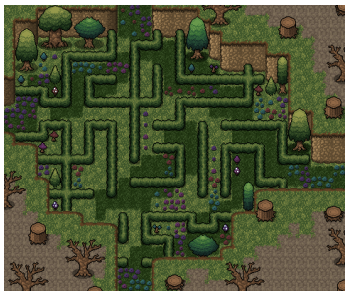
A2: Adventure



In this assignment, you will develop a *text adventure game* (TAG), also known as *interactive fiction*. The characteristic elements of TAGs include gameplay driven by exploration and puzzle-solving, and a text-based interface in which users type natural-language commands and the game responds with text. The seminal work in this genre is the Colossal Cave Adventure, which you can play online.

You will actually implement not just a single game, but a *game engine* that could be used to play many *adventures*. The game engine is an OCaml program that implements the gameplay and user interface. An adventure is a data file that is input by the game engine and describes a particular gaming experience: exploring a cave, hitchhiking on a spaceship, finding the missing pages of a powerful magical book, etc. This factoring of responsibility between the engine and input file is known as *data-driven design* in games.

This assignment is about the same difficulty as A1. I had originally planned a more difficult assignment. But because of the overlap with the Prelim, I have scaled back the scope. On a similar assignment last fall, students reported working a mean of 11.1 hours and a standard deviation of 4.0 hours. Please get started right away and make steady progress each day. Please track the time that you spend. We will ask you to report it.

Collaboration policy: This assignment is to be completed as an individual. You may have only limited collaboration with other students as described in the syllabus.

What you'll do: Implement and test a few OCaml modules.

Objectives:

- Design your own data types.
- Work with lists and trees.
- Use pattern matching and higher-order functions.
- Work with modules and compilation units.
- Read information from files, and interact with the user.
- Learn about JSON, a widely-used data format.

Table of contents:

FAQs

- **Q:** Do I have to check for validity of adventure files?

  A: Validity is a precondition of `from_json`. You've learned that when a precondition is violated a function may do anything it wants, including set the computer on fire.

- **Q:** I added helper functions. How do I unit test them?

  A: You don't. See the paragraph headed "Testing and interfaces" below.

- **Q:** How do I test that `from_json` is returning the right value?

  A: Its type is abstract, so you can't directly test it. Instead, test the result of applying other functions to that value. See the paragraph headed "Testing and interfaces" below.

- **Q:** I really want to build extra functionality to make the game more interesting, but it would require changing the provided specifications. May I do that?

  A: Awesome! But, no, sorry. We still need to be able to autograde your assignment. In past semesters we've had a second part to this assignment that permitted such changes, but with the compressed academic calendar this semester I regret that won't be possible.

- **All the FAQs from previous assignments still apply.**

Regarding the List module

Because of the particular focus on lists and trees in this assignment, we're going to pay unusual attention to how you use (or don't use) the `List` module from the standard library. Here are two coding practices to keep in mind from the beginning:

1. Do not reimplement a list function from the standard library. Use the library's version.

2. Do not use any of `List.hd`, `List.tl`, or `List.nth` on this assignment. This is the last assignment on which those will be prohibited. The purpose of that prohibition is guide you toward better solutions. Those functions might seem attractive at first, but you need to get used to coding without them. Note that, by the previous point, you cannot get around the prohibition by reimplementing them yourself under a different name.

Step 1: Get Git Going and Explore the Release Code

Create a new git repo for this assignment. **Make sure the repo is private.** Add the release code to your repo. Refer back to the instructions in A1 if you need help with that.

Now that we are using modules and creating larger programs, working in utop becomes difficult; your normal mode of interaction with OCaml is necessarily going to shift away from utop and toward VS Code and the command line. Nonetheless we do still provide a simple `make` target that will open utop with all your code available for use.

Here is a summary of all the Makefile targets:

- `make`: rebuild your code and launch utop with that code available
- `make build`: just rebuild your code
- `make test`, `make check`, `make finalcheck`, `make clean`: as usual
- `make docs`: build HTML documentation with Ocamldoc
- `make play`: launch your game interface
- `make zip`: create a ZIP file for CMS submission

The latter two targets won't be used until much later in the assignment; we discuss them below where they become relevant.

