Programming Paradigms

David Ng

June 30, 2017

Contents

§1 May 17, 2017

§1.1 Programming Paradigm

A **programming paradigm** is a style or way of programming. It influences how you solve problems and implement solutions. Languages often include features from one or more paradigms.

- **Imperative Programming**: The program consists of a sequence of commands. These commands describe how the computation takes place, such as through step by step or modifying state. Most modern imperative languages are structured. They contain conditionals, loops, and subroutines (functions) instead of gotos. Variables are generally local to a block or subroutine. Some examples include C, Python, Pascal, Fortran, and ALGOL.

- **Object Oriented Programming**: The program describes objects and the ways they interact. It emphasizes objects instead of actions, and data rather than logic. The principal concern is sending messages to objects. Objects respond to messages by performing operations. Messages can have arguments, which may be similar to calling a subroutine. Similar objects are abstracted into a class, and similar classes are grouped together into inheritance hierarchies. They often include mechanisms for encapsulation and information hiding. Some examples include C++, Java, C#, Smalltalk, and Python.

- **Functional Programming**: The focus is on functions and the application of functions to values. It is concerned with the evaluation of expressions rather than running commands. Functions can be constructed as a program runs, passed as parameters to other functions, or returned as results from functions. Pure functional languages do not include loops, functions are side effect free, and there is no (modifiable) global state. Some examples include Haskell, ML, Lisp, and Miranda.

- **Logic Programming**: The program is a collection of facts and inference rules. In a sense, it is almost the complete opposite of imperative programming, which works to the goal by specifying the procedure. Logic programming identifies a clear goal to be found. This is accomplished through unification and backtracking, as it finds a value that satisfies the goal. This occurs automatically, as the programmer does not specify how to find the solution. Some examples include Prolog and Gödel.

§1.2 Languages

In written human language, letters are used to form words, words are used to form phrases, phrases are used to form sentences, sentences are used to form paragraphs, paragraphs are used to form sections, sections are used to form chapters, and chapters are used to form books. However, not every sequence of letters is an "allowed" word, etc. Syntax (grammar) defines the structure of what is allowed. Humans attach meanings to words. The semantics concern the branch of linguistics and logic regarding meaning. However, we note that there may be many ambiguities in a language.

In programming languages, syntax is the rules that describe valid combinations of symbols and the structure of a program, while semantics is the rules that describe the meaning of the statements. In programming languages, we need to communicate instructions to the machine. The language needs to minimize ambiguity. A language may be defined by specification (such as C++ and Java) or by implementation (such as Perl).

Almost all programming languages are Turing complete. Thus, with enough time and memory, they can theoretically compute anything that is computable. Programming languages evolve as they learn from predecessors. New problems may be easier to solve with new languages, and different tools are used for different jobs. There are tradeoffs between safety, speed and convenience between programming languages. Different people have different tastes and abilities, so a variety of programming languages is beneficial!

§1.3 Haskell

Haskell is a functional programming language that was originally developed in the 1990s, considered a successor to Miranda. It is general purpose and "pure". It offers several interesting features, such as a strong type system with type inference and type classes, lazy evaluation, pattern matching, and list comprehensions. Haskell can be interpreted or compiled. It is popular in academia, and has seen some industrial use.

To define a function in Haskell, we require its name and type, including the type of parameters passed to the function, and the type of values returned by the function. Syntactic requirements require that function names start with lowercase letters, types start with uppercase letters, and function names separated from its type by `::`. To implement a function, the name is written, followed by the arguments. This is separated from the function body by `=`.

§1.4 Decision Structures

Most programming languages provide multiple decision making constructs. In imperative and object oriented languages, we have if, if-else, if-elif-else, switch/case, and conditional operators. In these languages, a switch statement works much faster than an equivalent if-else ladder. This is because the compiler generates a jump table for a switch during compilation. Consequently, during execution, instead of checking which case is satisfied, it only decides which case has to be executed. However, fall through may be a source of bugs even for experienced developers. Additionally, a switch only works for a limited number of types.

In Haskell, we have the following decision making structures:

- **Pattern Matching** permits multiple definitions for the same function. Patterns describe combinations of values that can be passed to the function. Each pattern is checked in sequence, with the first matching pattern executing. Patterns can include a mixture of formal parameter variables and values. Pattern matching works well for a small number of special cases. However, it is unable to handle two large collections of values like all positive integers and all negative integers. Pattern order matters, so more specific patterns must be listed first. Patterns can be applied to any data type. By convention, _ is used for formal parameter variables that are not used in the function body.

- **Guards** permit a function's behavior to vary based on arbitrary expressions. They often contain relational operators such as $<, \leq, >, \geq, =, \neq$. Pattern matching and guards can be combined, as the guards apply to the pattern that immediately precede them. If none of the guards for a pattern evaluate to true, Haskell will move on to the next pattern. This could be used in an alternative implementation of the comparison function. Guards are always evaluated in order, and only the statement associated with the first guard that evaluates to true is executed.

- **If-Then-Else** can be used within another expression. This is analogous to the ternary conditional.

- **Case** is pattern matching on an arbitrary expression.

§2 May 19, 2017

§2.1 Recursion

Recursion is of immense importance in functional languages. This is because familiar iteration constructs are not typically provided. Recursion is the general concept of defining something in terms of itself:

- **Recursive Function** are functions that calls themselves.
- **Primitive Recursion** is a recursive definition where the problem n refers only to problem $n-1$.
- **General Recursion** is a recursive definition where problem n refers to problems other than $n-1$. This may refer to multiple problem sizes.
- **Mutual Recursion** refers to two or more functions which call each other.

A well formed recursive function normally includes one or more recursive cases. The function calls itself, and the recursive call is normally to a smaller or simpler version of the problem. The recursive function should also include one or more bases cases, where the result of the function is determined without calling itself.

