given strings.
  - It is required that the syntax be defined and relatively fixed for the semantics to be well-defined.
- Dynamic Semantics — This is done while the program is running. These determine the behaviors of the operators while the program is in-use.

3.2

You should be able to read a BNF grammar and understand the difference between terminals and nonterminals.

- Terminals — Symbols that represent a terminating point in the recursion.
  - Typically, these are symbols that mean we have reached the end of our recursion.
  - In a programming language, these are usually keywords.
  - In this course, they are typically symbols that lack the $\langle$ and $\rangle$.
- Nonterminals — Symbols that continue the recursion of a production to another production with the same symbol.
  - These are things used to construct a parse tree of what we have input to the parser/grammar.
  - In this course, they are typically denoted as $\langle A \rangle$.

3.3

Given a BNF grammar, you should be able to write down examples of programs that can be generated by the grammar.

- This means you must be able to run through and recurse through the grammar.

3.4

Given a BNF grammar, you should be able to tell whether a given program can be generated by the grammar. If the program is generated by the grammar, you should also be able to generate a parse tree for the program.

- Essentially, you must run the grammar on the given program and confirm whether the program could be accepted/generated by the grammar.

3.5

You should be able to determine whether a given (small) BNF grammar is ambiguous (the problem is undecidable in general, so this skill only pertains to practically relevant examples as covered in the textbook).

- There needs to be more than one way to generate a parse tree given a grammar.

3.6

Given a BNF grammar, you should be able to determine the associativity of any operator used therein.

- Left associative operators are left-recursive in their grammar, so the parentheses build up towards the left, with the first operation to be performed deepest in the parse tree on the left, with all subsequent operations higher in the tree and to the right.
- Right associative operators are right-recursive in their grammar, so the parentheses build towards the right, with the first operation to be performed deepest in the parse tree to the right, with all subsequent operation higher in the tree and to the left.

3.7

You should be able to describe the difference between an object language and a meta-language.

- Object language — A language that is used for something, for example, C. These can be meta-languages in some cases.

- Meta-language — A language that describes other languages, Context-free grammars, for example.

3.8

Understand **arity**, **fixity**, and **precedence**, and **associativity** of operators

- Arity

  1. Unary — An operator that takes a single argument. Logical NOT is an example of this, as well as `i++`.
  2. Binary — An operator that takes 2 arguments. Most mathematical and logical operations are like this.
  3. Ternary — An operator that takes 3 arguments. The ternary `if` operator is an example of this.

- Fixity

  - Prefix — Operator symbol before arguments, $+\,3\,2 = 5$
  - Infix — Operator symbol between arguments, $3 + 2 = 5$
  - Postfix — Operator symbol after arguments, $3\,2\,+ = 5$

- Precedence

  - Given several operators in a single line, which one should have the highest priority?
  - Only way to figure this out is to run tests which force certain operations to have mutually disjoint outputs. Meaning for some inputs, you will get different outputs for operations with different precedences.

- Associativity

  - Given several operators of the same precedence, what order should they be evaluated in?
  - Most mathematical and logical operators are left-associative, meaning they should be left-recursive in their grammar.
  - The exceptions to this are exponentiation and function types, which are right-associative, and thus must be right-recursive in their grammar.

4 Natural Semantics

There are many ways to describe semantics. In this course, we focus on natural semantics (also known as Big-Step Operational Semantics).

4.1

You should be able to read a specification of natural semantics.

4.2

You should be able to undestand how **expressions** and **values** relate to each other.

1. An expression is built up of values.
2. An expression can return a value or another expression.
3. Values must be defined in terms of some mathematical numerical thing. If `v` is input as an integer, then $v \in \mathbb{Z}$ is the conditional requirement for $\texttt{v} \Downarrow v$.

4.3

Given a natural semantics and a parse tree (or unambiguous expression), you should be able to compute the semantics of the given program.

- You must use the semantic rules provided, and repreatedly recurse until the semantics of the entire program has been defined.

4.4

Given a natural semantics and a BNF grammar, you should be able to tell whether any parts of the semantics are undefined.

- The BNF grammar defines what symbols may be used. So if the semantics uses symbols that the BNF does not defined, the semantics are undefined.
- If there are operator symbols defined, but the semantics does not use them, then the semantics are undefined.
- If an operator's semantics do not recurse enough to handle all cases, then they are undefined.

4.5

Given an understanding of what a state-free expression language is supposed to do, you should be able to write down a simple natural semantics for it.

- Given a basic state-free expression, like an expression $1 + x$ where $x \in \mathbb{Z}$
- Create the necessary rules for such an expression to be able to exist

4.6

You should be able to understand the concepts of Environments in the context of natural semantics and be able to utilise it when reading and reasoning about natural semantics.

- These behave like a map, where if a meta-language's variable (meta-variable) has been defined, then it stores a value.
- They are updated like this: $E[c \mapsto 2]$
- They are read (Meta-variable values are retrieved) from like this: $E(c) \Rightarrow 2$
- The empty environment is $E_\emptyset = \{\}$

5 Bindings and Lifetimes

5.1

You should understand the difference between static and dynamic type binding and be able to take advantage of either property in your programming.

- Static Type Binding

  - C-Family Languages
  - Once things have been given a type, they cannot change their type for their lifetime.
  - Once something has reached the end of its lifetime, it is deallocated.
    * The name and storage bindings can be reused.

- Dynamic Type Binding
  - Python, JavaScript
  - Throughout a single thing's lifetime, its type can change as many times as the programmer wants. Only subsequent actions will require the binding to be of a certain type.

5.2

Given a syntax, a compiler and a run-time system for a language, you should be able to determine whether the language is using static or dynamic type binding.

