CS 131

Warren Kim

May 3, 2023

Contents

1 Overview

1.1 What is a Programming Language?

A programming language is a structured system of communication designed to express computations in an abstract manner.

1.2 Why Different Languages?

Different languages are built for different use cases. Below are popular languages that were built for their respective use cases:

(i) Javascript is the most popular language for anything related to web development. There are many frameworks for vanilla Javascript (e.g. React) as well as derivative languages (e.g. Typescript).

(ii) C/++ is a popular language for programs that require high performance (e.g. Linux).

(iii) C# is most commonly used for programs that are in Microsoft's .NET ecosystem.

(iv) Python is a popular language used in the field of artificial intelligence.

(v) ba/z/sh is a scripting language for UNIX-based operating systems.

(vi) R is a popular language among statiticians (not sure why).

(vii) Lisp is a functional language used in the field of artificial intelligence and was used to write Emacs.

(viii) SQL and its variants are a set of querying languages used to communicate with databases.

1.3 Language Paradigms

There are four main language paradigms:

(i) Imperative

(ii) Object-Oriented

(iii) Functional

(iv) Logic

1.3.1 Imperative Paradigm

Imperative programs use a set of statements (e.g. control structures, mutable variables) that directly change the state of the program. More specifically, these statements are commands that control how the program behaves. Common examples of imperative languages include FORTRAN and C.

1.3.2 Object-Oriented Paradigm

The object-oriented paradigm is a type of imperative programming, and contains support for structured objects and classes that "talk" to each other via methods (e.g. `d` is a `Dog` object with the class method `bark()`, where `d.bark()` will invoke the `bark` function for the object `d`). Common examples of object-oriented languages include Java and C++.

1.3.3 Functional Paradigm

Functional programming is a type of declarative programming. They use expressions, functions, constants, and recursion to change the state of the program. There is no iteration or mutable variables. Common examples of functional languages include Haskell and Lua.

1.3.4 Logic Paradigm

Logic programming the most abstract and is a type of declarative programming. A set of facts and rules are defined within the scope of the program. Common examples of logic languages are Prolog and ASP.

1.4 Language Choices

There are many things to consider when building a programming language. Some of these include:

(i) Static/Dynamic type checking

(ii) Passing parameters by value/reference/pointer/object reference

(iii) Scoping semantics

(iv) Manual/Automatic memory management

(v) Implicit/Explicit variable declaration

(vi) Manual/Automatic bounds checking

Generally, a programming language can be broken down into its syntax and semantics.

2 Functional Programming (Haskell)

We will talk about functional programming as it pertains to *Haskell*, a purely functional language.

2.1 Overview

Haskell is a **statically typed** language that uses **type inference**. All functions must have the following properties:

(i) Functions must take in an argument

(ii) Functions must return a value

(iii) Be pure (does not change the state of the program (***Note:*** This includes I/O!)

(iv) In functions, all variables are immutable

(v) Functions are **first-class citizens**, so they are treated as data

2.2 Pure Functions

Given a fixed input x, it always returns the same output y. That is, it does not modify any data beyond initializing local variables. Some consequences of this are:

(i) Multithreading easy in functional languages since there are no race conditions (everything is immutable)

(ii) Execution order doesn't matter: Functions are pure, so there are no side effects. **Haskell** has lazy evalueation, so it will only execute what is referenced.

2.3 Syntax

Haskell syntax for defining a function is as follows:

```
function_name params = function_body
```

2.3.1 Indentation

In Haskell, any part of an expression must be indented further than the beginning of the function. e.g.

```
mult x y =
  x * y
```

2.4 Data Types

Since Haskell is statically typed and uses type inference, though the variables' types are figured out at compile time, we need not explicitly annotate them (though possible). The following are some of Haskell's primitives:

`Int` 64-bit signed integer

`Integer` Arbitrary-precision signed integer

`Bool` Boolean (True/False)

`Char` Characters

`Float` 32-bit (single-precision) floating point

`Double` 64-bit (double-precision) floating point

Syntax `variable_name = value :: type`

Note `:t variable_name` Returns the type of a variable

2.5 Operations

Arithmetic operations include `+, -, *, /, ^ ‘div‘, ‘mod‘`.

Note: Parentheses are required for the unary `-` (e.g. `(-3)` represents -3)

Note: Airthmetic operators can also be called using prefixed notation (e.g. `(+) a b` is equivalent to `a + b`)

2.6 Composite Data Types

Some of the common composite data types are:

`()` Tuples: A **fixed-size** collection of data (may be different types)

`[]` Lists: A collection of data of the **same type** (internally, they are structured like a linked list)

`[Char]` Strings: A list of characters

2.6.1 Tuples

Tuples have two built-in functions: `fst, snd` which retrieve the first and second elements respectively. Consequently, accessing any element after the second requires a user-defined function.

2.6.2 Lists

Lists are **not** arrays, and are structured internally like linked-lists. Therefore, most operations are `O(n)` in time complexity.

`head :: [a] -> a`

`head LIST` Returns the head of the list.

`tail :: [a] -> a`

`tail LIST` Returns the tail of the list.

`take :: Int -> [a] -> [a]`

`take n LIST` Returns the first `n` elements of the list.

`drop :: Int -> [a] -> [a]` t

`drop n LIST` Returns the last `(length LIST) - n` elements of the list.

`(!!) :: [a] -> Int -> a`

`LIST !! n` Returns the $\mathtt{n}^th$ item of the list.

`zip :: [a] -> [b] -> [(a, b)]`

`zip LIST_1 LIST_2` Returns a list of tuples of the form `(a`$_i$`, b`$_i$`)`.[1]

`length :: [a] -> Int`

`length n LIST` Returns the length of the list.

`elem :: a -> [a] -> Bool`

`elem ITEM LIST` Returns `True` if `ITEM` $\in$ `LIST`, `False` otherwise.

`sum :: [a] -> a`

`sum LIST` Returns the summation of all elements of then list.

`(++) :: [a] -> [a] -> [a]`

`LIST_1 ++ LIST_2` Concatenates two lists of the same type.

`(:) :: a -> [a] -> [a]`

`ITEM ++ LIST_2` Appends a single element to the front of the list.

2.6.3 Strings

Strings are just a list of characters. Therefore, we can concatenate strings and perform list operations:

```
str :: String
str = "some"

other_str :: String
other_str = " string"

combined_str :: String
combined_str = str ++ other_string
```

will return `"some string"`. Moreover,

`"same" == ['s', 'a', 'm', 'e']`

will return `True`

[1] `(a`$_i$`, b`$_i$`)` where `a`$_i$ $\in$ `LIST_1`, `b`$_i$ $\in$ `LIST_2`, $0 \leq$ `i` $\leq$ `(min (length LIST_1) (length LIST_2))`

2.7 List Processing

2.7.1 Creating Lists (Concatenation and Cons)

Lists can be concatenated using the `(++)` operator.