Feel free to browse through the release code at this point, but don't worry about familiarizing yourself with all of it yet. The steps of the assignment, below, will take you through them in a guided order.

Documentation. With so many files now in the code base, it will be helpful to browse HTML documentation instead of source code. The makefile target `make docs` will use Ocamldoc (similar to Javadoc) to build that for you. Running it will produce two directories of documentation, `doc.public` and `doc.private`. Open `index.html` from either of those directories in your web browser to see the documentation.

The **public** HTML documentation includes only the names exposed through the interface files. Since those files were provided to you, they have been fully documented already. The **private**

Since those files were provided to you, they have been fully documented already. The **private** HHTML documentation includes all the names in the implementation files. If a name is documented in both the `.mli` and `.ml` files, then in the public HTML documentation it will contain only the comment from the `.mli` file; whereas in the private HTML documentation, it will contain **both** the comments from the `.ml` file and `.mli` files merged together. That means you can add information to comments for maintainers, but clients won't see it. That also means you should **not** copy comments from the `.mli` file into the `.ml` file: they are automatically added by Ocamldoc when you generate the HTML documentation, so copying them would lead to the comment showing up twice in the private HTML documentation.

Do this now:

1. Run `make docs`. Note that is plural: "docs" not "doc".
2. If you're running on Ugclinux: right-click on `doc.public` in the Explorer pane of VS Code and download it locally.
3. Open `doc.public/index.html` in your local web browser.
4. Browse to the documentation for the `Author` module. Read it.
5. Open `author.mli` and `author.ml`. For each comment in the source files, figure out whether it's showing up in the HTML, and where.

Check your understanding:

- What is the difference between `(**)` and `(** *)`?
- What effect does `[ ]` have inside `(** *)` in the HTML?

Finally:

- Set your name and NetID in the appropriate comment in `author.mli`.
- Regenerate the documentation and observe the result.
- Make a mental note that the Ocamldoc manual has the syntax of Ocamldoc comments in case you want to write fancy comments of your own.

Step 2: JSON Tutorial

The adventure files that your game engine will input are formatted in JSON, the widely-used JavaScript Object Notation. If you've never used JSON before, read the brief overview of it on the JSON webpage.

In OCaml, you can use the Yojson library's Basic module for parsing JSON. That library is large and provides much more functionality than you need for this assignment. So we provide a small tutorial in release-code file `json_tutorial.ml`.

Do this now: read the tutorial, following along and entering lines in utop to experience it firsthand.

Then answer the following questions to check your understanding:

- What is a polymorphic variant? How does it differ from a parameterized variant?
- What are the two functions you can use to input JSON from a file vs. from a string?
- What is the OCaml data structure that corresponds to a JSON object? What OCaml library provides useful functions for that data structure?
- What is the Yojson function you would use to extract a string from a `Yojson.Basic.t` value? What would happen if that value were not actually a string?
- What is the `{| ... |}` syntax in OCaml? Why is it useful?

(There's no need to turn in your answers.)

Caution in case by chance you stumble across the chapter in the book Real World OCaml *about JSON:* ***Ignore it.*** *The features used in that chapter are more complicated than you need, and will be more confusing than helpful for this assignment. The ATDgen library and all of that chapter are not permitted for use on this assignment, because using them would preclude some of the list and tree processing that we want you to learn from this assignment. Note that the Core library used in that book is not supported in this course and will cause your code to fail* `make check`.

Step 3: Load Adventure Files

The gameplay of TAGs is based on an *adventurer* moving between *rooms*. Rooms might represent actual rooms, or they might be more abstract—for example, a room might be an interesting location in a forest. Rooms have named *exits* through which the adventurer may move to other rooms. The human *player's* goal is to explore the rooms.

