Example solutions (fall 2020)

This is a subset of the questions that were asked in Java on 6.031 Fall 2020 Quiz 2. They have been translated to TypeScript here.

In this quiz, we will use an abstract type `Name` to represent full names of celebrities.

Examples of full names include:

- Alan M. Turing
- Sarah Jessica Parker
- Madonna
- M. Night Shyamalan
- Harry S Truman

A full name has up to three parts: a first name, a middle name, and a last name.

First, middle, and last are called *positions* within the name.

Only the last name is required, and a middle name requires a first name before it.

Each part of a name is a nonempty string of letters (unless abbreviated as mentioned next). Each name part starts with a capital letter.

First and middle names may be *abbreviated* by an initial letter and a period. Note that the S in Harry S Truman is not abbreviated; that letter is his middle name.

Note that this type represents only a particular subset of possible names. It was chosen for the sake of having a familiar domain for this quiz, and is not a good model for a real system dealing with human names.

Provided at the bottom of the quiz page are:

- `Name`
- `Position`

You can open this text and the code in a separate tab by clicking the "open in separate tab" button at the top right.

Problem 1. (28 points)

Suppose `Name` is represented as a recursive datatype using the following (incomplete) datatype definition:

```
Name = A(part:string, rest:Name) + ________________
```

Complete the datatype definition by filling in the blank with one more variant named `B`.

> B(lastname:string)

As the implementer of `Name` with this datatype definition, how many objects (of any type) are you using to represent the name Clive Owen? It may help to draw a snapshot diagram. Do not count objects that are not visible to you as the implementer of `Name`.

> 4 objects (1 A + 1 B + 2 strings)

As the implementer of `Name` with this datatype definition, how many objects (of any type) are you using to represent the name Cher? It may help to draw a snapshot diagram. Do not count objects that are not visible to you as the implementer of `Name`.

> 2 objects (1 B + 1 string)

As the implementer of `Name` with this datatype definition, how many objects (of any type) are you using to represent the name C. Thomas Howell? It may help to draw a snapshot diagram. Do not count objects that are not visible to you as the implementer of `Name`.

> 6 objects (2 A + 1 B + 3 strings)

Give a rep invariant for the `A` variant. Use regular expressions and `Name` operations where appropriate.

> ```
> part matches [A-Z](\.|[A-Za-z]*)
> rest.parts().size() < 3
> ```
>
> Note that referring this.lastName() or this.parts() would NOT be appropriate for a rep invariant – the rep invariant must be a function of the rep (i.e. the part field and the rest field), not a function of the abstraction (this). It can only use Name operations recursively on rest, not on `this`.

Implement the `lastName` method for `A` and `B`:

```
class A implements Name {
    ...
    public lastName(): string {

     return this.rest.lastName();

    }
}

class B implements Name {
    ...
    public lastName(): string {

     return this.lastname

    }
}
```

Implement the `parts` method for `A` and `B`. (You may use Python syntax for list manipulation here if you find it easier.)

```
class A implements Name {
    ...
    public parts(): Array<string> {

     return [this.part, ...this.rest.parts()];

    }
}

class B implements Name {
    ...
    public parts(): Array<string> {

     return [this.lastname];

    }
}
```

Problem 2. (25 points)

Write a grammar for full names that disallows middle-name abbreviations. Assume the parts of a name are separated by a single space when written as a string. Your grammar must have a root nonterminal called `s`, and include a `firstname` nonterminal that matches only the corresponding part of the name. It may have other nonterminals as well if you wish, and should be DRY.

```
s ::= (firstname ' ' (middlename ' ')? )? lastname ;
firstname ::= name | abbreviation ;
middlename ::= name ;
lastname ::= name ;
name ::= [A-Z][A-Za-z]* ;
abbreviation ::= [A-Z] '.' ;
```

Problem 3. (25 points)

Write a grammar for full names that disallows first-name abbreviations. Assume the parts of a name are separated by a single space when written as a string. Your grammar must have a root nonterminal called `root`, and include a `middlename` nonterminal that matches only the corresponding part of the name. It may have other nonterminals as well if you wish, and should be DRY.

```
root ::= (firstname ' ' (middlename ' ')? )? lastname ;
firstname ::= name ;
middlename ::= name | abbreviation ;
lastname ::= name ;
name ::= [A-Z][A-Za-z]* ;
abbreviation ::= [A-Z] '.' ;
```