- Static
  - You try to redefine the type binding for a variable.
  - For instance, you declare `x` to be an integer, then later declare `x` to be a string.
  - Java/Rust/SML/Haskell, for examples.
  - In a strongly-typed language, this would be a (before runtime) compiler error.
  - In a weakly-typed language, this error would occur during runtime, and may actually be passed over completely by giving default values or storing the error in the place of the variable's result.

- Dynamic
  - You try to redeclare something that is of one type to another and the language allows that.
  - Python, for example.
  - For instance, you declare `x` to be of type integer, then later declare `x` to be of type string.
  - The language allows this, and any subsequent operations happens with the new type binding.
  - In a strongly-typed language, this would be allowed, but invalid operations would be found during (before runtime) compilation stage.
  - In a weakly-typed language, this would be allowed, but may "fly under the radar" until you try to use the results of an erroneous operation.

5.3

Concepts: static binding, stack-dynamic binding, explicit heap-dynamic binding, and implicit heap-dynamic binding.

- Static
  - **Global** and **Constant** variable bindings are put here.
  - `const int x = 4;`
  - The storage binding lasts as long as the program is in execution.
  - These can be stored in the actual program binary when it is compiled and saved to disk, because they are unchanging.

- Stack-Dynamic
  - The typical variable bindings go here.
  - `int x = 4;`
  - They are usable within the scope they are declared in.
  - Their lifetime is the same as their scope.

- Explicit Heap-Dynamic
  - The programmer must **EXPLICITLY** allocate memory from the heap for the storage binding.
  - The programmer must **EXPLICITLY** deallocate memory from the heap when the binding reaches its end of life.
  - The lifetime of the variable is determined by the programmer.
  - `int* x = malloc(); *x = 4; free(x);`
  - The lifetime of an explicit heap-dynamic variable storage binding is separate from its scope.

- Implicit Heap-Dynamic

  - The compiler/runtime system determines the lifetime of the binding.
  - The compiler/runtime system handles the allocation of memory and deallocation of memory.
  - The compiler/runtime creates a garbage collection to ensure things are deallocated when their lifetime is over.
  - The lifetime of an explicit heap-dynamic variable storage binding is related to its scope, for the most part, they are the same.

5.4

Given a syntax, a compiler and a run-time system for a language, you should be able to determine which storage binding(s) the language is using.

- Implicit vs. Explicit Heap-Dynamic:
  - There needs to be a keyword/reserved word for explicit heap operations. If none present, not explicit.
  - The explicit keyword/reserved word may be given in the grammar, or it is a core function built into the language.
  - If none of that, implicit.
- Stack-Dynamic vs. Static:
  - Static bindings are done *statically*, and last the *life of the **program***.
  - If the static value is modified INSIDE a subprogram, it is updated **throughout the rest of the program's** execution.
  - Thus, static bindings do not allow for recursion to happen with the static binding's value.
  - If the stack-dynamic value is modified inside a subprogram, and this value is not visibly global, the value is not modified.
- Stack-Dynamic vs. Heap-Dynamic:
  - The difference here is where they are stored.
  - This can be hard to tell, you would need to be able to see the memory in which the program is running.
  - Larger things (arrays) are usually stored on the Heap, while smaller things (integers) are more likely to be stored on the stack.
  - However, the previous rule is **INCREDIBLY** flexible and is not a certainty. (It also depends on language implementation, to a degree.)
  - If there is a way to explicitly allocate memory on the heap, then you can find out much more easily.

5.5

Concept: lifetime of a variable.

- The lifetime of a variable is the period of time which it has a value binding.
- This is distinct from being in-scope.
  - In-scope, out of lifetime: An explicit heap-dynamic storage binding, the memory could be freed too early and the value that was there is going to be used later.
  - Out-of-Scope, in lifetime: A stack-dynamic variable is used in a program, but is not passed to the subprogram. It is out-of-scope of the subprogram, but still alive in the parent program.
- For stack-dynamic storage bindings, their lifetime and scope are usually the same, but can be different (as seen above).
- For static storage bindings, they are alive all the time, but may be out-of-scope or shadowed by other bindings with the same name.

5.6

Concepts: allocation and deallocation of a heap-dynamic variable.

- Allocation
  - If implicit, programmer doesn't need to worry about allocation, at worst has to write a constructor.
  - If explicit,
    - Programmer must tell compiler/runtime system **how much** memory to allocate.
    - Programmer must keep track of the pointer/reference that points to that memory.

            - Programmer must make sure that the pointer is used correctly (including pointer arithmetic)

- Deallocation
    - If implicit, programmer doesn't need to worry about deallocation, at worst, has to write a destructor. Lets the garbage collector handle it.
        - Many garbage collector algorithms.
            - Reference counter
                1. Keep count of how many pointers point to things in memory.
                2. If $count \leq 0$, deallocate.
                3. Cannot handle linked lists or circular structures.
            - Mark-and-sweep
                1. Start by identifing all memory as dead (includes stack-dynamic and static variables).
                2. Follow all pointers to heap and traverse the heap, marking everything that was reached as valid.
                3. Once done iterating over pointers and recursively descending, delete everything not yet marked.
    - If explicit,
        - Programmer must tell compiler/runtime system to deallocate the memory at some specified time.
        - Most of these errors are only found during runtime (Rust has special capabilities to figure this out),
            - If programmer forgets → **memory-leak**.
            - If programmer deallocates twice → **double-free**.
            - If programmer deallocates, but still uses it → **corrupted memory**.

5.7

Concept: binding time, especially the difference between static and dynamic binding times.

