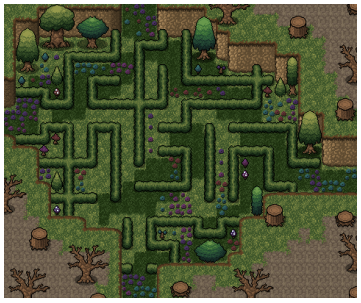
A2: Adventure



In this assignment and the next, 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.

In A2, you will build a pre-specified part of the game: exploration of a map. In A3, you will have the opportunity to make your game unique by extending your A2 solution with functionality of your own choice.

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. On a similar assignment last year, the mean hours worked was 9.5, and the standard deviation was 3.9. 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.

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

Objectives:

- Design your own data types.
- Work with lists and trees.
- Use pattern matching and higher-order functions.

- Read information from files, and interact with the user.
- Learn about JSON, a widely-used data format.

Table of contents:

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.

This is the first assignment in which the release code contains interface (`.mli`) files. See what the FAQ has to say about them.

Now that we are using modules and creating larger programs, working in utop becomes a little more 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 provide a simple `make` target that will open utop with all your code available for use. That command first builds all your code, then loads utop and runs all the commands in `.ocamlinit`, which is a file provided in the release code. Note that after changing any code, you must exit utop and re-run `make` for your changes to be reflected in utop.

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`: has changed; see the Documentation paragraph just below

- `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. We are now working with compilation units, which have two pieces: interfaces (`.mli`) and implementations (`.ml`). The documentation we produce will now also have two pieces:

- documentation for *clients* of our code base—the people who only need to understand the functions and other names exposed through the interface files, and
- documentation for *maintainers* of our code base—the people who need to understand all of the code, including the implementation files.

So the `make docs` command now produces two directories of documentation, `doc.public` and `doc.private`. The "public" documentation is for clients; the "private", for maintainers.

The **public** 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** 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 documentation it will contain only the comment from the `.mli` file; whereas in the private 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 the comments from the `.mli` file into the `.ml` file: they are automatically added by Ocamldoc when you generate the documentation.

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. In the release code, we provide a small tutorial on Yojson in the file `json_tutorial.ml`, which uses the file `cornell.json` as an example input. Although the Yojson library is large and provides a lot of functionality, all that you need to know for this assignment is covered by the tutorial.

Your task: 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: There is a chapter on JSON in Real World OCaml*, but you should 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 tool at the end 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.

Your task: Implement and test the `Adventure` compilation unit provided in the release code. Remember to use test-driven development: write a unit test that fails, then write code to make the test pass; 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` can be helpful.

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`, therefore your implementation is not required to check for validity. Consequently, in grading your submission we will never pass invalid adventures to that function. (If we did, your function would be free to do anything it wanted, including set our grading computers on fire.)

Furthermore, 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.

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. Or, if you want to get a head start on A3, keep going! The excellent scope of A2 will be part of the satisfactory scope of A3—so, one way or the other, you'll end up doing it.

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 prompts for the name of an adventure file to play. You may not hardcode the adventure file, nor assume anything about its name. In particular:
    - You may not assume that the adventure is in the current directory.
    - You may not assume that the adventure file ends in `.json`.
    - You may not assume that the adventure file even exists.

Pay attention to the above statements: you will lose points for changing the file name typed in by the player. Don't add any kind of path to the beginning. Don't add a file extension. Just leave it alone.

- Before prompting for a command, the interface always prints the description of the room in which the player is currently located. The interface does not need to print information about the exit names. (Indeed, level designers might prefer that it not. The `ho_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 displays an error message of your choice, then prompts for a new command.

- If the player issues the quit command, the interface prints a farewell message of your choice, then terminates 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 prints an error message of your choice, then prompts 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.

Scope

- 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. Here is what the graders will assess for your OUnit test suite:

- You have at least one unit test for each of the seven required functions in the `Adventure` interface.
- 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.

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.
- Your code is easily readable; it is neither too dense nor too verbose.
- **Documentation:** The graders will run `make docs` and read your extracted public and

private HTML documentation. Note that what we will read is the HTML, not the original source code comments. If you have added any additional names to interfaces, they must be documented with a specification comment. We also expect to see documentation for every name in the private documentation, including helper functions. We do not want to see duplicated comments showing up: so, don't copy comments from the `.mli` files into the `.ml` files.

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 anyway).
- 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.

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 penalty of 15 points because of the extra human processing they will require. 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.

Ensure that your solution passes `make finalcheck`. Submit your `adventure.zip` on CMS. Double-check before the deadline that you have submitted the intended versions of your file.

Congratulations! You've had an Adventure!

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