Problem 4. (25 points)

Write a grammar for full names. Assume the parts of a name are separated by a single space when written as a string. Your grammar must have a root nonterminal called `thename`, and include a `lastname` nonterminal that matches only the corresponding part of the name. It may have other nonterminals as well if you wish, and should be DRY.

```
thename ::= (firstname ' ' (middlename ' ')? )? lastname ;
firstname ::= name | abbreviation ;
middlename ::= name | abbreviation ;
lastname ::= name ;
name ::= [A-Z][A-Za-z]* ;
abbreviation ::= [A-Z] '.' ;
```

Problem 5. (25 points)

Suppose we introduce new operations `map` and `filter` that transform a name into another name by applying a function to each part of the name.

The `map` operation replaces a name part with another string, and `filter` keeps or removes a name part.

The parameters to the applied function include not only the part of the name, but also its *position* in the name, i.e. first, middle, or last.

The result of `map` and `filter` must be a valid `Name`. For `filter`, removing a name part may cause another name part to change its position in the output, e.g. a middle name may become a first name or last name.

For example, if `name` represents the name John Rhys Davies, then:

- `name.map((part, position) => part.toUpperCase())` returns the name JOHN RHYS DAVIES
- `name.filter((part, position) => position === Position.Middle)` returns the name Rhys (which is now a last name)

Given a `Name` variable `name`, use `map` and `filter` to abbreviate the first name and remove any middle name. So Sarah Jessica Parker should become S. Parker, and Madonna is of course always Madonna.

```
name.map((part, position) =>
         position === Position.First ? part.substring(0, 1) + "." : part)
    .filter((part, position) => position !== Position.Middle)
```

Problem ⤖6. (25 points)

Suppose we introduce new operations `map` and `filter` that transform a name into another name by applying a function to each part of the name.

The `map` operation replaces a name part with another string, and `filter` keeps or removes a name part.

The parameters to the applied function include not only the part of the name, but also its *position* in the name, i.e. first, middle, or last.

The result of `map` and `filter` must be a valid `Name`. For `filter`, removing a name part may cause another name part to change its position in the output, e.g. a middle name may become a first name or last name.

For example, if `name` represents the name John Rhys Davies, then:

- `name.map((part, position) => part.toUpperCase())` returns the name JOHN RHYS DAVIES
- `name.filter((part, position) => position === Position.Middle)` returns the name Rhys (which is now a last name)

Given a `Name` variable `name`, use `map` and `filter` to generate a name that expands an abbreviated middle initial "B." to "Bob", and removes all other middle names. So Billy B. Thornton should become Billy Bob Thornton, and Sarah Jessica Parker should become Sarah Parker.

```
name.filter((part, position) => position !== Position.Middle || part === "B."))
   .map((part, position) => position === Position.Middle && part === "B." ? "Bob" : part)
```

Fill in the blank for the type signature of `filter`, as an operation of the `Name` type:

```
filter: ____________________________________ => Name
```

```
Name x (string x Position => boolean)
```

Fill in the blank for the type signature of `map`, as an operation of the `Name` type:

```
map: ____________________________________ => Name
```

```
Name x (string x Position => string)
```

Consider the function `(part,position)=>...` that is passed to `map`. Let's call it `f`. Not all such functions `f` are legal inputs to `map`. Write the spec that `map` requires the function `f` to satisfy, putting the precondition of `f` in the first box, and the postcondition of `f` in the second box. You must use regular expressions wherever appropriate. Note that this question is asking about **map**.

Precondition:

```
if position is Position.First or Position.Middle,
then part matches [A-Z]('.' | [A-Za-z]*),
otherwise part matches [A-Z][A-Za-z]*
```

Postcondition:

```
if position is Position.First or Position.Middle,
then returned string matches [A-Z]('.' | [A-Za-z]*),
otherwise returned string matches [A-Z][A-Za-z]*
```

Provided Code

```
/** Immutable type representing a full name as described in the overview. */
interface Name {
    /** @return the last name of this name */
    lastName(): string;

    /** @return all the parts of the name, in order from first to middle to last. */
    parts(): Array<string>;
}

/** The position of a name part within a full name. */
enum Position {
    First, Middle, Last;
}
```