Unit 3 - Polymorphism in Functional Programming

In Unit 3 we introduced tuples and records in Typescript. In this chapter we focus on showing how polymorphism can be implemented in a functional programming style in Typescript

Inheritance via record nesting

Inheritance is an important feature of object-oriented programming that allows, among other things, to recycle the code and the definition of existing classes and, at the same time, to enrich them with additional functionalities. In Typescript a similar result like records' inheritance, can be achieved by nesting records. Consider again the record `Tank` used in the previous unit, and suppose that we want to define a new kind of tank with more than one weapon. This would be expressed in C# as `Tank2Weapons : Tank`. In Typescript we can define a new record `Tank2Weapons` that has a field of type `Tank` containing the base record `Tank`. This new tank will have an additional field defining the secondary gun and a new way of shooting: it will first shoot the main gun of the tank and then shoot the secondary gun.

```
interface Gun {
  name          : string
  penetration   : number
  damage        : number
}

interface Tank {
  name                : string
  weapon              : Gun
  armor               : number
  health              : number
}

export interface Tank2Weapons {
  secondaryWeapon       : Gun
  base                  : Tank
}

export const createTank2Weapons = (secondaryWeapon : Gun)
          : (tank : Tank) => Tank2Weapons => tank =>
          ({
            secondaryWeapon       : secondaryWeapon,
            base                  : tank
          })
```

Now let us refactor the `shoot` function and let us define it in `Gun` instead of `Tank`, so that it becomes:

```
export interface Gun {
  name          : string
  penetration   : number
  damage        : number
  shoot         : (this : Gun, shooter : Tank, tank : Tank) => Tank
}

export const createGun = (name : string, penetration : number, damage : number) : Gun => ({
  name:             name,
  penetration:      penetration,
  damage:           damage,
  shoot:
    function (this : Gun, shooter : Tank, tank : Tank) : Tank {
      if (this.penetration > tank.armor) {
        const updatedHealth = tank.health - this.damage
```

```
        console.log(`${shooter.name} shoots` +
                    ` ${tank.name} with ${this.name}` +
                    ` causing ${this.damage} --> HEALTH: ${updatedHealth}`
        )
        return {
          ...tank,
          health: updatedHealth
        }
      }
      else {
        const updatedArmor = tank.armor - this.penetration
        console.log(`${shooter.name} shoots` +
                    ` ${tank.name} with ${this.name}` +
                    ` reducing armour by ${this.penetration} --> ARMOUR: ${updatedArmor}`
        )
        return {
          ...tank,
          armor: updatedArmor
        }
      }
    }
})
```

and let us also redefine `fight` as member of both `Tank` and `Tank2Weapons`:

```
export interface Tank {
  name                : string
  weapon              : Gun
  armor               : number
  health              : number
  fight               : (this : Tank, tank : Tank) => [Tank, Tank]
}

export const createTank = (name : string, weapon : Gun, armor : number, health : number) : Tank => ({
  name:         name,
  weapon:       weapon,
  armor:        armor,
  health:       health,
  fight:
    function(this : Tank, tank : Tank) : [Tank, Tank] {
      const outcome = (loser : Tank) => (winner : Tank) : [Tank, Tank] => {
        console.log(`${loser.name}: KABOOOM!!! ${winner.name} wins`)
        if (this == loser)
          return [this,tank]
        else
          return [tank,this]
      }
      if (this.health < 0) {
        return outcome(this)(tank)
      }
      else if (tank.health < 0) {
        return outcome(tank)(this)
      }
      else {
        const t2 = this.weapon.shoot(this, tank)
        const t1 = tank.weapon.shoot(tank, this)
        return t1.fight(t2)
```

```
      }
    }
})
```