Adventure files are formatted in JSON. We provide a couple example adventure files (`lonely_room.json`, `ho_plaza.json`) in the release code. Take a look now to familiarize yourself with them. An adventure file contains these entries:

- The rooms. Each room contains these entries:
  - an identifier,
  - a description of the room, and
  - the exits from the room. An exit itself contains two entries:
    - the name of the exit, and
    - the identifier of the room to which it leads.
- The identifier of the starting room, where the adventurer begins.

Note that JSON strings are case sensitive and may contain whitespace. Unfortunately JSON does not support multiline strings. That's why the descriptions in one of those examples necessarily violate the 80-column limit. Likewise, any test JSON files you create might need to do the same, and that's okay.

Your task: Implement and test the `Adventure` compilation unit provided in the release code. Use test-driven development (TDD) as in A1: write a unit test that fails, then write code to make the test pass, then polish the code. Keep doing that until you are convinced that your unit tests are sufficient to demonstrate that your code is correct and complete. All your tests should be in the file `test.ml`, which is provided in the release code. The `make test` target will run your test suite from that file.

The `Adventure` documentation mentions *set-like lists*. A set-like list is a list in which no element appears more than once, and in which order is irrelevant. So `[1; 2; 3]` and `[3; 2; 1]` are both set-like lists and are considered equivalent, but `[1; 1; 2; 3]` is not a set-like list. The starter code provided in `test.ml` contains a couple helper functions for tests involving set-like lists. **Tip:** make sure that, anywhere a function specification says it returns a set-like list, you remove any duplicates that might be in the list. Otherwise, it is not a set-like list and will fail the staff test cases. The function `List.sort_uniq` is a good way to remove duplicates.

The `Adventure` documentation also mentions *valid* JSON adventure representations. A JSON representation of an adventure is valid if and only if:

- The JSON complies with the description given above, as well as the examples provided in the release code. More precisely, the JSON must match the *schema* provided in `schema.json` in the release code. The schema specifies what the required components of the JSON are, as well as their names and JSON types. Using a JSON schema validator, you can check the well-formedness of any JSON again the schema. That could be useful in developing your own unit test cases.
- Every room has a unique identifier.
- Every exit from a given room has a unique name.
- Exit names contain only alphanumeric (A-Z, a-z, 0-9) and space characters (only ASCII character code 32; not tabs or newlines, etc.).
- No exit name contains any leading or trailing whitespace. Internally only a single space is permitted between each word (i.e., consecutive sequence of non-space characters).
- The target of every exit actually exists. That is, the room identifier to which the exit

purportedly leads is, in fact, the identifier of a room in the file.

- The starting room actually exists.

Note that validity is a precondition, not postcondition, of `Adventure.from_json`. Recall what you have learned about preconditions: when they do not hold, a function is free to do anything it wants, including setting the computer on fire.

Although it is not technically part of the definition of "valid", we promise that in our testing of your submission the adventures we use will not be huge. There will be at most on the order of magnitude of 100 rooms, and each room will have at most on the order of magnitude of 100 exits. That means tail recursion generally will not be needed.

Testing and interfaces. You cannot and should not test anything that is not exposed in an interface. That includes the definitions of abstract types as well as helper functions. For example:

- Don't test whether a value of type `Adventure.t` is really the "right" value, which would require you to know how it's defined in `adventure.ml`. You don't get to know that; it's encapsulated—and that's a good thing. Instead, test whether the functions you can apply to it (`start_room`, `room_ids`, etc.) return the right values.
- Don't add helper functions to an interface and test them: they're not meant to be exposed to clients. Instead, test the functions already in the interface that use them.

For the latter, you might argue that you really do want to test helper functions. The designers of OUnit would disagree with you. Here's what their manual says:

> *"Test only what is really exported: on the long term, you have to maintain your test suite. If you test low-level functions, you'll have a lot of tests to rewrite. You should focus on creating tests for functions for which the behavior shouldn't change."*

This is the stopping point for a satisfactory solution.

Step 4: Parse Commands

The interface to a TAG is based on the player issuing text *commands* to a *prompt*, the game replies with more text and a new prompt, and so on. Thus, the interface is a kind of read-eval-print-loop (REPL), much like `utop`. For this assignment, commands will be phrases of the form `<verb> <object>`. Verbs are always a single word, whereas objects might consist of multiple words separated by spaces.

