CIS*2750
Assignment 2
Deadline: Monday, October 23, 9:00am
Weight: 10%

Part 1: Library update

1.1 New functionality

You will need to expand Assignment 1 to add the following functionality:

The parser now **must** handle folded lines exactly as described in the iCalendar specification.

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.

The Calendar object can contain multiple event components.

The Calendar library now has functions for validating Calendar structs and saving them to disk in valid iCalendar format.

Please note that `CalendarParser.h` has been updated to allow for this new functionality.

In addition, you will need to update the List API to support new functionality. List struct now includes a `length` field. Update the List API to add the following:
- Function for retrieving list length. You'll also need to update your insert/delete functions to update list size.
- Function for searching the list for a specific entry using a comparator function.

Please note that `LinkedListAPI.h` has been updated to allow for this functionality.

New required Calendar API functions

```
ErrorCode writeCalendar(char* fileName, Calendar* obj);
```

This function takes an iCalendar object and saves it to a file in iCalendar format. This might seem daunting, but your `printCalendar` function might already have a lot of this functionality (traversing the object links), so you can borrow a lot of ideas (and code) from it. You need to make sure that the text output is in correct iCalendar format, including line endings. You also need to make sure that the lines longer than 75 characters are correctly folded. The function must return `WRITE_ERROR` if writing fails for some reason.

A simple way to test this function is:
- Read a valid iCalendar file into an iCalendar object using `createCalendar`
- Write this object into a different file using `writeCalendar`
- As long as the original file didn't have any folded lines, comments, or ignorable properties, the output file should be identical. You can play with the order of properties in the input file to make sure that everything comes in and out in the same order.

```
ErrorCode validateCalendar(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:
- `INV_CAL` is returned if the calendar itself is invalid (missing required calendar properties or components, invalid opening/closing tags, invalid properties, etc.)
- `INV_EVENT` is returned if an event component is invalid (missing required properties or components, invalid opening/closing tags, invalid properties, etc.)

- `INV_ALARM` is returned if an event component is invalid (missing required properties or components, invalid opening/closing tags, invalid properties, etc.)
- `OTHER_ERROR` is returned is some other, non-calendar error happens (e.g. malloc returns NULL).

This function will be used by the user interfaces in Assignments 2 and 3 for validating Calendar objects created by the users.

In addition, since Event can now be stored in a list, you need the compare/print/delete functions for it. Hopefully you have already implemented print and delete for an event in Assignment 1.

1.2 Format validation

You did some basic iCalendar format validation in Assignment 1. In Assignment 2, you will need to expand the validation. You will need to update `createCalendar()`, and all the relevant functions that it calls, to add this.

Keep in mind that the iCalendar format is quite complex, so we are still implementing/validating only a subset of it.

In particular:
- We will **only** consider properties described in Sections 3.7 - 3.8.7. **All** other properties - e.g. `iana-comp`, `x-comp`, etc. - are considered **invalid**. If your parsing or validation code encounters properties other than those described in Sections 3.7 - 3.8.7, treat the component containing it as invalid.
- For our alarms, we assume that all alarm properties conform to the `audioprop` rule.

You will need to verify that the file you are parsing does not violate the property rules defined in Sections 3.6, 3.6.1, and 3.6.2. Some properties are optional, but must appear only one; some properties are optional, but must appear if another optional property appears (e.g. `dtstart/dtend/duration`), etc..

You have probably already noticed that the only iCalendar "things" that start with "Begin" are:
- Components
- "iana-comp" and "x-comp" lists in the Calendar object - which we must ignore
- "standardc" and "daylightc" properties in the Time Zone component - which we don't need to worry about, since we don't need to parse Time Zone components.

This makes is easy to figure out if a content line (after you unfolded it) contains a property. If the current property is described as " MUST NOT occur more than once" in the iCalendar specification, you should use the updated list API to search the property list of the component you're currently parsing and make sure it does not already contain this property.

Note: 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. For sake of simplicity (and to avoid multi-pass parsing) you can ignore this relationship, and treat `dtstart` in an event as **optional**. However, if `dtstart` is present, certain other properties become required (as described in the even specification).

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:
- `INV_CAL` if Calendar object contains an invalid property
- `INV_EVENT` if an event component contains an invalid property
- `INV_ALARM` if an alarm component component contains an invalid property

For example, a Calendar object can contain the following properties: `prodid`, `version`, `calscale`, and `method`. The last two are optional, but can occur only once. As you parse Calendar properties, search the property list before adding each property to it. If the file you are parsing contains contains `calscale` twice, you will find the first `calscale` in the list before adding it again and return `INV_CAL` (after freeing all memory, of course).

Part 2: front end

Create a simple command-line interface in C in a file called `A2main.c`. It will use your calendar parser library and allow the user to read/display/create calendars. It will be a menu system with the following options:

1. Read in an iCalendar file using `createCalendar`. Any errors that occur must reported in humanly readable form by the main UI program. If reading of a file fails, the program must display the error and ask the user to either enter a new filename or exit.
2. Display the file on the screen using `printCalendar`
3. Create a new Calendar object. To keep things simple, this calendar object will contain one event with one alarm. make sure all the required properties are filled in. You can implement a UI for adding optional properties, but you don't need to. Make sure the calendar object is **validated** before you finish creating it.
4. Save the new calendar object to a file using `writeCalendar`. The user interface should not allow the user to overwrite existing files without an explicit confirmation from the user.
5. Exit. If any calendar objects are present in memory, they must be deleted using `deleteCalendar.`

Assignment structure

The submission must have the following directory structure:

`assign2/`	- contains the README file and the `Makefile`
`assign2/bin`	- should be empty, but this is where the Makefile will place the static lib files and the executable main file with the UI
`assign2/src`	- contains `CalendarParser.c`, `LinkedListAPI.c`, and `A2main.c`
`assign2/include`	- contains `CalendarParser.h`, `LinkedListAPI.h`, and your additional headers

Makefile

You will need to provide a `Makefile` with the following functionality:
- `make list` creates a static library `libllist.a` in `assign2/bin`
- `make parser` creates a static library `libcparse.a` in `assign2/bin`
- `make A2main` creates a static library `A2main` in `assign2/bin`
- `make` or `make all` creates `libllist.a` , `libcparse.a`, and `A2main` in `assign2/bin`
- `make clean` removes all `.o` and `.so` files, as well as `A2main`

Evaluation

Your code must compile, run, and have all of the specified functionality implemented. Any compiler errors will result in the automatic grade of **zero** (0) for the assignment.

Marks will be deducted for:
- Incorrect and missing functionality
- Deviations from the requirements
- Run-time errors
- Compiler warnings
- Memory leaks
- Bad / inconsistent indentation
- Bad variable names
- Insufficient comments
- Failure to follow submission instructions

Submission

Submit your files as a Zip archive using Moodle. File name must be be `A2FirstnameLastname.zip`.

Late submissions: see course outline for late submission policies.

This assignment is individual work and is subject to the University Academic Misconduct Policy. See course outline for details)