The version of `fight` in `Tank2Weapons` will first shoot the base weapon of the current tank. This will return an updated copy of `Base` of the other tank, which must replace the current value in the `Base` field. Then it will shoot the secondary weapon thus obtaining another copy of `Base` that must replace the old value again. The same operations are performed for the second tank.

```
function(this : Tank2Weapons, tank : Tank2Weapons) : [Tank2Weapons, Tank2Weapons] {
  const outcome = (loser : Tank2Weapons) => (winner : Tank2Weapons) : [Tank2Weapons, Tank2Weapons] => {
    console.log(`${loser.base.name}: KABOOOM!!! ${winner.base.name} wins`)
    if (this == loser)
      return [this,tank]
    else
      return [tank,this]
  }
  if (this.base.health < 0) {
    return outcome(this)(tank)
  }
  else if (tank.base.health < 0) {
    return outcome(tank)(this)
  }
  else {
    const t2AfterPrimaryWeaponFired = {
      ...tank,
      base: this.base.weapon.shoot(this.base, tank.base)
    }
    const t1AfterPrimaryWeaponFired = {
      ...this,
      base: tank.base.weapon.shoot(tank.base, this.base)
    }
    const t2AfterSecondaryWeaponFired = {
      ...t2AfterPrimaryWeaponFired,
      base: t1AfterPrimaryWeaponFired.secondaryWeapon.shoot(t1AfterPrimaryWeaponFired.base,
                                                            t2AfterPrimaryWeaponFired.base)
    }
    const t1AfterSecondaryWeaponFired = {
      ...t1AfterPrimaryWeaponFired,
      base: t2AfterPrimaryWeaponFired.secondaryWeapon.shoot(t2AfterPrimaryWeaponFired.base,
                                                            t1AfterPrimaryWeaponFired.base)
    }
    return t1AfterSecondaryWeaponFired.fight(t2AfterSecondaryWeaponFired)
  }
}
```

The attentive reader will notice that now we have a design problem: we can let `Tank` fight another `Tank` and `Tank2Weapons` fight `Tank2Weapons` but we cannot mix them up (as it would make sense). This problem can be solved by using discriminated unions, thus by defining a function that accepts a `TankKind` that can be either a `Tank` or a `Tank2Weapons`, or function records, but we will explain these topics further ahead.

Inheritance through Algebraic Types (&)

Typescript supports a type operator that provides an alternatives to inheritance implemented through record nesting: it is possible to use the `&` operator with types, so that `T1 & T2` is a type that contains all the fields and methods of `T1` and `T2`. Let us redefine our `Tank2Weapons` by using this type operator:

```
export type Tank2Weapons =
  Tank & {
    secondaryWeapon       : Gun
    fightWith2Weapons     : (this : Tank2Weapons, tank : Tank2Weapons) => [Tank2Weapons, Tank2Weapons]
}
```

This will create a record with all the fields of `Tank` (including the base function `fight`) and the extra fields defined in the "extension" record passed as a second argument.

> We had to rename `fight` in `fightWith2Weapons` because otherwise the type system would be unable to disambiguate the function `fight` defined in `Tank` and the one part of the extension. The resulting type will contain both implementations.

The implementation of `fightWith2Weapons` is the following:

```
function(this : Tank2Weapons, tank : Tank2Weapons) : [Tank2Weapons, Tank2Weapons] {
  const outcome = (loser : Tank2Weapons) => (winner : Tank2Weapons) : [Tank2Weapons, Tank2Weapons] => {
    console.log(`${loser.name}: KABOOOM!!! ${winner.name} wins`)
    if (this == loser)
      return [this,tank]
    else
      return [tank,this]
  }
  if (this.health < 0) {
    return outcome(this)(tank)
  }
  else if (tank.health < 0) {
    return outcome(tank)(this)
  }
  else {
    const t2AfterPrimaryWeaponFired : Tank2Weapons = {
      ...this.weapon.shoot(this, tank),
      secondaryWeapon: tank.secondaryWeapon,
      fightWith2Weapons: tank.fightWith2Weapons
    }
    const t1AfterPrimaryWeaponFired : Tank2Weapons = {
      ...tank.weapon.shoot(tank, this),
      secondaryWeapon: this.secondaryWeapon,
      fightWith2Weapons: tank.fightWith2Weapons
    }
    const t2AfterSecondaryWeaponFired = {
      ...t2AfterPrimaryWeaponFired,
      base: t1AfterPrimaryWeaponFired.secondaryWeapon.shoot(t1AfterPrimaryWeaponFired,
                                                            t2AfterPrimaryWeaponFired)
    }
    const t1AfterSecondaryWeaponFired = {
      ...t1AfterPrimaryWeaponFired,
      base: t2AfterPrimaryWeaponFired.secondaryWeapon.shoot(t2AfterPrimaryWeaponFired,
                                                            t1AfterPrimaryWeaponFired)
    }
    return t1AfterSecondaryWeaponFired.fightWith2Weapons(t2AfterSecondaryWeaponFired)
  }
}
```

Notice that the logic of processing the result after firing the weapons has been changed a bit: shooting a weapon returns a record of type `Tank`, which needs to be boxed into an object of type `Tank2Weapons` through the spread operator.