Example 2.1

We can recursively define functions for factorial, GCD, and Fibonacci numbers. The naive recursive fibonacci function runs in $O(2^n)$ time, while an efficient iterative function (which can be rewritten recursively) runs in $O(n)$ time. Furthermore, there exists an explicit formulation of Fibonacci numbers that permits the calculation in $O(1)$ time.

To solve a problem recursively, we need to first identify a small version of the problem that we already know the solution to, or can work out easily. We then identify how a larger version of the problem can be solved by using a smaller version of the problem and some additional work. Recursion is of immense importance in functional languages.

§3 May 24, 2017

§3.1 Tuples

Tuples combine several pieces of data into one object. The pieces may have different types. This is similar to structs in C and C++, or data members in a Java or C++ class. This is also similar to records in Pascal, and tuples in Python.

Example 3.1

Tuples can contain different types. Some examples of tuple types can be `(Char, Int)`, `(Int, Int, Int, Int)`, and `(String, Float)`. The size and types do not change over time. Some examples of tuple literals can be `('a', 97)`, `(1, 2, 3, 4)`, and `(‘‘pi", 3.14159)`. Tuples with exactly two elements are referred to as **pairs**. `fst ('a', 97)` returns `'a'`, while `snd ('a', 97)` returns `97`. A tuple can represent a fixed length vector. For example, `(1, 2, 3)` and `(-1, 0, 0)`.

§3.2 Lists

Lists combine several pieces of data into one object. Every piece of data must have the same type, but the number of elements is flexible. This is similar to arrays in C, C++, and Java, and lists in Python. The elements in arrays are contiguous in memory, while in Haskell, lists are implemented as linked lists where memory is not necessarily contiguous. However, due to the linked list implementation, we obtain constant time insert at the front of the list. This also means that we have linear time access to subsequent elements in a list. The lists in Haskell are purely functional (they cannot be changed), and allow for lazy evaluation.

Example 3.2

Examples of list types can be `[Int]`, `[Char]`, and `[(String, Float)]`. Note that a character list is equivalent to a String in Haskell. Examples of list literals can be `[1, 2, 3, 4]`, `['a', 's', 'd', 'f'] = "asdf"`, `[("pi", 3.1415), ("e", 2.71)]`, and `[]`.

There are many functions that can be performed on lists:

- `head` returns the first element in the list. For example, `head [1, 2, 3, 4] = 1`. We cannot apply head to an empty list, since it will generate an exception.
- `tail` returns a list consisting of all of the elements except for the first element in the list. For example, `tail [1, 2, 3, 4] = [2, 3, 4]`. We also cannot apply tail to an empty list, since it will generate an exception.
- `take` returns a list of the first n elements from another list. For example, `take 3 [1, 2, ..., 10] = [1, 2, 3]`.
- `drop` returns a list of the elements from another list after the first n elements have been dropped. For example, `drop 3 [1, 2, ..., 10] = [4, 5, 6, 7, 8, 9, 10]`.
- `length` returns the number of elements in a list. The length of the empty list returns 0.
- `:` allows for insertion at the front of a list. For example, `2 : [4, 6, 4]` returns `[2, 4, 6, 4]`.
- `++` allows for the concatenation of lists. For example, `[1, 2] ++ [3, 4] = [1, 2, 3, 4]`.
- `!!` allows for an element from the list to be extracted by position. For example, `"qwerty" !! 0` returns `'q'`. We note that it is linear time to use this operation.

Functions that operate on lists often include recursion. This involves performing some task involving the first element in the list, and calling the function recursively on the rest of the list.

Example 3.3 (Run Length Encoding)

Run length encoding is a basic compression technique. It works well when there are long sequences of repeated values. It was used by the PCX image format. The decode operation can be used to convert a list of pairs into a string. The first element in each pair is a character, while the second element in the pair specifies the number of times it occurs.

Sometimes, it is inconvenient to write out all of the elements in a list. The `..` operation removes the need to do so in some situations. The simplest case occurs when each element is the successor of its predecessor. Different increment values are also possible.

Example 3.4

For example, `[1 .. 10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`, and `[1, 3 .. 10] = [1, 3, 5, 7, 9]`. Haskell assumes that we are incrementing unless we show another value that we are decrementing. For example, `[5 .. 1] = []`, while `[5, 4 .. 1] = [5, 4, 3, 2, 1]`.

§3.3 Repetition Structures

Many repetition structures exist in imperative and object oriented languages. While loops, do-while loops, repeat until loops, and if-goto loops are **condition controlled loops**. The for loop is a **count controlled loop**, and the for each loop is a **collection controlled loop**. To repeat a task in Haskell, we use recursion, list comprehensions, and higher order functions.

List comprehensions allows us to construct a new list from an existing list. Elements in the new list are the result of a calculation performed on the elements in the existing list. Elements can be removed (filtered) from the old list. They take the general form of:

```
[calculation involving name | name <- old list, condition 1, ...].
```

Only when all of the conditions are satisfied is the element included in the new list. List comprehension can also be performed on multiple lists.

Elements in a list can be tuples. For example, `[(1, 2, 0), (3, 0, 4), (5, 7, 6)]`. A list of tuples can be constructed by zipping two lists together. This process can be reversed with unzip.

Example 3.5

`zip [1, 2, 3] [2, 5, 2]` results in `[(1, 2), (2, 5), (3,2)]`. `unzip [(1, 2), (2, 5), (3,2)]` results in `([1, 2, 3] [2, 5, 2])`.

Lists and tuples make it practical to work with larger amounts of data. Tuples are similar to a C struct or data members in a class, while lists are an ordered collection of values that all have the same type. List and tuple elements are accessed via pattern matching and functions. Elements in a list can be processed using either recursion or list comprehensions.

§4 May 26, 2017

§4.1 Higher Order Functions

We can visualize functions as a box. It takes some inputs to produce an output. Inputs in Haskell can be base types, tuples, lists, or other functions, and output can also be a base type, tuple, list, or function. In functional languages, the inputs and output can also be functions. A function can take a function as a parameter, and can return a function as its result. **Higher order functions** are functions that take one or more functions as a parameter and/or returns a function as its result.