There are only two verbs your engine needs to support:

- **go:** The player moves from one room to another by with the verb "go" followed by the name of

an exit.

- **quit**: The player exits the game engine with this verb, which takes no object.

Commands are case sensitive, as are exit names. So whereas `go clock tower` would move the player from Ho Plaza to McGraw Tower in the sample adventure file, neither `GO clock tower` nor `go Clock Tower` would.

Your task: Implement and test the `Command` compilation unit. The non-deprecated functions in the standard library `String` module are perfectly adequate for the work you need to do. *Hint:* investigate `String.split_on_char`.

Step 5: State Transitions

As the player progresses through an adventurer, some information does not change: the rooms, their exits, and so forth. But other information does change: the player's current room, and the set of rooms the player has visited. In this assignment we'll keep track of the latter kind of information as part of the game *state*. In an imperative language, the game state would be a mutable variable that is changed by functions that implement the game. But in a functional language, the game state must instead be an immutable value. Which leads to the question: how to represent changes?

Looking back at A1's `step` function (which you might or might not have implemented), we can spot an answer: functions can take in an old state and return a new state. That's exactly the solution we'll use in this assignment. In particular, when the player attempts to move the adventurer from one room to another, the function that implements that movement will take in the current state of the game, and return a new state in which the adventurer has moved. Or perhaps the movement will turn out to be impossible, in which case the state will not change.

Your task: Implement and test the `State` compilation unit. Note carefully that the `State.go` function's specification does not permit it to print, which is intended to guide you toward an idiomatic and functional implementation.

This is the stopping point for a good solution. If you want to stop here, that's perfectly fine.

Step 6: Interface

At last, it's time to build the user interface and make the game playable. The requirements for the interface are relatively minimal:

- When the engine starts, the interface must prompt for the name of an adventure file to play. You must not hardcode the adventure file name. You must not change the name provided by the user in any way. Just leave it alone. Common mistakes in the past have include adding a path to the beginning of the name, or adding a `.json` extension to the end. Don't do either of those. You could lose points for changing the file name typed in by the player. Why? The player (or grader) might want to play an adventure file that is stored in a directory on their filesystem that you can't possibly predict.

- If the file entered by the user does not exist, the interface must print a helpful error message. The interface must not visibly show an exception to the user. (You wouldn't want to see an exception from your PS4 or Xbox.)

- The first time a player visits a room, the interface must print its description. Upon subsequent visits, the interface may repeat that description, print some shorter message, or may omit it. That is up to you.

- The interface optionally may print information about the exit names. Level designers might prefer that it not. The `no_plaza.json` example adventure includes an Easter egg based on that.

- If the player attempts to move illegally—that is, to an exit that does not exist in the current room— then the interface must display an error message of your choice, then prompt for a new command.

- If the player issues the quit command, the interface must print a farewell message of your choice, then terminate without any exceptions or error messages from the operating system. That can be implemented simply by allowing all functions to return, or with the expression `exit 0`. (The function `Stdlib.exit` terminates the running process, and the `0` return code indicates a normal termination.)

- If the player issues a command that cannot be understood, the interface must print an error message of your choice, then prompt for a new command.

We leave the rest of the design of the user interface up to your own creativity. In grading, we will not be strictly comparing your user interface's text output against expected output, so you have freedom in designing the interface.

The Makefile contains a new target, `make play`, that will build your game engine and launch the interface.

Your task: Implement the `Main` compilation unit. Your user interface must be implemented entirely within `main.ml`. It may not be implemented in `state.ml`. As the specification of `State.go` says, that function may not have any side effects, especially not printing.

All the console I/O functions you need are in the `Stdlib` module. The output functions under the heading "Output functions on standard output" are sufficient for this assignment. The `read_line` function is what you should use for input. You're welcome to investigate the `Printf` and `Scanf` modules, but they are overkill for this assignment. You will likely find the `String.concat` function useful in manipulating object phrases.