Discriminated Unions through Algebraic Types (|)

In Typescript it is possible to specify a type as a discriminated union, which is essentially a type that can be constructed in multiple ways. This can be achieved by using the `|` type operator.

```
type T = {
  kind: "k1",
  T1:     T_1
} | {
  kind: "k2",
  T2:   T_2
} |
...
| {
  kind: "kn"
  Tn:   T_n
}
```

where `T` is the name of the type that we declare as usual, each `kind` is the name of each possible constructor or case that we can use to build our polymorphic type, and each `T_i` is the type of the input argument of that case. Notice that in typescript, you can use values as types, and that would be perfectly type safe. In this case, the only possible values assignable to `kind` are the strings in `"k1", "k2", ..., "kn"`. Any other string assigned as value will cause a type error.

This would be equivalent to the following C# code:

```
interface T { }

class k1 : T
{
  public T_1 Data;
}

class k2 : T
{
  public T_2 Data;
}

...

class kn : T
{
  public T_n Data;
}
```

For instance, let us assume that we want to model a vehicle and each vehicle is characterized by a list of components. A car as four wheels, and an engine, a tank has two tracks, an engine,and a gun, a plane has an engine and two wings. This can be modelled by the following discriminated union (assuming that we already have type definitions of `Wheel`, `Engine`, `Track`, and `Gun`):

```
type Vehicle = {
  kind: "car"
  wheels: [Wheel, Wheel, Wheel, Wheel]
  engine: Engine
} | {
  kind: "tank"
  tracks: [Track, Track]
  engine: Engine
  gun: Gun
} | {
```

```
  kind: "plane"
  wheels: [Wheel, Wheel, Wheel]
  engine: Engine
}
```

> The field `kind` has no special behaviour. You can rename it, for instance, into `case` and you would achieve the same effect. It is just as any other field of the record

Notice that we can combine the type operators `|` and `&` together: in the example above, all three kinds of vehicles have an engine, so the type above can be rewritten as:

```
type Vehicle = ({
  kind: "car"
  wheels: [Wheel, Wheel, Wheel, Wheel]
} | {
  kind: "tank"
  tracks: [Track, Track]
  gun: Gun
} | {
  kind: "plane"
  wheels: [Wheel, Wheel, Wheel]
}) & {
  engine: Engine
}
```

This allows us also to add methods common to all cases of the discriminated union, for instance

```
type Vehicle = ({
  kind: "car"
  wheels: [Wheel, Wheel, Wheel, Wheel]
} | {
  kind: "tank"
  tracks: [Track, Track]
  gun: Gun
} | {
  kind: "plane"
  wheels: [Wheel, Wheel, Wheel]
}) & {
  engine: Engine
  run: (this : Vehicle) => ...
}
```

Pattern Matching

Let us suppose that we want to give the implementation of the method `run` described in the snippet above, so that each kind of `Vehicle` outputs a different sound. In order to do that, we must check which specific case of the vehicle polymorhpic type we are considering. In pure object-oriented programming this is usually achieved through a visitor or strategy design pattern. Several functional programming languages (but now also OO languages are starting to offer this feature, for instance C#) offer instead built-in abstractions to achieve the same behaviour. In Typescript this abstraction can be done by checking the value of the `kind` property in the discriminated union.

Example:

Consider the two following type definitions:

```
interface DegreeCourse
  {
    code              : string
    name              : string
    enrollmentDate    : [bigint, bigint, bigint]
  }

interface Contract
  {
    serial              : string
    salary              : number
    beginDate           : [bigint, bigint, bigint]
  }

interface Student
  {
    id          : string
    name        : string
    lastName    : string
    enrollment  : DegreeCourse
  }

interface Employee
  {
    id          : string
    name        : string
    lastName    : string
    enrollment  : Contract
  }

type SchoolPerson = {
  kind: "student"
} & Student | {
  kind: "employee"
} & Employee
```

and the following code:

```
const schoolPerson : SchoolPerson = {
    kind: "student",
    id     : "0963963",
    name   : "John",
    lastName : "Doe",
    code    = "INF",
    name    = "Computer Science",
    enrollmentDate = (15n,08n,2018n)
```

```
  }

if (schoolPerson.kind == "student") {
  ...
}
else {
  ...
}
```