A function that computes a new list by applying a function to every element in a list is traditionally called `map`, and a function that retains or discards items in a list is traditionally called `filter`.

Example 4.1

To remove all of the odd numbers from a list, we can perform `filter isEven [list]`. To square all the even integers after discarding the odd integers, we perform `map square (filter isEven [list])`.

Polymorphic functions are functions that can be applied to multiple types of data. Many functions in the Prelude module are polymorphic. We can write our own polymorphic functions. Ideally, we want one version of map that works for all data types, and one version of filter that works for all data types. In Haskell, Polymorphic functions include one or more type variables in their type definition. By convention, the type variables are lower case letters starting from 'a'. A function's type can include a mixture of base/list/tuple types and type variables. In a polymorphic map, the input list and result do not need to be the same type. The type of the function passed to the map is therefore of the form `(a -> b)`. In a polymorphic filter, the input list and output list do need to be the same type. The type of function passed to the filter is therefore of the form `(a -> Bool)`.

Folding reduces a list to a single value by repeatedly applying a function to the value computed so far and the next item in the list. Folding can occur from the left or the right, and can be viewed as inserting an operator (function) between each value in the list. This operation is also commonly referred to as reduce.

Example 4.2 (Folding)

Left folding `+` into the list `[1 .. 5]` gives `((((1 + 2) + 3) + 4) + 5)`. Right folding `min` into the list `[1 .. 5]` gives `(1 min (2 min (3 min (4 min 5))))`.

When folding an empty list, we need to report an error. When folding a list with only one item, we simply return the value. These functions that we just described are referred to as `foldl1` and `foldr1`. They only work on lists that contain at least one element. More general functions `foldl` and `foldr` are also available. Their types are of the form `foldl :: (b -> a -> b) -> b -> [a] -> b` and `foldr :: (a -> b -> b) -> b -> [a] -> b`. These more general function can be safely applied to an empty list, where the identity element is returned.

§5 May 31, 2017

§5.1 Other Higher Order Functions

Recall that `zip` combines elements from two lists, forming pairs. On the other hand, `zipWith` combines elements from two lists, performing an arbitrary operation where `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`.

Example 5.1

We can sum pair of integers using `zipWith plus [1, 2, 3] [4, 5, 6] = [1 + 4, 2 + 5, 3 + 6]`. We can also use zip to implement the dot product as `foldl plus (zipWith multiply [1, 2, 3] [4, 5, 6])`

Sorting can be accomplished with `sortBy`, which is of the form `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. Ordering is a Haskell type with the three possible values of `LT`, `GT`, and `EQ`.

Polymorphic higher order functions can be used on lists of any type. Each function takes a parameter that is a function. Examples include `map`, `filter`, `foldr`, `foldl`, `zipWith`, and `sortBy`. This makes it possible to perform complex tasks without writing very much code.

§5.2 Lambda Functions

Small (even tiny) helper functions are often needed when working with higher order functions. Writing named helper functions can become tedious. **Lambda functions** provide another mechanism for writing these functions. Lambda functions are anonymous since they do not get a name, and are defined inline. The backslash \ is used to represent lambda. This is immediately followed by the function's parameters while `->` separates the parameters from the function's body.

Example 5.2

A function that sums two numbers is given by `\x y -> x + y`, and a function that converts a string to uppercase is given by `\s ->[toUpper ch | ch <- s]`. Thus, combining higher order functions and lambda functions, we can write expressions that remove all odd numbers from a list, `filter (\x -> x ‘rem‘ 2 == 0) [1..10]`, and expressions that convert a list of temperatures in Celsius to a list of temperatures in Fahrenheit `map (\t -> 9/5*t + 32) [-40, -30, 10]`

The value returned by a function can be a lambda expression. For instance, we can create a function that takes a two parameter functions as its only parameters and returns a new two parameter function as its result. The new function will perform the same task, but its arguments will be in the opposite order. This operation is commonly referred to as `flip`.

The concepts introduced here are not unique to Haskell. Some other languages allow functions to operate on many different types of values. They may permit a function to be passed to another function or returned as a function's result. Other languages may provide map, filter and reduce operations. Higher order functions take functions as parameters, or return them as the function's result. Polymorphic functions can operate on different types of data, and polymorphic higher order functions are powerful tools for manipulating data, often with a small amount of code. Lambda functions reduce the need to write separate named helper functions.

§5.3 Partial Application

The parameters to a function can be viewed in two different ways:

1. As a single combined unit: All values are passed as one tuple. This is how we typically think about parameter passing in Java, C++, Python, etc. This is referred to as an **uncurried form**. In Haskell, the uncurried form is used by placing round brackets around the parameters. This is syntactically similar to familar languages. Uncurried form may be useful in some situations, such as for mapping or filtering a zipped list.

2. As a sequence of values that are passed one at a time: As each value is passed, a new function is formed that requires one fewer parameters than its predecessor. This is how parameters are passed in Haskell. It is not a detail that we need to concentrate on, except when we want to make use of it. This is referred to as a **curried form**. Curried form permits partial application. This is the standard way to define functions in Haskell. A function of $n + 1$ arguments is an incremental extension of the function with n arguments.

These functions can be transformed from one form to the other. A curried function can be transformed into an uncurried function by calling the `uncurry` function. This takes one parameter, which is a curried function and returns one result, which is a function that performs the same task in uncurried form. An uncurried function can be transformed into a curried function by calling the `curry` function. This takes one parameter, which is an uncurried function, and returns one result, which is a function that performs the same task in curried form.

Example 5.3

We can compute the dot product in a single expression using `foldl1 plus (map (uncurry multiply) (zip [1, 2, 3] [4, 5, 6]))`.

The type of a function is given by its parameters and return type. When we pass in some number of parameters to the function, its type changes to be a function that requires that many fewer parameters.

