CIS*2750
Assignment 2

Module 2

Overview

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

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

You will be provided with an updated temporary header file (`CalendarParser_A2temp2.h`). This header will contain the prototype for the new function, and a slightly expanded `ICalErrorCode` enum.

Note:
The Calendar object can contain multiple optional calendar properties. As mentioned below, we will consider all `iana-comp` and `x-comp` content to be **invalid**, including the "begin"/"end" lines surrounding this content.

Module 2 functionality

```
ICalErrorCode validateCalendar(const Calendar* obj);
```

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

We will validate two aspects of the Calendar struct:
- It must match the specification in `CalendarParser_A2temp2.h`
- It must further validate some - but not all - aspects of the iCalendar format.

Validating Calendar struct implementation constraints

Validating the Calendar object against our Calendar struct specification is pretty straightforward. Each struct (type) defined in `CalendarParser_A2temp2.h` has a number of constraints for each member. For example, none of the struct members may be NULL, and none of the required properties may be empty. Some lists (e.g. `Calendar->events`) must have at least one value, while others (e.g. `Event->properties`) might be empty. Several struct fields have a fixed length, and you must verify that they do not exceed it.

Please note that the constraints in `CalendarParser_A2temp2.h` have been updated and clarified relative to the earlier iterations of the header. You must verify that **all** of these constraints are satisfied for **all** types defined in the Calendar header file. Every one of these constraints is listed in the comments

in `CalendarParser_A2temp2.h`. Read the comments in `CalendarParser_A2temp2.h` carefully, and make sure you don't miss anything.

Validating Calendar struct against iCalendar specification

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

In particular:

- All components other than those specified in Sections 3.6 (Calendar), 3.6.1 (Event), and 3.6.6 (Alarm) are considered **invalid** (i.e. TimeZone, etc.).
- We must verify that the file you are parsing does not violate the property rules defined in Sections 3.6 (Calendar), 3.6.1 (Event), and 3.6.6 (Alarm). Remember, some properties may only appear in the Calendar, some in an Event, etc.. Some properties are optional, but must appear only once; some properties are optional, but must appear if another optional property appears (e.g. `DTSTART`/`DTEND`/`DURATION`), etc..

  Notes on components:

  - For our alarms, we assume that all alarm properties conform to the `audioprop` rule.
  - Some of you may have noticed that there is a relationship between the `method` property of the calendar and the `dtstart` property of an event. Since we made the `dtstart` property required, certain other properties become required (as described in the event specification).

    Ideally, we would update the Event struct to reflect this, but we won't, to avoid adding unnecessary API revisions and simplify regression testing of Assignment 2. It is a bad design decision that we are going to have to live with.
- We will **only** consider properties described in Sections 3.7 - 3.8.7 to be valid. **All** other properties - e.g. `iana-comp`, `x-comp`, etc. - are considered **invalid**. If your validation code encounters properties other than those described in Sections 3.7 - 3.8.7 in some property list, treat the struct containing this list as invalid.
- We must validate the number of occurrences for all properties. For example, if an optional property is described as " MUST NOT occur more than once" in the iCalendar specification, you should use the search API of the list containing that property and make sure that the list does not contain any additional instances of this property.
- We **do not** need to validate that property values follow rules in Sections 3.7 - 3.8.7. Each Property struct **must** have a non-empty `propDescr` filed with the property value, or parameters+value. As long as `propDescr` is not an empty string, we treat it as valid.

Error Codes

If a property "rule" is violated inside a component, we consider **that component** to be invalid. Return an error that corresponds to the component in which the invalid property occurs. Your function must return the following error codes:

- `INV_CAL` is returned if the **calendar object itself** is invalid, e.g.:
  - missing required calendar properties or components, including an empty `events` list
  - invalid optional calendar properties
    - A property that should not occur in a Calendar
    - A property that violates its "number of occurrences" rule

    - A property with an empty value
    - etc.
  - any of the lists are `NULL`
  - etc..

  If the error is in one of the components contained within a Calendar object, return a component-specific error instead (see below).

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

- `INV_EVENT` is returned if an **event component** is invalid:
  - missing required event properties or components
  - invalid optional event properties, including
    - A property that should not occur in an Event
    - A property that violates its "number of occurrences" rule
    - A property with an empty value
    - etc.
  - any of the lists are `NULL`
  - etc..

  If the error is in an Alarm component contained inside an Event, return `INV_ALARM` (see below).

- `INV_ALARM` is returned if an **alarm component** is invalid:
  - missing required alarm properties or components
  - invalid optional alarm properties, including
    - A property that should not occur in an Alarm
    - A property that violates its "number of occurrences" rule
    - A property with an empty value
    - etc.
  - etc..

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

  Note: if you run into a `NULL` pointer anywhere in this function, return an error that correspondents to the component that contains the `NULL` pointer. So if the argument `obj` is `NULL`, return `INV_CAL`. If the `trigger` field in an Alarm is `NULL`, return `INV_ALARM`, etc..

Note: if the struct contains multiple errors, the error code should correspond to the highest level error that you encounter. For example, if the argument to `validateCalendar` contains:
- an invalid Calendar property and
- an invalid Alarm component inside an Event component

you **must** return `INV_CAL`, not `INV_ALARM`.

Testing Module 2

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

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