In this case typescript will be able to discriminate that the `SchoolPerson` is indeed of specific type `Student` because the comparison in the conditional will return `true`. Notice that, thanks to Typescript advanced type system, when you are in the scope of the `if` block, all the fields of the specific type `Student` will become available

Options

One simple example of a discriminated union is the type `Option<T>`. An option represents an optional value, which is used in a situation when we are not sure whether we have a value or not. It is a safe version of `null`, because having a discriminated union will force you to explicitly test whether you have a value through pattern matching to be able to access its value. The optional type is defined as :

```
type Option<T> = {
  kind : "none"
} | {
  kind: "some"
  value: T
}
```

Consider now a simulation of a server connection, where a server can fail to reply. A server is defined by an address, which consists of four bytes, a reply probability, which simulates the reliability of the server, and a reply message. A connection is built by providing the ip address of the server you want to connect to. The connection holds an optional `data` field which contains a possible answer from the server. Note that this is optional because we cannot know if the server will ever reply or not. The connection has a method `connect` that takes as input a `Server` and tries to connect to it and get a reply. The connection fails to get a reply if the address of the server does not match or if a random number is greater than the chance of getting a reply (we simulate a server failure). Note that to access the reply of the server we must pattern match the `data` field against the possible pattern of `Option<T>`: if `data` has kind with value `some` then we can access `data` contained in it. You can see the implementation of this example below.

```
interface Server
  {
    address   : [bigint, bigint, bigint, bigint]
    replyProbability : number
    reply     : string
  }
const createServer = (address : [bigint, bigint, bigint, bigint],
                      replyProbability : number, reply : string) : Server => ({
  address: address,
  replyProbability: replyProbability,
  reply: reply
})

interface Connection {
    address   : [bigint, bigint, bigint, bigint]
    data      : Option<string>
    connect   : (this : Connection, server : Server) => Connection
}
const createConnection = (address : byte * byte * byte * byte) : Connection => ({
  address : address,
  data    : { kind : "none" }
  connect :
    function (this : Connection, server : Server) : Connection {
      if (this.Address != server.Address || Math.random() > server.replyProbability) {
        return this
      }
      return {
        ...this,
        data: {
          kind: "some",
          value: server.reply
        }
      }
```

```
    }
  }
)


const helloServer = createServer([169n, 180n, 0n, 1n], 0.35, "Hello!")
const connection = createConnection([169n, 180n, 0n, 1n])
if (connection.connect(helloServer).data.kind == "some") {
  console.log(`The server replied: ${connection.data.value}`)
}
else {
  console.log "404 server not found"
}
```

Immutable Lists

Immutable Lists are another data type that can be defined by using discriminated unions. We can think of a list as either an empty collection, or an element followed by a collection of elements (possibly empty). This element is usually defined as `head`. We can thus define a list as

```
type List<T> = {
  kind: "empty"
}
| {
  kind: "list",
  head: T,
  tail: List<T>
}
```

We can implement an utility function that takes as input a Typescript array and generates an immutable list.

```
export const List = <T>(array : T[]) : List<T> => {
  let x : List<T> = { kind : "empty" }
  for (let i = array.length - 1; i >= 0; i--) {
    x = {
      kind: "list",
      head: array[i],
      tail: x
    }
  }
  return x
}
```

> Notice that we start from the last element of the array, because otherwise the elements would be in the inverse order.

Functional versions:

```
export const List = <T>(array : T[]) : List<T> => {
  if (array.length == 0) return { kind : "empty" }
  else if (array.length == 1) return {kind: "list", head: array[0], tail: {kind : "empty" } }
  return {kind: "full", head: array[0], tail: List(array.slice(1)) }
}
```

```
const List = <T>(array : T[]) : List<T> =>
                        (array.length == 0) ?
                        { kind : "empty" } :
                        (array.length == 1) ?
                        {kind: "list", head: array[0], tail: {kind : "empty" } } :
                        {kind: "list", head: array[0], tail: List(array.slice(1)) }
```

Functions on lists are generally recursive, for example the following code computes the length of a list:

```
const length = <T>(l : List<T>) : bigint => {
  if (l.kind == "empty") {
    return 0n
  }
  return 1n + len(l.tail)
}
```

Example

Let us compute the sum of the elements of a list. The base case of the recursion is when we have an empty list, whose elements add to 0. The recursive step will call `sum` on the tail of the list and add the result to the element in the head.

```
export const sum = (list : List<number>) : number => {
  if (list.kind == "empty") {
    return 0
  }
  return list.head + sum(list.tail)
}
```

Example

Let us implement the function `unzip` that, given a list `l` of pairs, creates a pair of lists `l1` and `l2`, where `l1` contains all the elements in the first item of the pairs, and `l2` contains all the elements in the second item of the pairs.