- Static Binding Time
    - These are things that are bound when the program is compiled (before it is run).
    - Static storage bindings are done statically, during compilation.
    - That goes along with their names too.
    - The binding of names and types to functions is static.
- Dynamic Binding Time
    - The binding of storage is dynamic for all categories except static.
    - In a dynamically typed language, the type bindings for variables is dynamic.
    - In an OOP language, you can dynamically type objects if you declare them with a supertype.

6 Scoping

6.1

You should understand the difference between static scoping and dynamic scoping and be able to exploit either in your programming.

- Static Scoping
    - The scope of the variable is determined by its placement within the program.
    - This is done by looking at **scopes** and their nesting (In block-delimited languages, scopes are usually the same as BLOCKS) and determining parents.
    - The benefit of these are:
        - Readability:
            - Easy to find out exactly what the program is going to do, just by reading the program.
        - Writability:
            - By knowing exactly what the program will do with a quick read of the source code, you can more quickly write.
        - Reliability:
            - All variables can be determined statically, before the program runs.
            - So, if the program compiles these without errors, you are assured that every use of a variable is allowed by the language.

        · But, you now have more things to check during static semantic analysis.

- Dynamic Scoping
  - The scope of the variable is determined by the order in which subprograms were called.
  - Although variables may be redefined in a subprogram, the original value is not lost.
  - The original value is restored when the scope that changed it ends.

6.2

Given a language implementation, you should be able to write a program to determine whether the language uses static or dynamic scoping.

- You need to generate at least 3 scopes, 2 of which are nested in the first and are siblings.
- This can be done with subprograms.
- One of the child scopes must get to the other scope.
- Results from code below:
  - Static Scoping: When `sub1()` prints `x`, it will print 4. When `main()` prints `x`, it will print 4.
  - Dynamic Scoping: When `sub1()` prints `x`, it will print 6. When `main()` prints `x`, it will print 4.
- Because `sub1()` and `sub2()` are sibling scopes, when the `x` in `sub1()` is looked up in the referencing environment for its value, it looks up to `main()`.

```
int main() {
  int sub1() {
    printf("%d", x);
  }
  int sub2() {
    int x = 6;
    sub1();
  }
  int x = 4;
  sub2();
  printf("%d", x);
}
```

6.3

Concept: referencing Environment.

- This is a **list of variables to their types, values, and any other information that must be maintained** about the variable.
- Functions also go in referencing environments.
- This can be thought of as the list of names that are usable at any given point in time, and the environment maps these names to their other information.

7 Types

7.1

Concepts: the types of integers, floating-point numbers (floats), and decimal numbers (decimals)

- Integers can either be signed or unsigned.
  - They can be various sizes, depending on the processor architecture.
  - Signed:
    - Represent numbers between -2^{b-1} and $+2^{b-1}-1$
    - The first bit is the sign bit
    - If the number is negative, it is stored in 2's complement format
  - Unsigned:
    - Represent numbers between 0 and 2^b-1, where b is the number of bits available

- Floating-point can be either 32-bits or 64 bits.
  - The first bit is the sign bit
  - The last 31 or 63 bits are defined by the respective IEEE standard.
- Decimal numbers, for example in banking, should **NOT** be floating point numbers.
- **THESE ARE NOT ITEROPERABLE BY JUST REINTERPRETTING THE BITS**. The integer 1 and floating-point 1 are represented completely differently in binary.

7.2

Concept: the type of booleans

- 2 Values:
  - True
  - False
- Are the basis to preform logical operations.
- Generally used for deciding if an operation should occur.
- Returned by the $<$, $>$, $\leq$, $\geq$, $==$, $!=$ operations.

7.3

Concept: enumeration type.

- Type that takes one value at a time from a list given at the types definition.
- A colors enumeration type might have red, green, and blue. If a variable is of this enumeration type, at any point in time, you can change the value.
- There are 2 ways to represent enumeration types:
  1. A fancy way to bind integers to descriptive names.
     - From the example above, `red=0`, `green=1`, and `blue=2`.
     - This does not allow for good error checking.
     - If the language implementation implicitly coerces types, the type of an enumeration might become an integer, allowing for things like `red + 1` which would evaluate to `green`.
  2. Discrete values.
     - In this case, the elements of an enumeration type are distinct things with distinct representations that have type checking done on them.
     - There is no way to coerce values in the enumeration type to anything else.
     - This allows for strong type checking to be performed on these.

7.4

Concepts: character and string types

- Character
  - This heavily depends on the encoding used.
  - ASCII uses 8 bits, 1 byte. UTF-8 may use up to 4 bytes, but the first characters are the same as ASCII for interoperability.
  - The variable storing the character will usually only be able to store single characters for this reason.
- String
  - There are 2 ways to store strings:
    1. Arrays of characters
    2. A primitive type with its own binary representation and storage allocation scheme
  - Are strings statically sized or dynamic?

7.5

Concept: subrange types

- These allow you to select **JUST** a range of values.
- If defined in terms of some other type, it may be type-equivalent.
- If you only want to work with numbers from 1 to 10 and statically disallow anything else, you can create a subrange type from 1 to 10. Then, the static system will ensure all values passed are inside that subrange.
- A subrange with a larger range is a subtype of one with a smaller range.
  - This is because the larger range contains the same information as the smaller range.
  - But, it also contains more information than the smaller subrange, thus it is a subtype.

7.6

Concept: record types

- Aggregate several types into one.
- Similar to a tuple.
- **Name** the things stored inside the record and use those names to access the values. This is what separates a record from a tuple.
- A record has memory equal to the size of all types allocated each time.

7.7

Concept: tuple types

- Aggregate several types into one.
- Similar to a record.
- The values stored inside are integer-indexed.
- A record has memory equal to the size of all types allocated each time.

