CS 131

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 and Operations

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

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`)

2.5 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

2.5.1 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.

2.5.2 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 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.

2.5.3 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))`

2.6 List Processing

2.6.1 Creating Lists (Concatenation and Cons)

Lists can be created directly or indirectly via concatenation or consing:

Direct Creation

```
some_list = [1, 2, 3]
```

Concatenation

```
some_list = [1, 2, 3]
new_list = some_list ++ [4, 5, 6]
```

Here, `new_list` is `[1, 2, 3, 4, 5, 6]`

Consing

```
some_list = [1, 2, 3]
new_list = 0 : some_list
```

Here, `new_list` is `[0, 1, 2, 3]`

2.6.2 Ranges

Ranges are inclusive from both sides. By default, they are incremented by 1.

Incremented by 1

```
one_to_ten = [1..10]
```

Here, `one_to_ten` is `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

Incremented by X (= 2)

```
odds = [1, 3..10]
```

Here, `odds` is `[1, 3, 5, 7, 9]`

Infinite Lists

```
infty = [1..]
```

Here, `infty` is `[1, 2, ...]`

Cyclic Lists

```
cyclic = cycle [1, 2, 3]
```

Here, `cyclic` is `[1, 2, 3, 1, 2, 3, ...]`

2.6.3 List Comprehensions

List comprehensions create lists from one (or more) existing lists. They have a similar syntax to set-builder notation in Mathematics. A list comprehension has the following:[2]

(i) Generators: One or more input list(s)

(ii) Guards: Filters or conditions on the generators

(iii) Transformation: Applied to inputs (that satisfy the guard clauses) before adding them to the output list

[2] See HW for more examples

Example with Numbers

```
squares = [x^2 | x <- [1..3]
```

Here, `squares` is `[1, 4, 9]`

Example with Strings

```
nouns = ["party","exam","studying"]
adjs = ["lit","brutal","chill"]
combos = [adj ++ " " ++ noun | adj <- adjs, noun <- nouns]
```

Here, `combos` is `["lit party", "brutal exam", "chill studying"]`

2.7 Control Flow

2.7.1 `if then else`

Haskell has standard `if then else` logic. However, note that in Haskell, an `if` must be closed with an `else`. The syntax is as follows:

```
some_function =
  if *condition* then
    *statements*
  else
    *statements*
```

2.7.2 Guard Clauses (|)

Guard clauses are similar to switch cases. Note that they don't require an `=` after the function name. Additionally, `otherwise` can be replaced with _. The syntax is as follows:

```
some_function
  | *condition* = *statements*
  | *other condition* = * statements*
  ...
  | otherwise = *statements*
```

2.8 Pattern Matching

In the following subsections, we discuss different ways to pattern match. Note that in all of them, we can replace a parameter with _, which indicates that we will not use it and therefore it does not matter.

2.8.1 Constants

To pattern match functions with constants, we define multiple versions of the same functions that all have the same number and types of arguments. Then, we define **unique** parameters. Note that **order matters**. Haskell will try to match top to bottom. The syntax is as follows:

```
multiply :: Int -> Int -> Int
multiply 10 b =
  10 * b
multiply a 10 =
  a * 10
multiply a b =
  a * b
```

2.8.2 Tuples

Pattern matching with tuples is similar, but we can decompose the tuple. The syntax is as follows:

```
exp :: (Int, Int) -> Int
exp (_, 0) = 1
exp (a, b) = a ^ b
```

2.8.3 Lists

Pattern matching lists are similar to pattern matching tuples. Note that we can decompose a list multiple different ways:

(i) `(x:xs)` where `x` is the head and `xs` is the rest of the list (excluding `x`).

(ii) `x:...:xs)` where `...` represents the second to $\texttt{i}^{th}$ element of a list.

(iii) `[a, b, c]` where `a, b, c` correspond to the first, second, and third elements of the list respectively.

2.9 Local Bindings (`let` and `where`)

To create temporary variables, we use `let` and `where`. Note that we can create (nested) helper functions using these keywords. The syntax is as follows:

2.9.1 `let`

```
some_function arg =
  let
    x = 10 + arg
    *statements*
  in
    if x == 10 then
      *something*
    else
      *something else*
```

2.9.2 `where`

```
some_function arg =
    if x == 10 then
      *something*
    else
      *something else*
  where
    x = 10 + arg
    *statements*
```

2.10 Type Annotations

Functions have optional type declarations that take the form
`name :: type -> ... -> return_type` where `type -> ... ->` are parameter types. Type variables are denoted by any lowercase variable name in place of `type` and represent any type.
Note: Type annotations of the form
`name :: type -> (type -> ... -> return_type) -> ... -> return_type` indicate that the function takes in another function `(type -> ... -> return_type)` as a parameter.