Let us start by reasoning on the type signatures of the function: the input list is a list of pairs. There is no restriction on the specific types of the elements of the pairs, so we use two generic types, which means `List<'a * 'b>`. The function returns a pair of lists, where the first one contains all the elements in the first item of each pair and the second all the elements in the second item. This means that the return type of the function is `List<'a> * List<'b>`.

The function is implemented recursively: the base case is of course an empty list, for which we return a pair of empty lists. The recursive case applies `unzip` to the tail of the list. This results in a pair of lists `l1` and `l2`. After this step has been computed we insert the first element of the pair in the head in front of `l1` and the second element in the front of `l2`.

```
export const unzip = <T1, T2>(list : List<[T1, T2]>) : [List<T1>, List<T2>] => {
  if (list.kind == "empty") {
    return [{ kind: "empty" }, { kind: "empty" }]
  }
  const [l1, l2] = unzip(list.tail)
  return [{
    kind: "list",
    head: list.head[0],
    tail: l1
  }, {
    kind: "list",
    head: list.head[1],
    tail: l2
  }]
}
```

Note that we used a tuple decomposition to individually bind the elements of the pair to `l1` and `l2`.

Example

Let us consider a list where each element can be either an atomic element or a nested list. We want to implement a function that flattens the elements of this list into a list with just one layer. For instance `[3;5;[4;3;[2;1;3];1];3;4;[1;2;3];6]` becomes `[3;5;4;3;2;1;3;1;3;4;1;2;3;6]`. Unfortunately (or rather, fortunately) in our definition of immutable list, elements of a list are homogeneous, meaning that they must have

the same type. Achieving our purpose requires to define a polymorphic type that can be either an atomic element or a nested list:

```
export type ListElement<T> = {
  kind: "element",
  value: T
} | {
  kind: "nested list",
  nestedList: List<ListElement<T>>
}
```

Note that a `nested list` can recursively contain other nested lists, so the type definition is recursive.

Our `flatten` function will take as input a `List<ListElement<T>>`, so that each element can be either a simple element or a nested list. It will return simply `List<T>`, because, after it runs, all the elements will be at the same level of nesting. The base case of the recursion is an empty list, which will simply return an empty list. The recursive case must decide what to do depending on whether the head is a simple element or a nested list. In the case of a simple element we extract the content of `element`, we recursively flatten the tail of the list, and then we prepend the content of `element` to it. The case of a nested list is slightly more complex: we apply pattern matching on the head of the list extracting the list itself from the polymorphic type. Whe then recursively flatten the nested list, we flatten the tail, and finally we concatenate the head flattened list with the flattened tail. We provide also the implementation of the `concat` support function necessary to complete the implementation:

```
export const concat = <T>(list1 : List<T>, list2 : List<T>) : List<T> => {
  if (list1.kind == "empty") {
    return list2
  }
  return {
    kind: "list",
    head: list1.head,
    tail: concat(list1.tail, list2)
  }
}

export const flatten = <T>(list : List<ListElement<T>>) : List<T> => {
  if (list.kind == "empty") {
    return { kind : "empty" }
  }
  if (list.head.kind == "element") {
    return {
      kind: "list",
      head: list.head.value,
      tail: flatten(list.tail)
    }
  }
  return concat(flatten(list.head.nestedList), flatten(list.tail))
}
```

Exercises

Exercise 1

Implement a function

```
const last = <T>(list: List<T>) : Option<T> => ...
```

that returns the last element of a list. The function should handle appropriately the case of an empty list given as input.

Exercise 2

Implement a function

```
const reverse = <T>(list: List<T>) : (newList: List<T>) => List<T> => newList = ...
```

that creates a list containing the elements of `l` in reversed order.