The `Main` compilation unit is the only part of this assignment for which you are not required to write unit tests. Instead, you should interactively *playtest* your interface. You would likely find it helpful to recruit a friend to play your game and observe what confuses them about the interface. It's fine for that friend to be another 3110 student as long as you adhere to the "Limited Collaboration" policy, but you might find a non-programmer to be a better test subject.

Step 7: New Adventure

Create your own adventure by constructing your own JSON file. It must not be based on any sample files we have already given you. We encourage you to create an interesting and creative adventure! But your grade won't be based on that. Instead, we simply require that it have at least five rooms. If you are stuck for ideas, consider trying to model your dorm at Cornell, or the set of Friends or The Office. Name your file `new.json` so that the grader can quickly find it.

Rubric

- 25 points: submitted and compiles
- 25 points: satisfactory scope
- 25 points: good scope
- 10 points: testing
- 10 points: code quality
- 5 points: excellent scope

Testing Rubric. The graders will continue to assess these aspects of your test suite:

- Whether your tests have descriptive names.
- Whether your tests are constructed with helper functions to minimize duplicated code.

Your test suite additionally needs to satisfy these requirements:

- You have at least one unit test for each of the required functions in the `Adventure` interface, excluding `from_json`. Function `from_json` cannot be tested independently; instead, it will be involved in testing each of the other six required functions.

- You have at least four unit tests for `Command.parse`: one each for `Go`, `Quit`, raising `Empty`, and raising `Malformed`.
- You have at least two unit tests for `State.go`: one for a legal result, another for an illegal result.

It would be wise to unit test much more extensively than that, of course, but you will get full credit on testing for doing the above.

As usual:

- Make sure your test suite can be run with `make test`.
- Your test suite will be graded independently from the rest of your implementation. So you can gain or lose points for it regardless of which functions you complete.
- You may violate the 80-column rule for long string literals in test cases.

Code Quality Rubric. The graders will assess the following aspects of your source code:

- Everything that was already assessed on A1.
- Value identifiers use `snake_case` as before; constructor and module identifiers use `CamelCase`.

- You have used `List` library functions rather than rewriting them yourself.

- You have not used `List.hd`, `List.tl`, or `List.nth`.

Excellent Scope Rubric. The graders will attempt to play your game by checking the following:

- Entering the name of an adventure file, which might be located anywhere in their filesystem—so again, do not modify what is entered by prepending or appending anything to it or changing it in any way.
- Moving through some rooms to check that the engine correctly prints their descriptions.
- Attempting to move to an illegal room and checking that an error message is displayed.
- Entering the quit command and checking that the game engine does not produce an exception or error message.
- Entering malformed commands and checking that the game engine does not produce an exception or otherwise misbehave, but, instead informs the player of their error.

The new adventure will be worth about 1 out of the 5 points. You won't get that point if your game interface is not implemented.

Submission

Record your name and NetID in `author.mli`, and set the `hours_worked` variable at the end of `author.ml`.

Run `make zip` to construct the ZIP file you need to submit on CMS. Our autograder needs to be able to find the files you submit inside that ZIP without any human assistance, so:

→ DO NOT use your operating system's graphical file browser to construct the ZIP file. ←

Use only the `make zip` command we provide. Mal-constructed ZIP files will receive a zero from the autograder. If CMS says your ZIP file is too large, it's probably because you did not use `make zip` to construct it; the file size limit in CMS is plenty large for properly constructed ZIP files.

For the excellent scope: as long as your new adventure file is named with a `.json` extension and is located in the same directory as the rest of your source code, it should automatically be included in your ZIP file. But double check just to be sure. We're not going to accept any regrade requests or restore any points if it's missing from your submission.

Ensure that your solution passes `make finalcheck`. Submit your `adventure.zip` on CMS. Double-check that the MD5 sum is what you expected. Re-download your submission from CMS and double-check before the deadline that the contents of the ZIP are what you intended.

Congratulations! You've had an Adventure!

Acknowledgement: Adapted from Prof. John Estell (Ohio Northern University).