7.8

Concept: list types

- A mutable (maybe) and dynamic list of any type.
- The list could contain things of the same type and of different types.
- These can be infinite and can be lazily evaluated.
- Heavily used in functional languages.

7.9

Concept: associative array types

- There is a key-value pair.
- Using the key (usually its hashed), a location is found in the structure and the value is stored.
- When trying to access a value, you provide a key.
- These are Hashtables/dictionaries in languages.
- The issue of storing multiple values in the same spot and having hashes collide are implementation issues.

7.10

Concept: union types, both free and discriminated

- A union type is a type that **takes one of several types**.
- Free:
  - The memory that is used for the storage of data is interpreted differently based on the type that is retrieving data from it.
  - Thus, type-checking cannot be performed statically, and rarely can it be done dynamically.
  - In C, the memory would be retrieved like normal, but interpreted differently based on the type of the caller.
- Discriminated:
  - When storing values, when one of the storage options is chosen, it also becomes tagged with that type.
  - This way, type checking can be performed statically.

7.11

Concept: operator overloading

- Done in the natural semantics.
- Need to provide typing rules in the semantics
- Based on the typing information that is present in the semantic rules, choose which production to follow.
- The operation must be defined for the type for it to work.

7.12

Concepts: strong typing and weak typing

- Strong Typing:
  - If there is a type error (**STATIC OR DYNAMIC**), the language prevents this operation.
  - Also **ALL** type errors are found.
- Weak Typing:
  - If there is a type error (**STATIC OR DYNAMIC**), the langauge does **NOT** prevent the operation.
  - In some languages, these erroneous operations **might return some default value**.
  - Not all type errors may be found.

7.13

Concepts: type checking and the differences between dynamic type checking, static type checking.

- Type Checking:
  - Checking that everything **HAS** a type
  - Checking that all operations and expressions use their correct types
  - Ensure that things with certain types are used in the correct way, i.e. an integer doesn't have an attemp at array indexing happening.
  - If there is an issue with types, an error is raised. This is handled differently for strongly- and weakly-typed languages.
    * Strongly: Stop compilation (if static type checking) or stop execution (if dynamic type checking).
    * Weakly: Maybe stop compilation (if static [depends on language implementation]) or use a default type value (if dynamic [this is just one solution]).
- Dynamic:
  - This type checking occurs during runtime
  - Use static and dynamic semantics
- Static:
  - This type checking occurs before runtime, during compilation
  - Use static semantics

7.14

Concepts: type equivalence, including the difference between nominal and structural type equivalence

- Type Equivalence
  - If one type T can be used in place of type U, without type coercion, T and U are equivalent types.
- Nominal Type Equivalence
  - They are equivalent if they are defined in the same declaration or in declarations that use the same data type.
  - They have the same compiler/runtime system unique name/id.
- Structural Type Equivalence
  - There are several steps to check this:

1. If the types have the same type
2. Same number of fields (if applicable), and each of the fields is pairwise nominally equivalent.
3. If each field is nominally equivalent and are references/pointers perform structural equivalence on them
4. You perform these steps until you reach a state where no step above is correct, i.e. contradicts some aspect of the type.

7.15

Concept: type constructors

- Creates a new type
- Does this by generating a unique internal system name for that type.

7.16

Concept: typing rules as part of type systems, and how to read such rules and utilise them in your reasoning about program semantics

- These behave quite similarly to semantic rules for operators.
- These define the types of values and expressions.
- For expressions they define:
  - Their input type(s)
  - Their output types
- These **DO NOT** define the actual mechanics of the operation, for example, they do not say that the `+` operator is for addition, but rather the types to input to the `+` symbol and the resultant type.

7.17

Concept: the type preservation property, also known as subject reduction, of type systems

- If an operator is adding 2 integers, you expect an integer to return.
- More generally, if an expression e is of type τ, and $e \Downarrow v$, then v should also be of type τ.

7.18

Concepts: type parameters and parametric polymorphism

- Type Parameters
  - These are parameters that must be passed when working with types, functions, or other operations.
  - For many things, you can think of these just like normal parameters, but with types instead of values.
- Parametric Polymorphism
  - These are parameters that must be passed to things to construct them for that type, but the general case is the same for each type.
  - For example, an array should be able to hold any number of one type.
  - The basic operations on this array (`length`, `index-access`) are the same between all type cases.

7.19

Concept: function types for subroutines

- The arrow, $\rightarrow$, `->`, is right associative
- It states the input types, if there is more than 1, then it specifies them as a tuple
- It states the output type.

7.20

Concept: type classes

- For a given type, you can "add" it to a class such that it **must** implement some functionality.
- If you create a new type that needs to have the $>$ operation defined, in Rust, you must add it to the `PartialOrd` type class, then define what the $>$ operator means on your type.
- Now, this type can be used anywhere a regular type would be, and you can use it in any case where the type is bound correctly.

7.21

Concept: subtyping

- If the values on a type encompass **more** data, i.e. all integers, a record with 3 integer fields, etc. any type that restricts the range of values for that type is a **subtype**.
- Thus, a record with 3 integer fields is a subtype of a record with 2 integer fields, because the larger record contains all the same information as the other record, but has additional information. Think of these like OOP subclasses.
- An subrange with a smaller range is a supertype to one with a larger range.
- $T :> U$ means U is a subtype of T. The "arrow" always points towards the subtype.
- Any type is a subtype and supertype of itself.