2.11 Functions

Functions are left-associative by default. That is,
`a b c` $\iff$ `((a b) c)`. Additionally, they get called before operators. Below are equivalences between mathematical functions and Haskell's function-calling syntax

$$
\begin{aligned}
f(x) &\iff \texttt{f x} \\
f(x,y) &\iff \texttt{f x y} \\
f(g(x)) &\iff \texttt{f (g x)} \\
f(x,g(y)) &\iff \texttt{f x (g y)} \\
f(x)g(y) &\iff \texttt{(f x) * (g y)} \iff \texttt{f x * g y}
\end{aligned}
$$

2.11.1 First Class/Higher Order Functions

First class functions are treated like data. That is, they can be:

(i) Stored in variables

(ii) Passed as arguments to other functions

(iii) Returned as values by a function

(iv) Stored in data structures

Higher order functions are functions that accept another function as an argument or return another function as its return value.

Functions as Arguments

`Int -> (Int -> Int) -> Int` indicates a function that takes in an `Int` and a **function** that takes in and returns an `Int` and returns an `Int`.

Returning a function

`Int -> (Int -> Int)` indicates a function that takes in an `Int` and returns a function that takes in an `Int` and returns an `Int`.

2.12 Map, Filter, Reduce

`map, filter, reduce` are a set of powerful higher-order functions:

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

`map` A (one-to-one) transformation on a list

```
filter :: (a -> Bool) -> [a] -> [b]
```

`filter` Filter a list via a predicate function

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

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

`reduce` Collapse a list into a single output value

2.12.1 `map`

To implement `map`, we do

```
map :: (a -> b) -> [a] -> [b]
map func [] = []
map func (x:xs) =
  (func x) : (map func xs)
```

An example of `map`:

```
cube :: Double -> Double
cube x = x^3

mapped = map cube [1, 2, 3]
```

Here, `mapped` is `[1, 8, 27]`

2.12.2 `filter`

To implement `filter`, we do

```
filter :: (a -> Bool) -> [a] -> [a]
filter predicate [] = []
filter predicate (x:xs) =
  | (predicate x) = x : (filter predicate xs)
  | otherwise = filter predicate xs
```

An example of `filter`:

```
even :: Int -> Bool
even n =
  n `mod` 2 == 0

filtered = filter even [1, 2, 3, 4]
```

Here, `filtered` is `[2, 4]`

2.12.3 `reduce`

There are two ways to reduce: `foldl, foldr` to force associativity of functions. `foldl` is left-associative; that is, `foldl` calls `f(f...f(accum, x1)..., xn)` whereas `foldr` is right-associative; that is, `foldr` calls `f(x1, f(x2, ... f(xn, accum)...))` To implement `foldl`, we do:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl func accum [] = accum
foldl f accum (x:xs) =
  foldl f new_accum xs
  where new_accum = (f accum x)
```

To implement `foldr`, we do:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr func accum [] = accum
foldr func accum (x:xs) =
  func x (foldr func accum xs)
```

An example of `foldl`:

```
sub :: Int -> Int -> Int
sub x y = y -x

left = foldl sub 100 [10, 20]
right = foldr sub 100 [10, 20]
```

Here, `left` is `110`
Here, `right` is `-90`

2.13 Lambda Functions

A lambda function is a function without a name attached to it. They are most often thought of as throwaway functions (e.g. `cube x = x ^3` $\iff$ `\x -> x ^3`).

2.13.1 Defining Lambdas

To define a lambda, we do

```
\param, param, ..., param -> expression
```

2.14 Closures

A closure is a combination of a lambda expression and all captured variables. For example, consider

```
slope_int m b = (\x -> m * x + b)
two_x_plus_one = slope_intercept 2 1

result = two_x_plus_one 9
```

Here, `result` is `19`. When we define `two_x_plus_one`, it is assigned `(\x -> 2 * x + b)` Therefore, calling `two_x_plus_one 9` passes in `9` to `x`, which then evaluates the lambda expression `\(x = 9) -> 2 * (x = 9) + 1` $\implies$ `19`, hence, `result` is `19`.

A closure captures all free variables and its lambda's bounded parameters. That is, `m, b` are free while `x` is bounded to the lambda expression (from the example above). Both of these make up its closure.

More generally, a closure consists of two things:

(i) A function of 0 or more arguments that we wish to run at some point in the future.

(ii) A list of free variables and their values that we captured at the time the closure was created.