Example 5.4

The type of `plus` is `Int -> Int -> Int`, the type of `plus 1` is `Int -> Int`, and the type of `plus 1 2` is `Int`.

Partial application allows us to create new functions. We set one or more parameters to a specific value, leaving a function that still requires at least one parameter. We then use the resulting function in any situation where a function is needed. For instance, we may pass it to `map`, `filter`, `foldl`, `foldr`, `zipWith`, etc.

§5.4 Operator Sections

An operator section is a partial application of an operator. This eliminates the need to write functions that duplicate existing operators.

Example 5.5

Using a map and an operator section to add 5 to every element in a list, we use `map (+ 5) [1 .. 10]`. To divide every element in a list by 5, we use `map (/5) [1 .. 10]`. Lastly, to compute 5 divided by every element in a list, we use `map (5/) [1 .. 10]`.

Forming a right section of `(-)` is problematic since `(-3)` is the integer negative 3, not a right section of the subtraction operator. To remedy this, we use `subtract` in the prelude module. It takes arguments in the opposite order you would intuitively expect. For instance, `subtract 4 2` is -2, not 2. Thus, `subtract 3` is equivalent to the right operator section of 3 and `(-)`. Alternatively, we could use `(flip (-) 3)`.

Example 5.6

To subtract 3 from a list, we can use `map (subtract 3) [1..10]` or `map (flip (-) 3) [1..10]`.

§6 June 2, 2017

§6.1 Function Composition

Two (or more) functions can be combined (composed) into a single function. Suppose we want to apply h to a parameter x, and then pass the result of that function application as a parameter to g, and then pass the result of that function application as a parameter to f, etc. Up until now, we would have evaluated this as $f(g(h(x)))$. Another option is now to use $(f \cdot g \cdot h)(x)$. The function passed to a higher-order function can be:

- A named helper function written by the programmer.
- A named function from a Haskell module.
- A lambda function.
- A partial application of function.
- An operator or operator section.
- A composition of any of the above.

§6.2 Type Classes

We recall that the function's type signature includes type variables, and the function can be applied to values of any type. However, there may be situations when we want to write a function that works only for some types.

Example 6.1

The `==` operator cannot be applied to functions. We need to restrict the types to only those that can be tested for equality.

Type classes allow restrictions to be placed on type variables. A function can be applied to many types, but not all types. Restrictions are specified as part of the function's signature. This is done using `=>`. Haskell provides numerous built-in type classes:

- Equality `Eq`. This includes operations such as `==` and `/=`.
- Ordering `Ord`. Operations include `<`, `<=`, `>=`, `max`, `min`, and `compare`.
- Enumeration `Enum`. Operations related to enumerations.
- Convert to String `Show`. The operation `show :: a -> String` and others.
- Numeric `Num`. Operations include addition, subtraction, multiplication, negation, absolute value, etc.

Type classes form a hierarchy. We can create our own type classes. This allows us to impose our own restrictions on the types that can be passed to a polymorphic function. We can also form a hierarchy among our own type classes. A type can be a specialization of another type.

§6.3 Algebraic Data Types

Some **base types** include `Int`, `Float`, `Char`, and `Bool`. Tuples and Lists are **composite types**. **Algebraic types** provide a powerful mechanism for defining our own types.

Enumerated types are an algebraic type that has a finite list of possible values. The values of an enumerated type are referred to as **constructors**. Type names must begin with an uppercase letter, and constructors must also begin with an uppercase letter. New types can be part of an existing type class. They can be derived from the default implementation, or we can provide its own implementation. Enumerated types can be passed as parameters to functions and returned as results.

Product types are an algebraic type that is similar to a tuple. One or more values are provided to the constructor when an instance of the type is created. Product types provide better type checking, better error messages, and better code readability. Tuples however, are beneficial for functions that cannot be applied directly to product types, and require less typing.

Example 6.2

We can represent information using a tuple. However, different information can be represented using tuples. Haskell cannot differentiate. Thus, one could erroneously enter data that should not be present.

Product types can have multiple constructors. Each constructor can take a different number of parameters.

Recursive algebraic types are a type defined in terms of itself.

Example 6.3

Nodes in a binary tree can be represented using recursive algebraic types. The base case would be an empty tree, while the recursive case would be a value, followed by a left and right tree.

Functions that operate on recursive data structures are often recursive. Tree traversal in pre order, in order, post order, and level order can be implemented. We can also write a function that maps a function over every node in a tree.

Example 6.4

We can write a function that determines the height of a binary tree, and a function that determines whether or not a value is present in the tree. The function for a general binary tree is different from the function for a binary search tree. We can also insert new values into a binary search tree, or construct a binary search tree from a list of values.

Polymorphic recursive types can be made to overcome the limitation of associating with a certain type when we make it polymorphic.

§7 June 7, 2017

§7.1 Lazy Evaluation

Lazy evaluation describes the phenomenon where objects and expressions are evaluated when they are needed. Expressions that are not used subsequently are not evaluated. Haskell is lazy. In Haskell, lazy evaluation applies to function parameters, list elements, guards, logical expressions, etc. With lazy evaluation, parameters are passed to the function without being evaluated. Parameters are evaluated in the called scope only as required. Thus, repeated uses of the same parameter in the called scope are only evaluated once. Composite parameters, such as lists and tuples, may only be partially evaluated.

Example 7.1

Consider a multiply function written with the optimizations of `multiply 0 _ = 0` followed by `multiply _ 0 = 0`. If we called `multiply 0 (naiveFib 40)`, it would evaluate almost instantly. However, if we called `multiply (naiveFib 40) 0`, it takes more time since pattern matching has to determine whether `naiveFib 40` evaluates to 0. In the event that we remove the first pattern, then `multiply (naiveFib 40) 0` would evaluate almost instantly, and `multiply 0 (naiveFib 40)` would take more time since there is no pattern.