7.22

Concept: covariance and contravariance of type parameters, and the arrow rule

- Covariance:
  - If type is an input to a function, the function would accept more general data.
  - If a function takes a subrange from 1 to 10, but you give it 1 to 20, it contains all the same information for the function to run, so it does.
- Contravariance:
  - If a type is an output from a function, the function is also able to output more specific data.
  - If a function returns a subrange from 1 to 10, it can also return a subrange from 1 to 3. So long as the output is not dependent on having a subrange from 1 to 10, the function can output the subrange from 1 to 3 without issue.
- Arrow Rule:

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

  - T_1 is a supertype of T_2
  - U_1 is a subtype of U_2
  - $T_1 \to U_1$ means we can broaden the type of the input without problems
  - $T_2 \to U_2$ means we can narrow the type of the output without problems
  - This means that $T_1 \to U_1$ is a supertype of $T_2 \to U_2$ (supertype meaning the same type or an actual supertype)

7.23

Concept: bounded parametric polymorphism

- This is a way to restrict what types can be used as parameters for something.
- Using our > and `PartialOrd` example in Rust from Section 7.20, if we had a function that sorted our type, we would **require** that all types used in the function implement this operator and be a part of the `PartialOrd` type class.
- This works for parametric polymorphism in much the same way.

7.24

Concept: definition-site variance and use-site variance of type parameters

- Definition-Site Variance
  - When defining the function/subprogram, the programmer specifies how the function can be varied.
- Use-Site Variance
  - The programmer must define the variance of the type when creating an instance of that type.
  - In Java:
    * Use-Site Covariance: You can only use functions which return the original type or have no relation to the original type.
    * Use-Site Contravariance: You can only use functions which take in the original type, or have no relation to the original type.
    * Use-Site Invariance: You can only use functions which have nothing to do with the original type. This can be viewed as if **both** use-site covariance and use-site contravariance apply.

7.25

Concept: Algebraic Datatypes as in Standard ML and their use in pattern-matching

- Algebraic Datatypes
  - An algebraic datatype is a generalization of datatypes already present.
  - They also allow for record-like things to be created
  - If used for unions, they form tagged unions
  - Each case of an algebraic datatype can have arbitrarily many types present
- Pattern-Matching
  - There needs to be a case for every case in the algebraic datatype
  - There is a variable _ that matches everything
  - You can also use this on tuples to match on the type in the index that you are interested in and disregard the other informations.

7.26

Concept: Automatic Type Inference

- Many things can be inferred by the information given to a function.
- If the type of things are not specified, the system is usually able to figure out the types required by looking at the static semantics of the language.
- Sometimes this isn't perfect and some type information **must** be given for the function/thing to be correctly typed.
- If a parameter is used in addition, it can be determined that the parameter is of the same type as the other operand.
- The last expression evaluated is the return value from the entire function
- If there is no information that is usable, it assigns type variables that behave like regular variables, but for the types a function can take. In SML, denoted `'a`, `'a` $= \alpha$.
  - In the case of a function with these, every instance of the same type variable is replaced by the type used.
  - Different type variables, `'a` and `'b`, are allowed to be the same.

8 Expressions

8.1

Concept: arithmetic expressions

- Pretty obvious.
- `+`, `-`, `*`, `/` or `div`, `%`
- The first 3 (`+`, `-`, and `*`) will only return integers if they are given integers.
- Division is defined by the language designer. There might be separate symbols for integer and real-number division.

8.2

Concepts: boolean expressions and relational expressions

- Logical operators:
  - `||` (OR)
  - `&&` (AND)
  - `~` (NOT)
- Relational Expressions: All of which return boolean values (True or False)
  - `>`, greater than
  - `<`, less than
  - `>=`, greater than or equal
  - `<=`, less than or equal
  - `==`, equal (equality is differently defined for different types)
  - `!=`, not equal

8.3

Concepts: Different forms of object equality, including reference equality and structural equality

- Primitive Equivalence
  - If the bit patterns of the types are the same
- Reference Equivalence
  - If 2 things point/refer to the same location in memory
- Structural Equivalence
  - If the 2 variables have identical structures.
  - Involves comparing each type present and recursively checking if they match at some point
  - Allows things to have the same structure, but be different, if the user desires..
- User-Defined Equivalence
  - Requires either a user-defined type, or an overriding of an already present type's equality
  - The user defines what equality between 2 instances of their type

8.4

Concept: operand evaluation order and how it affects the outcome of programs

- Evaluation Order:
  - Do we go from left to right? Right to left? Somewhere from the middle out?
  - Only way to check is have one operand change the state somewhere to let us know that it has been evaluated first.
    - Could be done with 2 subprograms that print different things, and whichever prints first is evaluated first.
- Outcome, assuming no other major rules change (like assignment):
  - This might make some programs slower, because they have to perform heavy recursion before a simple addition, for example.
  - Overall, the change is fairly minor. The programmer just has to get used to the syntax and execution direction.

8.5

Concept: short-circuit evaluation and how it affects the outcome of programs

- Short-Circuit Evaluation:
  - Skip unnecessary boolean evaluations.
  - OR
    - If the first operand is 1, then logical OR will always evaluate to 1, no matter what the second operand is.
    - Why evaluate the second operand if it's not needed to find an answer?
  - AND
    - If the first operand is 0, then logical AND will always evaluate to 0, no matter what the second operand is.
    - Why evaluate the second operand if it's not needed to find an answer?
- Outcome:
  - Obvious, but **ONE OF THE OPERANDS IS NOT EVALUATED**!
  - If we are required to check the second operand, or both operands, short-circuit evaluation defeats that.