Exercise 3

Implement a function

```
const append = <T>(list1: List<T>) : (list2: List<T>) => List<T> => list2 => ...
```

that creates a list containing all the elements of `list2` after those in `list1`.

Exercise 4

Implement a function

```
const nth = <T>(n : bigint) : (l : List<T>) => Option<T> => l => ...
```

that returns the element in position `n` in `l`. The function must handle appropriately the case where the index is invalid.

Exercise 5

Implement a function

```
const palindrome = <T>(l : List<T>) : boolean => ...
//In order to compare two lists:
const listEquals = <T>(l1 : List<T>) : (l2 : List<T>) => boolean => l2 => ...
```

that detects if a list is palindrome. A list is palindrome if it is equal to its inverse.

Exercise 6

Implement a function

```
const compress = <T>(l : List<T>) : List<T> =>  ...
```

that removes consecutive occurences of the same element in `l`. For example compressing `[a;a;a;a;b;b;c;c;b]` result in `[a;b;c;b]`.

Exercise 7

Implement a function

```
const caesarCypher = (l : List<string>) : (shift : number) => List<string> => shift => ...
const shiftChar = (c : string) : (shift : number) => string => shift => ...
```

The Caesar's cypher take a text, represented as a list of characters, and shifts all the letters (so only if the character is an alphabetical character) in it up by the number of position specified by shift. If the letter goes past z it restarts from a. You can assume that all the text is in lower-case letter (97 <= charCode <= 122).

```
shiftChar("c")(5) = "h"
shiftChar("y")(5) = "d"
```

The ASCII code for a specific character can be obtained by using method `charCodeAt`, the opposite conversion using method `fromCharCode`.

Advanced Support also upper-case letters.

Exercise 8

Implement a function

```
const splitAt = <T>(i : number) : (l : List<T>) => [List<T>, List<T>] => l => ...
```

that splits the list into two lists, the first one containing all the elements of `l` from position 0 to position `i` included, and the second one containing all the remaining elements. The two resulting lists are returned in a tuple. For example `split 3 [3;5;4;-1;2;2] = ([3;5;4;-1],[2;2])`.

Exercise 9

Implement a function

```
const Merge = <T>(l1 : List<T>) : (l2 : List<T>) => List<T> => l2 => ...
```

that merges together two **sorted** lists into a single sorted list.

Exercise 10

Implement a function

```
const MergeSort = <T>(l: List<T>) : List<T> => ...
```

implement the Merge Sort algorithm. The Merge Sort returns the list itself when the list has only one element. Otherwise it splits l in two lists, one containing the elements between position 0 and `l.Length / 2`, the other containing the elements from `l.Length / 2 + 1` until the end. Merge Sort is called recursively on the two lists. After this step use the function `merge` above to merge the two sorted lists.

Exercise 11

Implement a function

```
const eval = (expr : Expr) : Expr => ...
```

The function allows to evaluate arithmetic expressions. An arithmetic expression is defined as a discriminated union with the following cases:

- An atomic value. This is just a number
- The sum of two expressions.
- The difference of two expressions.
- The product of two expressions.
- The division of two expressions.

The function eval for an atomic value simply returns the atomic value. For the other cases it simply evaluates recursively the arguments of the operator and then combine the two results together. For example, evaluating the sum of two expressions `expr1 + expr2` recursively evaluates `expr1` and `expr2` and then sum ups the two atomic values together. The other operators behave analogously.

Exercise 12

Implement a function

```
const eval = (expr : Expr) : (stack : List<[string, number]>) => Expr => stack => ...
```

Extend the previous version of the function `eval` by supporting also variables. Variables are another case of the union. Evaluating a variable means looking it up in the stack. A stack is a list of tuples mapping a variable name to its value. If the lookup is successful `eval` returns the value of the variable, otherwise it throws an error.

Exercise 13

```
const run = (program : List<Statement>) :
            (stack : List<[string, float]>) => List<[string, number]> => stack => ...
```

The function interprets a program from a small imperative language that can consists of the following statements, expressed as a discriminated union. - Variable assignment. A variable assignment contains two arguments: the name of the variable, and an expression that is used to assign a value to it. First the expression must be evaluated and then, if the variable already exists in stack, its value is updated, otherwise we add the variable name and the value in stack. - Print to output. This data structure contains an expression, which is evaluated and then prints the value on the console.

The evaluation stops when the list of statements is empty.