In Java, C++, and Python, evaluation is almost always eager, since expressions are fully evaluated in the calling scope before they are passed to the function. In C++, the optimizer can (sometimes) avoid fully evaluating expressions that are never used subsequently. We can mark a function as a `const` function, and then turn on optimization with `-O` in `g++`. This is a rather different (and limited) form of lazy evaluation from what Haskell provides, but it provides some of the same benefits. The exception to eager evaluation in these languages is the built-in logical operators. This limited form of lazy evaluation is referred to as **short circuit evaluation**. If the value of the left operand conclusively determines the outcome of the logical expression, then the right operand is not evaluated (even if it has side effects).

Because of lazy evaluation, infinite data structures are possible in Haskell. Lazy evaluation permits infinitely large data structures, so long as they are never fully evaluated. In Haskell, evaluation is lazy rather than eager. Expressions are not evaluated until they are needed, and unused expressions are never evaluated. Lazy evaluation also permits the creation of (but not complete evaluation of) infinite data structures.

§8 June 9, 2017

§8.1 Type Checking

There are (at least) three different ways that we can compare type systems. Static Typing vs. Dynamic Typing, Strong Typing vs. Weak Typing, and Explicit Typing vs. Implicit Typing.

- In **static typing**, the type of a variable is known at compile time. It is explicitly stated by the programmer, or it can be inferred from context. Type checking can be performed without running the program. This is used by Haskell, C, C++, C#, Java, Fortran, Pascal, etc.
- On the other hand, **dynamic typing** occurs when the type of a variable is not known until the program is running. This is typically determined from the value that is actually stored in it. The program can crash at runtime due to incorrect types. However, dynamic typing permits **downcasting**, dynamic dispatch, and reflection. It is used by Python, Javascript, PHP, Lisp, Ruby, etc.

Thus, type checking at compile time before the program runs is static, while type checking during execution is dynamic. Type errors are caught earlier in static typing. All type errors that can be found by the system will be found, and there may be some performance benefits. Dynamic typing on the other hand, provides greater flexibility and may permit faster development.

Example 8.1

Some languages do a mixture of both of static and dynamic typing. For instance, most type checking in Java is performed at compile time, but downcasting is permitted, and cannot be verified at compile time. The generated code includes a runtime type check, where a `ClassCastException` is thrown if the downcast was not valid. Most type checking in C++ is performed at compile time, but downcasting is permitted, and cannot be verified at compile time. A C-style cast on incompatible pointer types results in a C++ `reinterpret_cast` and often causes unpredictable behaviour for incompatible types. Using `dynamic_cast` in C++ returns `NULL` for an invalid cast.

- In a **strongly typed** language, one must provide a value that matches the expected type. This is an exact match, subtype match, type class match, or "duck" match.
- In a **weakly typed** language, one has much more freedom to mix values of different types. Type strength is a continuum, and is independent of when the type checking is performed. Strong and weak typing indicates how much type checking is done.

Example 8.2 (Adding a Character to an Integer)

This is prohibited in Haskell, prohibited in Python, permitted in Java only if the result is stored in an `int`, permitted in C++, permitted in C, and permitted in Perl (with a warning).

Example 8.3 (Adding a String to a String)

This is prohibited in Haskell with `+` but permitted with `++`, permitted in Python, permitted in Java, permitted in C++, and prohibited in C with `+` but permitted with `strcat` or `strncat`.

Example 8.4 (Adding an Integer to a Float)

This is permitted in some form in almost all other languages, but it is prohibited in Haskell. We start encountering errors when we have used up all 32 bits.

- **Explicit typing** occurs when the programmer is responsible for specifying types while writing their code. Examples include C and Java.
- **Implicit typing** occurs when types are inferred by the type system, or simply are not needed. Examples include Python and Perl.

Explicit and implicit typing concerns whether the programmer specifies the types directly, or whether that is left for the compiler or interpreter to infer. Haskell allows the programmer to explicitly specify types, but explicit types are rarely required. Haskell will infer the types (in most situations) when they are not provided. C++ 11 added the auto keyword. The variables still need to be declared, but the type is automatically inferred by the compiler. This is really convenient for complicated template types in C++.

§8.2 Type Checking in Haskell

In Haskell, an expression can include literals, operators, function calls, and parentheses. In Haskell, the type of every expression can be determined at compile time, and is thus checked for type correctness.

Example 8.5

We can determine whether an expression is type correct. We know from previous type signatures that `chr :: Char -> Int` and `ord :: Int -> Char`. Haskell does type checking before anything is run. Each subexpression needs to fit the corresponding types specified by functions.

Example 8.6

We can determine whether a guard is type correct. Pattern matching requires each of the patterns p_i match the corresponding type t_i. The guards g_i must all be boolean expressions, and the returned values e_i in each case must be of the type specified by the function header.

§8.3 Polymorphic Type Checking

Some expressions cannot be reduced to a single type. The expression's result could be one of several (or many) types:

- **Monomorphic type checking**: Each expression can be reduced to a single type, or declared type incorrect.

- **Polymorphic type checking**: Each expression is reduced to a set of possible types, or declared type incorrect.

Example 8.7

When we use the `..` operator, the beginning and end arguments must be the same type. We can also make deductions about the function's type signature based on values for which we know the type inside the function through their use. We may also deduce the types of values based on our knowledge of the input and output of predefined functions.

Unification identifies the set of types that satisfy two (or more) types. For instance, `(a, [Char])` can represent `(Int, [Char])`, `(String, [Char])`, `([Int], [Char])`, etc. On the other hand, `(Int, [b])` can represent `(Int, [Char])`, `(Int, [Int])`, `(Int, [[Char]])`, etc. The unification of `(a, [Char])` and `(Int, [b])` results in the ability to distinguish one possibility from an infinite set. In this case, the only result is `(Int, [Char])`.

