CIS*2750
Assignment 2

Module 2

You did some basic vCard format validation in Assignment 1. In Assignment 2, you will need to expand the validation. We do not want to re-write `createCard()`, so we will define a new function, which will validate an already created Card object.

This, like the rest of Assignment 2 functionality, will be necessary in Assignment 3. Assignment 3 will have to create/modify existing vCard files by creating and manipulating Card objects through a Python interface. The validation functionality we define here will help us apply defensive programming principles and validate Card objects before we pass them to `writeCard()` and save the to the file.

Module 2 functionality

```
VCardErrorCode validateCard(const Card* obj);
```

This function validates a Card object. If the argument satisfies all Card rules, the function returns `OK`. Otherwise, the function returns an appropriate error. Error codes are discussed below.

We will validate two aspects of the Card struct:
- It must match the specification in `VCParser.h`
- It must further validate some - but not all - aspects of the vCard format.

Validating Card struct implementation constraints

Validating the Card object against our Card struct specification is pretty straightforward. Each struct (type) defined in `VCParser.h` has a number of constraints for each member. For example, some members may be NULL (e.g. `Card->birthday`), while others may not (e.g. `Card->group`, `Card->optionalProperties`, `Property->values`, etc.).

You must verify that **all** of these constraints are satisfied for **all** types defined in the `VCParser.h` header file. Every one of these constraints is listed in the comments in `VCParser.h`, and you should be familiar with them from implementing Assignment 1.

Validating Card struct against vCard specification

In addition to validating the "shape" of the Card struct, we must also validate its contents to make sure that they conform to the vCard specification. The vCard format is quite complex, so we are still implementing/validating only a subset of it.

We will **only** consider properties described in Sections 6.1 - 6.9.3. **All** other properties - e.g. `iana-token`, `x-name`, etc. - are considered **invalid**. If your validation code encounters properties with names other than those described in Sections 6.1 - 6.9.3, treat these properties as invalid.

Validation details:

- We must verify that each `Property` name is from Sections 6.1 - 6.9.3.
- We must verify that all required properties are present in the object - i.e. `FN` is not `NULL`.
  - Note: a `Card` object does not need to have a VERSION property specified. We assume that the version is always 4.0. If a Card object contains a VERSION in its `otherProperties`, we treat it as a duplicate and therefore an error.
- We must validate the cardinality for all properties **and** their values.
  - For example:
    - the N property must occur only once, though it is optional. So if we find more than one property with the name N in the `optionalProperties` list, the object is invalid and we return an error.
    - the N property must have be a list of exactly five components, some of which might be empty. In other words, a property with the name N must have the `values` list of length 5.
- Property values are not NULL. Each entry in the list is a string (which may be empty).
- We must validate that the `Parameter` name and value are not empty strings. We do not need to do any further validation of `Parameter` structs. The only exception will be the two date-time properties (BDAY and ANNIVERSARY), which are discussed in more detail below.
- For the `birthday` and `anniversary` fields (`DateTime` type), we need to make sure that they are internally consistent:
  - All constraints for the `DateTime` struct are satisfied (text `DateTime` must have empty strings in `date` and `time` fields, etc.)
  - text `DateTime` cannot be UTC
  - when `DateTime` is not text, the `date` and `time` fields must have the correct format.

You **do not** need to do any further validation of property values and parameters . For example, you do not need to verify that the type of the property/parameter values is correct, etc..

Error Codes

Your function must return the following error codes:

- `INV_CARD` is returned if the **card object itself** is invalid, i.e.:
  - one of more of the `Card` struct requirements is not satisfied
  - VERSION appears in other properties

    If the error is in one of the sub-components (e.g.property or parameter) contained within a `Card` object, return a component-specific error instead (see below).

    If the argument `obj` is NULL, return `INV_CARD`

- `INV_PROP` is returned if a Property inside a Card is invalid:
  - `Property` or its `Parameter`(s) violate any of the `Property` or `Parameter` struct requirements
  - property is not from the list specified in in Sections 6.1 - 6.9.3
  - property violates the cardinality rules for that property (except BDAY and ANNIVERSARY - see below)
  - property value list does not have the length listed in the vCard spec for that property
  - missing property value
  - missing parameter name / value

If the error is in the `DateTime` struct, return `INV_DT` (see below).

- `INV_DT` is returned if the `DateTime` structs inside a Card are invalid or BDAY/ANNIVERSARY rules are violated:
  - Inconsistent `DateTime` struct
  - BDAY or ANNIVERSARY appear in the `optionalProperties` list of the card

- `OTHER_ERROR` is returned if some other, non-card error happens (e.g. malloc returns NULL). Since the Card object has already been created, we would not expect a lot of errors of this type, but you might run into something.

Testing Module 2

To test this functionality you have a few options. You can manually create a temporary stub with a `main()` function, which manually creates a Card object, fills it with valid/invalid data, and send it to `validateCard` - which must return an appropriate error code.

You can also pass structs created by `createCard` from valid files to it, though toy must first be sure that your `createCard` works properly.