8.6

Concept: referential transparency and side effects

- Referential Transparency:
  - This is a thing if any 2 expressions can replace each other in a program, without changing the action.
  - For example, if $1 + 5$ is highly prevalent in a program, 6 would replace the $1 + 5$ expression.
- Side Effects:

  - A function changing the state of a program "beneath its feet".
  - For example, a function deleting the last element in a list every time it is called, or a function to increment a global counter.
  - Printing to the screen, writing to disk, getting a network packet are all stateful effects, so functional languages have facilities to handle them.
  - Purely functional languages remove the possiblity of making side-effects.

8.7

Concept: list comprehensions and their semantics

- List Comprehensions
  - Succinctly iterate over a list and apply a function to all values present in the list.
  - Some languages (Python) allow for conditionals to be present in the list comprehension as well.
- Semantics
  - Different semantics depending on the language.
  - Python3: `incremented = [x + 1 for x in originalList]`
  - Haskell: `incremented = [x + 1 | x <- originalList]`

8.8

Concept: type coercion expressions, both explicit and implicit, including narrowing conversions and widening conversions

- Type Coercion Expressions:
  - Explicit
    * Requires the programmer to explicitly tell the programming language to convert a value between types.
    * This is usually only used on the narrowing conversion, because it is a potentially unsafe operation, some of the original data could be lost.
  - Implicit
    * The programmer doesn't have to explicitly tell the language to convert a value between types, but they can.
    * This is usually only used on the widening conversion, because it is usually a safe operation, meaning the original data is preserved.
- Conversions:
  - Narrowing
    * Must be done explicitly, if the language even allows it.
    * For example, moving from a floating-point number to an integer. Or, converting between a 64-bit integer to a 32-bit integer.
    * This is dangerous because data can be lost by narrowing a type.
  - Widening
    * Usually done implicitly.
    * For example, converting between an integer and a floating-point number to perform an addition.
    * This is a safe operation because the original data is preserved, but the range of possible values has been extended.

8.9

Concept: conditional expressions

9 Statements and Control Structures

9.1

Concept: assignment statements, including compound assignment

- Assignment:

  - The regular assignment.
  - Use the storage binding that was set aside and perform a value binding.

- Compound Assignment:

  - Short-hand to specify an operation on a value and reassign that new value to the variable with the same name used.
  - `count += 1`

9.2

Concept: two-way selection statements

- Standard `if` statement.
- If the condition is true, take one path, else take the other path.
- When we reach the end of either path, they come back together.

9.3

Concept: multiple-selection statements

- Standard `switch` statement or `if-elseif-else` statement
- Based on the condition, choose one path of many.
- If the first condition is not met, then choose another.
- The different paths usually need to be broken up by a `break` statement for them to be completely separate.
- Without the `break`, one path might run into another, because the condition is only evaluated once and then checked in a `switch` statement.
- In an `if-elseif-else` statement, the condition is evaluated as many times as there are `if` and `elseif` branches.
- At the end of all paths, they come back together.

9.4

Concept: counter-controlled loops

- `while`-loops can be used for these, where their condition checks if the count is correct.
- `for`-loops are a construction that eases the writing of counter-controlled loops
- The counter can increment at the beginning or at the end of the loop.

  - `for`-loops have this done at the end.
  - `while`-loops can have it done wherever the programmer specifies, so long as it is done.
  - If a programmer forgets to manage the counter in a `while`-loop, an infinite loops occurs.

- Counter-controlled loops are special logic controlled loop, where the condition is checking if a counter has met some threshold.

9.5

Concept: logically controlled loops

- `while`-loops are used for these.
- The condition is evaluated before entering the loop, and evaluated after each iteration through the loop.
- These can be used to create an infinite loop while waiting for some stateful change, like keyboard input, or a network packet.
- The condition must evaluate to a boolean value, in old versions of C either 0 or 1.

9.6

Concept: data structure-controlled loops

- These are colloquially called `for-each`-loops.
- They are used on more complex data structures, like vectors, lists, etc.
- They return one item on each iteration of the loop.
- The loop ends when the data structure has iterated over each element once.

10 Subprograms and Parameter Passing

10.1

Concepts: subprograms, including formal arguments and actual arguments

- Subprogram:
  - A piece of code that has been abstracted away from the main processing.
  - This allows for a common operation to be maintained separately from other bodies of code.
  - This also allows for easy reuse and easy debugging.
- Formal Arguments:
  - The arguments that are defined along with the subprogram definition.
  - The type of the argument is given, and a name local to this subprogram is also given.
  - When passed an Actual Argument, they behave like locally-scoped variables.
- Actual Arguments:
  - The actual variables that are passed as arguments to the subprogram in the code.
  - These are just the expressions that will evaluate to become the values in the subprogram that are refered to by the name the Formal Arguments defined.

10.2

Concept: local variables in subprograms

- These are related to the fact that subprograms have their own scope.
- These are also related to the fact that subprograms work best when there are stack-dynamic variables.

10.3

Concept: nested subprograms

- Inside a subprogram is another subprogram.
- If a block of code is only needed inside a subprogram subprogram, why make a globally visible subprogram?
- This allows us to create a subprogram inside a subprogram, which is only scoped inside the outer subprogram.

10.4

Concepts: parameter passing modes

- These are ways to pass Actual Arguments to subprograms.

10.4.1

by value

- This is the basic way to pass things around.
- This is an in-mode passing method. Meaning the value is evaluated at the caller location and passed to the subprogram.
- For each subprogram, a value is passed to the binding.
- Meaning, if a subprogram takes an integer `3` in, the bit value of `3` is passed to the subprogram and is stored/used.

10.4.2

by result

- This is an out-mode passing method. Meaning the Actual Argument receives their value(s) when the subprogram exits.
- This is distinctly different than a return value, because the last value that the Formal Argument held is what is returned.