Example 8.8

Unifying the types `(a, [a])` and `([b], c)`, we obtain the unified type of `([b], [[b]])`. Unifying the types `[Int] -> [Int]` and `a -> [a]` results in an error since we have a type incorrect statement.

The same literal can appear multiple times in an expression. It does not necessarily have the same type each time it occurs. This results from using polymorphic literals and functions. In the expression `zip ([] ++ [1]) (['a','b'] ++ [])`, the empty list takes on different types, as it is used as a numeric list and as a character list. Similarly, polymorphic functions can have a different type every time that they occur.

Type classes must also be considered when performing type checking. Type classes are attached to type variables as necessary. Unification then ensures that the types identified are instances of the required classes. Type class requirements are simplified where possible. For instance, a type variable that includes requirements `Eq a` and `Ord a` is reduced to `Ord a` because `Ord` extends `Eq`.

Our goal is to generate correct code. Incorrect code should either not compile, or crash when it is executed. Incorrect code that runs and does not crash is a nightmare, especially as the project gets larger.

§9 June 14, 2017

§9.1 Prolog

Prolog is a logic programming language that expresses the logic of the computation without describing its control flow. To create a Prolog program, we write facts (statements that are unconditionally true) and rules involving those facts (statements that are conditionally true given that something else is). It is important to note that the programmer does not specify the control flow for the program. To run a Prolog program, we write a query (or goal) and let Prolog's engine search for an answer. **Search processes similar to those used by Prolog are an important part of AI**.

We start a Prolog knowledge base by constructing functors that include the facts we want. To load the knowledge base into Prolog, we choose `consult` from the file menu.

We can then start querying facts. When a `?` appears, we can press enter to accept the answer, or press `;` to see if there are any more answers that match the variable. Alternatively, we can type `a` to see all of the answers.

§9.2 Rules, Variables, and Conditions

Rules allow the knowledge base to contain conditional knowledge. Something is true if something else is true. **Modus Ponens** is a rule of logical inference that is used repeatedly in Prolog to answer queries. Given a rule that states a is true if b is true, and the knowledge that b is true, modus ponens allows us to conclude that a is true as well.

Example 9.1
We can add rules that state that a food is healthy if it is a fruit. We can add additional rules that state that a food is healthy if it is a vegetable.

Variables are placeholders for information that will be determined in the future. Any name that starts with an uppercase letter is a variable

Example 9.2
We can write Prolog goals that answers what items Alice has, who has pears, which items are vegetables, which items are healthy, etc.

A variable can appear more than once in a Prolog goal. For instance, we can create a query that determines who has both apples and plums. Variables can appear in Prolog rules. Instead of listing individual instances, we can write our rules to incorporate variables.

Example 9.3
We can rewrite our rule for `healthy()` so that it does not have to list off every healthy item. All fruits are healthy and all vegetables are healthy.

Conjunction (And) means that both items must be true for the result to be true. It is denoted by a `,` in Prolog. **Disjunction** (Or) means that at least one item must be true for the result to be true. It is denoted by a `;` in Prolog, or by having two separate rules.

§9.3 Terms

Prolog knowledge bases are constructed from three types of terms::

- **Constants**: These must be a number or an atom. Atoms begin with a lowercase letter, look like an operator (a collection of symbols), or are quoted inside single quotes. Examples include `1`, `31`, `lisa`, `james`, `+`, `:?`, and `'Apple'`.
- **Variables**: These must begin with an uppercase letter or an underscore. Examples include `X`, `_123`, and `ListOfValues`.
- **Compound Terms**: These consist of multiple terms in the form `f(t1, t2, ..., tn)`. The atom `f` is referred to as a **functor**, while the terms `t1 ... tn` are referred to as **components**. The number of terms inside the parentheses is the **arity** of the term `f`.

§9.4 Matching

Matching is performed between the goal and the knowledge base. Prolog uses matching to determine whether or not a goal is true, and whether there are values for variables that satisfy the goal. `A \= B` is equivalent to `not (A = B)`. Two terms t_1 and t_2 match if

- t_1 and t_2 are both the same constant term.
- t_1 is an uninstantiated variable and t_2 is any other term. In this case, t_1 is instantiated to t_2.
- t_2 is an uninstantiated variable and t_1 is any other term. In this case, t_2 is instantiated to t_1.
- t_1 and t_2 are both uninstantiated variables. In this case, t_1 and t_2 now co-refer.
- t_1 and t_2 are both compound terms with the same functor name and arity. Additionally, all corresponding terms inside parentheses match, and all variable instantiations within the parentheses are compatible.
- No other terms match.

Example 9.4 (Family Tree)
We can create a knowledge base that describes a family. Facts are written to identify each person as male or female, and identify each parent and child relationship. Rules are used to describe more complex relationships such as sibling, brother, sister, aunt, uncle, grandfather, grandmother, cousin, etc. We can then form goals or queries that test if two people have a certain relationship, and identify everyone who is related to a certain person in a given way.

§10 June 16, 2017

§10.1 Backtracking

In order for a Prolog program to report the correct result, it must be **logically correct** and **procedurally correct**. A program can be logically correct while being procedurally incorrect. Such a program will not return a correct result. In theory, the order in which clauses in a conjunction or disjunction are evaluated does not matter. In practice however, Prolog evaluates clauses in a defined order. This impacts how quickly an answer is found, or even if it is ever found.

Prolog searches the knowledge base from top to bottom and left to right within each line. Once a matching line is found, and the left-most rule is satisfied, then the next rule on the line is attempted. If a term fails, **backtracking** is performed. The search moves back one level and tries another value. If all of the values have been tried, we move back another level. This is performed until all combinations have been tested.

Example 10.1

When we query a result with a conjunction between two expressions, Prolog first matches the first expression with a rule. It continues searching the rest of the rules in the knowledge base to see if the second expression can be satisfied. If it cannot, it matches the first expression with another rule and tries again. Prolog does not prompt for another result if the only result it has found requires the last line in the knowledge base.