2.15 Currying

Currying is the process of transforming a single function with multiple parameters into multiple functions with a single parameter.[3] For example,

```
f :: Int -> Int -> Int -> Int
f(x, y, z) = x + y * z

f :: Int -> (Int -> (Int -> Int))
f = \x -> (\y -> (\z-> (x + y * z)))

f :: Int -> Int -> Int -> Int
f = \x -> \y -> \z -> x + y * z
```

All three functions behave identically. Note that currying is **right-associative**. Currying is used for partial function application.

2.16 Partial Function Application

Parital function application fixes one or more parameters of a function. For example, consider:

```
mult = \x -> (\y -> (\z -> x * y * z))
f1 = mult 2
f2 = f1 3
result = f2 5
```

[3] Haskell curries functions implicitly

Here, `result` is `5`. `f1` is defined as
`\(x = 2) -> (\y -> (\z -> (x = 2) * y * z`
and `f2` is defined as
`\(x = 2) -> (\(y = 3) -> (\z -> (x = 2) * (y = 3) * z`.
So, when we call `f2 5`, we get
`\(x = 2) -> (\(y = 3) -> (\(z = 5) -> (x = 2) * (y = 3) * (z = 5)` $\implies$ `30`, hence, `result` is `30`.

2.17 Algebraic Data Types/Immutable Data Structures

An ADT is a user-defined data type that can have multiple fields (similar to C++ enum). Note that all type names must be capitalized. For example,

```
data Drink = Water | Coke | Redbull
data Veggie = Cucumber | Lettuce | Tomato
data Protein = Eggs | Beef | Chicken | Beans
data Meal =
  Breakfast Drink Protein |
  Lunch Drink Protein Veggie |
  Dinner Drink Protein Protein Veggie |
  Fasting

meal = Breakfast Redbull Eggs
```

Here, `meal` is a `Meal` data type with values `Redbull` and `Eggs` for `Drink` and `Protein` respectively.

Note that Algebraic Data Type fields are positional, and the pipe operator `|` distinguishes variants.

3 Scripting (Python)

Python is a dynamically typed scripting language that executes statements top-down (excluding control flow).

3.1 Program Execution

A main function in Python is not necessary, though you can specify one by doing

```
if __name__ == "__main__":
  main()
```

Additionally, identifiers in Python are function-scoped, meaning when created, identifiers persist until the end of the function.

3.2 Classes

In Python, the `self` keyword is explicit. By default, all attributes are public. To make them private, we do `__var` and to make them protected, we do `_var`.

Destructors (`__del__()`) are rarely used in Python since Python has automatic garbage collection.

3.3 Objects

In Python, all variables are object references (similar to pointers); that is, everything is allocated on the heap. Some objects include:

(i) Strings: Immutable

(ii) Lists: Mutable, but assignment (`=`) creates a new list

(iii) Tuples: Immutable

To get the object id of an object, we do `id(var)`.

3.4 Parameter Passing

In Python, since everything is an object reference, we also pass everything by object reference.

3.5 List/Set/Dictionary Comprehension

List and set/dictionary comprehensions have the same syntax, but are wrapped in `[]` and {} respectively. Examples are below:

```
input = [1, 2, 3, 4]
list_result = [x * 2 for x in input if x % 2 == 0]

s = "David's dirty dog drank dirty water down by the dam"
set_result = {w for w in s.split() if w[0] == 'd'}
dict_result = {w:len(w) for w in s.split()}
```

Here, `list_result` is `[4, 8]`
`set_result` is {`'drank', 'dam', 'dirty', 'down', 'dog'`},
`dict_result` is {`"David's": 7, 'dirty': 5, 'dog': 3, 'drank': 5, 'water': 5, 'down': 4, 'by': 2, 'the': 3, 'dam': 3`}

4 Data

This section covers how languages internally manage data (including types, variables, and values).

4.1 Variables and Values

A variable is a symbolic name associated with a storage location that contains either a value or a pointer (to a value). For example, in `x = 5`, `x` is a variable while `5` is the value. A variable has:

(i) Name: Variables most often have a name (identifier)

(ii) Type: A variable may or may not have a type associated with it

(iii) Value A variable stores a value (and the value's type)

(iv) Binding: How a variable is connected to its (current) value

(v) Storage: Memory location associated with a variable

(vi) Scope: Where a variable is accessible

(vii) Mutability: Can the variable's value be changed?

A value is a piece of data with a type that is referred to by the variable or computed in an expression. A value has:

(i) Type: A variable may or may not have a type associated with it

(ii) Value A variable stores a value (and the value's type)

(iii) Storage: Memory location associated with a variable

(iv) Lifetime: The longevity of the value

(v) Mutability: Can the variable's value be changed?