10.4.3

by value-result

- This uses the principles of both Pass-by-Value and Pass-by-Result.
- The values passed to the subprogram are evaluated at the caller's lcoation, then passed to the subprogram.
- The subprogram may modify these values.
- When the subprogram is done, the last value bound to the Formal Arguments are passed back into these variables and the subprogram returns.

10.4.4

by reference

- Instead of passing the bit pattern that is the actual value, pass a pointer/reference to the value.
- This allows us to pass huge amounts of data without much penalty during subprogram call, but now there is apointer redirection everytime we dereference the Formal Argument.
- This means that any change that occurs to the pointer's data will be reflected everywhere else in the program where the memory the pointer points to is valid.

10.4.5

by name

- Expressions are passed as Actual Arguments to the subprogram.
- These expressions are evaluated everytime the Formal Argument is evaluated.
- This means that if $1 + 5$ were passed as an Actual Argument, to a Formal Argument of `int x`, every time `x` is used, `x` is essentially replaced by $1 + 5$.
- If the passed Actual Argument is a function, this means that the function is evaluated several times.
- This is not generally used because of this runtime overhead.

10.4.6

by need

- Similar to Pass-by-Name.
- Only used by Haskell right now.
- The Formal Parameter's actual expression is evaluated **AT MOST ONCE**.
- The result of this evaluation is cached by the runtime system for the current function.
- Every subsequent evaluation of that Formal Parameter will return the cached result instead.

10.5

Concepts: subprograms as parameters, subprograms as return values, Closures

- Closures:
  - You have a function and some context in use as a "variable"
  - This context could contain some binding of values to names that are used within the closure.
  - The closure acts like a normal function (for all intents and purposes), the main difference is that it will have a function's type signature instead of a more traditional type.
  - The "variable" can have expressions that match the type signature of the closure passed to it, and it will evaluate them as if they were normal functions.
- As Parameters:
  - A function that takes a function requires a closure be passed to it.
  - These are heavily used in functional programming, where a function only modifies data based on the input data.
- As Return Values:
  - A function that returns a function is returning a closure.
  - These are heavily used in functional programming, where a function only modifies data based on the input data.

10.6

Concept: activation records

11 Pointers, References, and Arrays

11.1

Concepts: pointers and references

- Pointer:
  - Points to a location in memory
  - Can be dereferenced to get actual value in memory
  - Pointers can be nested
  - Pointers can have arithmetic done on them to move them around arbitrarily
- Reference:
  - Refers to an object or value in memory
  - Can be dereferenced to get the actual value
  - References cannot be moved by pointer arithmetic

11.2

Concept: the dangling pointer problem

- If there are 2 pointers, but one of them deallocates the memory from the heap, the other one is pointing to garbage.
- This is the dangling pointer problem.
- If a pointer goes out of scope and is deallocated, but the memory isn't, that's a memory leak.

11.3

Concepts: garbage collection in the forms of reference counting and mark-and-sweep collection

- This only happens for implicitly heap-allocated programming languages.
- Reference Counting:
  - The runtime system keeps count of the number of references to a thing in memory
  - When the number of references increases, nothing happens.
  - When the count gets to 0, the system will deallocate the memory.
  - This doesn't work for cyclical structures.
  - While incremental, it also introduces a lot of operations between any actual programmer calculations.
- Mark-and-Sweep:
  - The runtime system will pause program execution occassionally and perform this algorithm.
    - Consider all memory invalid/not-in-use.
    - Mark all memory in static memory and on the stack as valid.
    - Go through pointers on the stack and in static memory and follow them.
    - Recursively descend through the pointers, marking that memory as valid.
    - Once the entire memory segment has been visited, anything not marked as valid is deallocated.
  - This is done less frequently, but it uses up a lot more system resources.
  - It also handles deallocation of circular structures that may still be on the heap because the structure isn't reachable from anywhere else.

11.4

Concept: arrays

- Arrays are continuous blocks of memory
- The size is calculated based on the number of elements expected to be stored and the size of the elements.
- These are quick to access because the index of any element is a fixed distance away from the start of the array.
- The head of the array, the thing that is at index 0, is actually a pointer to the memory of index 0.
- The `array[index]` syntax is just a nice way to present `*(array + (index * sizeOfElement))`

11.4.1

static arrays

- The array is statically sized
- The storage is statically bound
- Depending on the compiler, language, and memory setup, the values in storage could be bound statically too
- They can only hold one type, to allow for indexing.
- This exists inside the static memory of the program
- Like static variables, these exist for the duration of a program's execution.

11.4.2

fixed stack-dynamic arrays

- These are the usual arrays (relatively small-sized)
- Placed on the stack in memory
- The memory storage is allocated dynamically, during runtime
- The index range is bound statically, meaning they are of fixed size
- They can only hold one type, to allow for indexing.

11.4.3

fixed heap-dynamic arrays

- These are the usual arrays (relatively large-sized)
- Placed on the heap in memory
- The memory storage is allocated dynamically, during runtime
- The index range is bound statically, meaning they are of fixed size
- They can only hold one type, to allow for indexing.

11.4.4

heap-dynamic arrays

- These are less commonly found
- Placed on the heap in memory
- The memory storage is allocated dynamically
- The index range is also bound dynamically, meaning they can be of any size
- They can only hold one type, to allow for indexing.