Remark 10.2. Prolog includes a debugger that allows one to see how it is running. It is enabled with `trace.` and disabled with `notrace.`. Within the debugger, `<enter>` creeps through each step, `s` skips to the answer for the current call, `l` leaps to the next answer, and `a` aborts debugging of the query.

Prolog uses brute force to check all possibilities. Facts and rules are processed from top to bottom, while clauses within a rule are satisfied from left to right. When a failure occurs, backtracking undoes the most recent match and choses another value, or moves back one level if there are no further values. This process repeats until an answer is found, or all possibilities have been exhausted. The order of facts, rules and clauses within a rule determine how quickly an answer is found (or if it is ever found) and the order in which the answers are found.

§10.2 Recursion

Recursion is critically important in logic programming. Recursive rules use the same functor on both sides of the `:-` symbol. Recursive rules must have a base case that is either a version of the rule that is not recursive (explicit), or a smaller version of the problem that results in failure (implicit).

Example 10.3

In a family tree knowledge base, we can define the grandparent of A to be B, who is the parent of the parent of A. The great grandparent of A is B, who is the parent of the parent of the parent of A. We can define the ancestor B of A as stating that A has a parent X, and X has an ancestor B. The base case can be listed on another line, where B is the parent of A.

The order in which we place the base case and the recursive case in relation to each does not change the set of results. However, it changes the order in which results are returned. It is important to ensure that the recursive call is done with at least one instantiated variable. That is, if we make the recursive call first in the recursive case, this may result in an infinite loop. As a general rule, we should delay the recursive call.

Example 10.4

Consider a transportation company that has routes between cities. We can create a Prolog knowledge base that represents these routes. We can then create a Prolog rule that allows us to identify a route from any city to any other city. We are limited in that we cannot obtain the actual route, only the truth of whether there is a route. Our code would also produce an infinite list of results should there be a match, as the route from A to B can lead to a route from A to B to A to B, ad infinitum.

§10.3 Lists

Lists in Prolog are similar to Haskell, both structurally and syntactically. They are represented as a head element followed by the tail of the list. Elements are enclosed in square brackets, where `[]` represents the empty list. Lists can hold an arbitrary number of terms (including duplicates).

Example 10.5
`[1, 2, 3]` is a list of three integers, while `[H|T]` is equivalent to the Haskell expression of `(h:t)`.

Matches are performed on Prolog lists like other variables.

Example 10.6
Examples of matches are found below:

- `[1, 2, 3] = [A, B, C]`. This matches since `A = 1`, `B = 2`, and `C = 3`.
- `[1, 2, 3] = [A, B|C]`. This matches since `A = 1`, `B = 2`, and `C = [3]`.
- `[1, 2, 3] = [A, B, C|D]`. This matches since `A = 1`, `B = 2`, `C = 3`, and `D = []`.
- `[1, 2, 3] = [A|B]`. This matches since `A = 1`, and `B = [2, 3]`.
- `[1, 2, 3] = [A, B]`. This does not match, as `A = 1`, but there is no way to match 2 and 3 to `B`.
- `[A, 2] = [1, B]`. This matches since `A = 1`, and `B = 2`.
- `[A, 2] = [1, B|C]`. This matches since `A = 1`, `B = 2`, and `C = []`.
- `[A, B] = [1, C|D]`. This matches since `A = 1`, `B = C`, and `D = []`.

Prolog includes numerous built-in list predicates. The exact list of predicates varies from implementation to implementation. Some are more standard than others. For instance, gProlog includes `length`, `member`, `reverse`, `permutation`, `sublist`, and `maplist`. Predicates that operate on lists are often recursive

The `length/2` predicate (where `/2` indicates that it takes two parameters) determines how many atoms are in a list. The first parameter is a list, and the second parameter is a non-negative integer length. There are different uses:

- The first and second parameters are both instantiated. The predicate succeeds when the second parameter is the length of the list.
- The first parameter is a list and the second parameter is an uninstantiated variable. The variable is instantiated to an integer equal to the length of the list.
- The first parameter is an uninstantiated variable and the second parameter is an integer. The variable is instantiated to a list of the indicated length.
- The first and second parameters are both uninstantiated. The predicate returns lists of all possible lengths.

The `member/2` predicate determines whether or not a value is present in a list. The first parameter is a term, and the second parameter is a list of terms. There are different uses:

- The first parameter is a term and the second is a list. This succeeds if and only if the term is present in the list.
- The first parameter is an uninstantiated variable, and the second is a list. The variable is instantiated to each term in sequence (by backtracking).
- The first parameter is a term, and the second is an uninstantiated variable. The variable is instantiated with an infinite sequence of lists containing the value (by backtracking).
- The first and second parameters are both uninstantiated variables. The variable is instantiated with an infinite sequence of lists containing the uninstantiated variable (by backtracking).

Example 10.7
We can write a predicate that provides the same functionality as `member/2`. We can use a recursive solution.

To create bigger lists, one element `H`, can be added to the front of an existing list `T` with `NewList = [H | T]`. Multiple items `H1`, `H2`, and `H3` can be added to the front of `T` with `NewList = [H1, H2, H3 | T]`. Two lists can be concatenated with the `append/3` predicate. There are different uses:

- The first parameter is a list, the second parameter is a list, and the third parameter is an uninstantiated variable. The third parameter is an uninstantiated variable that is instantiated to a list containing all of the elements in first followed by all of the elements in second.
- One of the first two parameters is a list, the other is an uninstantiated variable, and the third parameter is a list. The uninstantiated variable is instantiated to a list containing the elements necessary to form the list provided as the third parameter.
- The first two parameters are uninstantiated variables, and the third parameter is a list. All combinations of lists that can be concatenated to form the third list are generated in sequence (by backtracking).

Example 10.8
We can write a predicate that provides the same functionality as `append/3`. We can use a recursive solution.