4.2 Types

Variables need not be typed, but values **must** have a type associated with them. We can infer about a value (given its type):

(i) The set of legal values

(ii) The set of valid operations

(iii) Memory requirements

(iv) How we interpret bytes stored in RAM

(v) How values are converted between types

Primitive types include:

(i) Integer

(ii) Float

(iii) Char

(iv) Enum

(v) Boolean

(vi) Pointer

Composite types include:

(i) String

(ii) Tuple

(iii) Union

(iv) Array

(v) Container

(vi) Struct

Other types include:

(i) Generics/Templates

(ii) Function

(iii) Boxed (Only data member is a primitive)

4.3 Static and Dynamic Typing

Languages are either **statically, dynamically** typed.

4.3.1 Static Typing

A type checker checks operations and types at **compile time**. Statically typed languages need not be explicitly typed (like C++: e.g. Haskell is statically typed but uses type inferenced).

Note: Statically typed languages **must** have a fixed type bound to each variable at the time of definition.

Statically typed languages are inherently conservative; that is, they may disallow valid programs that don't violate typing rules.

Pros

(i) Produces faster run-time code

(ii) Earlier type-related bug detection

(iii) No custom type checking

Cons

(i) Conservative (and therefore may not compile even when the program is correct)

(ii) Slower compile-time since it has to type check

4.3.2 Dynamic Typing

A dynamically typed language will check the type of a variable as they are called/referenced. Types are associated with a variable's value, not the variable itself. Additionally, they don't require explicit type annotations for variables.

Note: A type tag: All values have embedded types.

Pros

(i) Increased flexibility

(ii) Easier to implement generics that operate on different types of data

(iii) Simpler code (fewer type annotations)

(iv) Faster prototyping

Cons

(i) We detect errors much later

(ii) Slower run-time (runtime checking)

(iii) Requires more testing

(iv) No safety guarantees

Note: Downcasting (in C++) allows for runtime checking in an otherwise statically typed language

4.3.3 Duck Typing

Duck typing is a consequence of dynamic typing. If an object has the desired function name, the function will be called. (See HW for examples)

Note: Duck typing can be done (to an extent) in statically typed languages via templates (in C++).

4.3.4 Gradual Typing

Gradual typing is a paradigm that allows for some variables to be statically typed and others dynamically typed.

4.4 Strong and Weak Typing

Languages can be either **weakly, strongly** typed.

4.4.1 Strong Typing

A strongly typed language guarantees that we never get undefined behavior at runtime due to type issues. This implies that there is no possibility of an unchecked runtime type error. The minimum requirements to be strongly typed are:

(i) The language is type-safe: Prevent operations on a variable if it's incompatible.

(ii) The language is memory-safe: Prevents invalid memory access, has bounds checking, no dangling pointers.

These can be enforced either statically or dynamically. To implement strong typing, we need to

(i) Validate before evaluation

(ii) Check all conversions

(iii) Assign all pointers to either null or a valid object

As a consequence, we have that strongly typed $\implies$ memory safe.

4.4.2 Weak Typing

A weakly typed languages are generally not type-safe (may not detect type errors) nor memory safe (may allow invalid memory access). Therefore, undefined behavior is allowed.[4]

4.5 Conversions, Casts, Coercion

4.5.1 Type Relationships

We say that T_{sub} is a **subtype** of T_{super}[5] if and only if:

(i) $T_{sub} \subseteq T_{super}$

(ii) **All** operations (e.g. `+, -, *, /`) that work in T_{super} also work in T_{sub}.

4.5.2 Conversions

Values will get copied from $T_A \to T_B$. Conversions will occupy a new memory location with a different bit encoding and will now have T_B. Conversions most often happen between primitives (e.g. float $\to$ int).

4.5.3 Casts

Values do **not** get copied, but rather treats the value of T_A as if it were of T_B; that is, no conversion takes place. Casts are typically used with objects.

4.5.4 Widening and Narrowing

A conversion/cast can either be widening or narrowing.

- A conversion/cast is said to widen if $T_{sub} \to T_{super}$ (e.g. short $\to$ long).
- A conversion/cast is said to narrow if $T_{super} \to T_{sub}$ or if $T_A,\ T_B$ are not related (e.g. int $\to$ string, float $\to$ int).

4.5.5 Explicit and Implicit

A conversion/cast can either be explicit or implicit.

- An explicit conversion/cast tells the compiler to delay type checking to runtime rather than compile time.
- An implicit conversion (coercion)/cast do not require explicit syntax.

[4] ASIDE: We are moving towards strong and statically typed languages.

[5] Note that T_{sub} is a subtype of $T_{super} \iff T_{super}$ is a supertype of T_{sub}