12 Abstract Datatypes

12.1

Concepts: information hiding and encapsulation

- The way in which we implement something shouldn't affect how it is used
- If we have a stack, someone should just be able to use the stack.
- They don't need to know if the stack is implemented with an array, a linked list, a binary tree, or anything else.
- These have related children with subprogram abstraction and data abstraction

12.2

Concept: abstract datatypes

- This is a way to specify data types that a person doesn't need to know the implementation for.
- There are 3 things an abstract datatype requires:
  1. The interface is what the programmer will interact with; the face the abstract datatype shows the world.
  2. The implementation is how the interface was realized, the actual code behind the abstract datatype.
  3. A reasonable specification of what the expected actions of an operation are, for example, one would expect a `length` subprogram to return the length of the abstract datatype, not a random number.

12.3

Concept: generic abstract datatypes, that is, abstract datatypes that take one or more type parameters

- These are just abstract datatypes that require a type parameter.
- For example, a vector abstract datatype would require a type parameter to know the types of the things its going to be storing.
- If the vector was implemented with an array, it could use the bit size of the type to correctly index your information.
- This is a combination of parametric polymorphism and abstract datatypes, but can be extended with bounded parametric polymorphism to form interesting structures.

13 Object-Oriented Programming

13.1

Concept: object-oriented language

- Requires 3 things:
  1. Support for abstract data types
  2. Inheritance of classes
  3. The ability to use dynamic dispatch, for dynamically binding methods to objects
- Most types are usually classes
- These classes are abstract datatypes, because they show the world just their interface, but they also have their implementation there too.
- These classes can be made generic too (support for parametric polymorphism)
- The functions in a class will rely on variance (covariance, contravariance, and invariance) to choose the functions to support.

13.2

Concept: dynamic dispatch, also known as dynamic binding of methods

- This allows objects to use functions in both their instantiation class and from their parent classes.
- The correct one will be chosen by passed Actual Arguments, and which function will be found first.
- The search will always start in the object's instantiating class.

13.3

Concept: inheritance

- Classes can inherit data and functions from a parent class.
- This is quite similar to subtyping of normal variable types.
- However, this allows for functions to be passed down too.
- Some OOP languages support multiple inheritance, where a class can inherit from more than one parent.
  - This leads to the diamond problem.
  - One class has 2 parent classes that share the same parent, thus forming a diamond in the inheritance graph.
- An OOP language might or might not have a "top-level" class that acts as the parent to every subsequent class.

13.4

Concept: method overriding

- If a subclass requires a modified version of some function from the parent class, the subclass can redefine the function for itself.
- Then, if the subclass instantiates some data, it will first look for functions in the subclass's definition, before moving up to the parent(s).
- The correct function will be called by an object using dynamic dispatch.

13.5

Concept: the combined use of static types and dynamic types in statically typed object-oriented languages

- This happens when you write a variable will be of a superclass's type, but you instantiate it with a different type.
- Then, the static type will be the superclass's type, but the dynamic type will be the instantiating class's type.

14 Functional Programming

14.1

Concept: local let bindings of variables

- This allows declarations to occur between the `let` and `in` keywords.
- These definitions/declarations will shadow any that exist outside of the `let`-expression.
- Between the `in` and `end` keywords are expressions.
- Any expression that uses a declaration from inside the first portion will use that value.
- After the end of the `let`-expression, any information from the `let` portion of the `let`-expression is discarded.
- The `Let`-expression returns an expression, which may be used inside another `let`-expression, or used in another expression of any other type.

14.2

Concept: anonymous functions, also known as lambda expressions

- Since functions are expressions that map input types to output types, they don't necessarily need names.
- If a function does not have a name, they are anonymous functions, or lambda expressions.
- They **can** take parameters and use them.
- They are subject to the same type checking that all other functions get.

14.3

Concept: first-class functions

- Functions are treated just like data.
- Meaning functions can be assigned to variables (every function is anonymous until it is given a name to bind to).
- Functions can be passed into other functions, and returned from other functions, just like traditional data.

14.4

Concepts: pattern matching, including wildcards

- This is closely related to algebraic data types.
- You can pattern match on tuples too.

14.5

Concepts: exhaustiveness and redundancy in pattern matching

- Exhaustiveness
  - The pattern match must cover all cases to be exhaustive.
  - There is a wildcard operator _ that matches anything, but doesn't preserve its information, meaning you can't use it.
- Redundancy
  - When multiple patterns match the same thing.
  - Cannot have this, otherwise the pattern matching won't know which to choose.
  - It might choose the first pattern that it matches with instead.

14.6

Concept: lists as algebraic datatypes

- Think of these like BNF lists.
- There is either a "null", or a node with another list after it
- You can pattern match on this using `head::list` in SML.
- After this, the name `head` will refer to the first element in the original list.
- The `list` name will refer to the rest of the list after `head`.

14.7

Concept: the `map` function

- In SML `fn :  ('a -> 'b) -> 'a list -> 'b list`
- This function takes a function, which performs some action on elements of a list, which returns another function that takes in a list and returns a new list where the function was applied to all element of the original list.

14.8

Concept: currying

- Like in the `map` function, the act of taking in parameters that will return functions.
- This allows for great generalization in the language.
- However, the tradeoff to make is the programmer must determine the first things to be evaluated to create the function.
- If the first things to evaluate are chosen correctly, large performance benefits can ensue.
- It also allows for functions to be created more generally and slowly become more specfic, as needed.

For example, `map'` is a non-curried version. `map'` takes in a function and a list **AT THE SAME TIME**! `map` is a curried version. `map` takes in a function, and returns a function that takes in a list and returns a list.

```
fun map' (f, nil) = nil
  | map' (f, h::t) = (f h) :: map' (f,t);
(* val map' = fn : ('a -> 'b) * 'a list -> 'b list *)

fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t);
(* val myMap = fn : ('a -> 'b) -> 'a list -> 'b list *)
```

14.9

Functional Programming in Standard ML

15 Exceptions and Continuations

1. Concepts: exceptions and exception handlers