Some other useful list predicates are `permutation/2`, which determines whether one list is a permutation of another (can generate all permutations of a list by backtracking), and `sublist/2`, which determines whether the first list is a sublist of another (can generate all sublists of a list by backtracking).

Remark 10.9. A sublist in this context is a list such that all of its elements can be found in the second list in the same order, possibly with other elements in between them

§10.4 Negation

We may want to negate the outcome of a predicate. Specifically, we would like to test if an element is not present in a list. The `\+` predicate reverses the outcome of a predicate. Failure is treated as success.

Example 10.10

`member(1, [1, 2, 3])` returns true and `member(4, [1, 2, 3])` returns false. Thus, `\+member(1, [1, 2, 3])` returns false and `\+member(4, [1, 2, 3])` returns true.

§11 June 21, 2017

§11.1 Structures

So far, we have written numerous complex terms of the form

$$functor(t_1, t_2, ..., t_n),$$

that are used in facts and rules. Matching is determined by the truth of such facts and rules, with the program possibly instantiating variables in order to do so. Such complex terms can also be used as a simple data structure similar to a product type in Haskell, a record in C, or the data members of a Java class.

Using a complex term as a **structure**, the name of the functor is the name of the data structure. The terms enclosed inside parentheses are the values stored in the structure. Elements within the data structure are unnamed, as they must always occur in the same order. The elements are also untyped, as no type checking is performed on the elements. Such structures can be passed to other functors and stored into uninstantiated variables by matching.

Example 11.1

We can create structures that represent different animals. We can store its name, species, and weight. We can then create a rule that retrieves an animal's weight. To accomplish this, our first parameter is an animal structure, and our second parameter is the animals' weight. This second parameter is usually an uninstantiated variable.

Some problems are effectively modeled by state machines. Structures help us represent such problems in Prolog, since we can make use of a structure that holds information about the current state. Facts and rules describe legal transitions between states.

Given a list of statements, we would like to determine whether there is a list of steps from the start state that we could follow to arrive at a desired end state. We first need a way to represent this information. Together, this information comprises the current **state**. **Transitions** between states represent moves from one state to another. To represent valid transitions in Prolog, we describe our current state, a possible action, and the new state that results. This can be listed, with each component of state and action leading to some new state with new components.

§11.2 Cuts

Prolog is thorough, as It will search for an answer until all possible combinations are checked. Sometimes it is too thorough, and can be too time consuming. It may even find answers that we do not really want. **Cuts** prevent Prolog from backtracking, and is denoted by `!`. They can also be used to improve efficiency. There are three main uses of cut:

1. If Prolog has reached this point, it has found the only solution to the problem. This improves efficiency by preventing backtracking when there are no more solutions to find (or no more solutions that we want it to find). The cut appears as the last predicate in the body of the rule.

   Example 11.2
   Consider a knowledge base that contains facts that describe the colors of various items. We want to write a rule, `color(X, C)` that instantiates `C` with the color of `X` if `X` is present in the knowledge base, and instantiates `C` with `unknown` if `X` is not present.

2. If Prolog has reached this point, it has found the correct rule. Consider any rule with two (or more) mutually exclusive definitions. If the first definition succeeds the second one must fail. If the first definition fails, the second one must succeed. Using a cut will reduce the amount of work that Prolog must perform.

   Example 11.3
   We can create a Prolog rule that cuts after one definition. For instance, a certain set of numbers are among one's favorite. At the end of this definition, we include `!`. Thus, the second definition is used for all other cases.

3. If Prolog has reached this point there is no result that we want. This allows us to specify a list of conditions that describe failure. Cut appears as the second last clause in a conjunction. The built-in predicate `fail` appears as the last clause.

   Example 11.4
   We can create a Prolog rule about which foods can be taken to school. Nuts cannot be taken to school, junk food cannot be taken to school, and anything else can be taken to school.

Cuts allow us to limit Prolog's search. They introduce a point of no return within the current rule, thus committing Prolog to every choice it has made since (and including) selecting the current rule. When a cut is present in a rule, Prolog will never backtrack past it. Making a cut the last clause in a rule ensures that it will return at most one result, since no further rules are considered. Making a cut the first clause in a rule ensures that no further rules are considered, but the current rule may be matched multiple times.

§11.3 Mathematics

Prolog is primarily designed for symbolic computation, but mathematical calculations are also possible, and are needed to solve some problems. Numeric constants can be used directly, and functors are available to represent common mathematical operations. These can be used with either prefix or infix notation.

Mathematical functors do not evaluate expressions when they are used. Thus, `3 + 2` forms a structure named `+` with its first parameter set to 3 and its second parameter set to 2. The structure can be evaluated later. Recall that `=` is the matching operator. It does not force evaluation of mathematical functors, so `3 + 2 = 3 + 2` succeeds, but `3 + 2 = 2 + 3` fails. To force evaluation, we use the `is` operator. Thus, `M is 3 + 2 = 5`. The relational operators also force evaluation. `1 + 1 < 2 + 2` succeeds. `2 + 1 > 1 + 1` also succeeds. Some of the relational operators use different symbols from familiar languages. Less than or equal is `=<`, greater than or equal is `>=`, equality is `=:=`, and not equal is `=\=`.

§11.4 Different Notions of Equality

There are different notions of equality in Prolog:

- `=` is the matching operator where `A = B` succeeds if `A` matches `B`. One or both of `A` and `B` may be uninstantiated variables. Thus, `1 + 2 = 2 + 1` fails.
- `==` is the term identical operator where `A == B` succeeds if `A` and `B` are both the same term. It fails if `A` or `B` (or both) are uninstantiated variables. It does not evaluate mathematical functors, so `1 + 2 == 2 + 1` fails.
- `=:=` is the arithmetic comparison operator where `A =:= B` succeeds if `A` and `B` both evaluate to the same number. It throws an exception if `A` or `B` (or both) are uninstantiated variables or contain a non-numeric term. Thus, `1 + 2 =:= 2 + 1` succeeds.