Project 1

Due: Feb 13, before midnight.

Important Reminder: As per the course Academic Honesty Policy, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. It then hints at how these requirements can be met. Finally, it describes how it can be submitted.

Aims

The aims of this project are as follows:

- To encourage you to use regular expressions to implement a trivial scanner.
- To make you implement a recursive-descent parser for a small language.
- To use JSON to represent the results of parsing.

Requirements

Use the implementation of either `java`, `node` or `python3` available on `remote.cs` to implement a parser for the following language of which is a subset of the syntax used for elixir data literals. (You may also use a language like TypeScript which compiles to one of these languages as long as you provide all the steps needed to compile that language to one of the above languages).

- A sentence in the language consists of a sequence of zero-or-more **data-literal**'s.
- A **data-literal** is either a **list** literal, a **tuple** literal, a **map** literal, or a **primitive** literal.
- A **primitive** literal is either an **integer**, **atom** or **boolean**.
- A **list** literal is a sequence of 0-or-more comma-separated **data-literal**'s within square brackets `[` and `]`.
- A **tuple** literal is a sequence of 0-or-more comma-separated **data-literal**'s within braces `{` and `}`.
- A **map** literal is a sequence of 0-or-more comma-separated **key-pair**'s within a decorated left brace `%{` and a regular right brace `}`.
- A **key-pair** is either a sequence consisting of a **data-literal**, a right-arrow `=>` followed by a **data-literal**, or a sequence consisting of a **key** followed by a **data-literal**.
- An **integer** consists of a sequence of one-or-more digits, possibly containing **internal** underscores `_`.
- An **atom** consists of a colon `:`, followed by an alphabetic character or underscore `_` followed by a sequence of zero-or-more alphanumeric characters or underscores `_`.
- A **key** is just like an atom but the `:` must be at the end instead of the start.
- A **boolean** is one of the words `true` or `false`.

Whitespace and `#`-to-end-of-line comments should be ignored.

Note that a map literal of the form `%{ key: 22 }` is syntactic sugar for `%{ :key => 22 }`.

You will specifically need to submit a zip-archive which unpacks into a `prj1-sol` directory minimally containing a file `elixir-data.ebnf` plus two shell scripts `make.sh` and `run.sh`:

1. `elixir-data.ebnf` must contain a grammar for the above language using the EBNF notation described in class.

2. Running `make.sh` from any directory should build any artifacts needed to run your program within the `prj1-sol` directory.

3. Running `run.sh` from any directory should read and parse a sentence of the above language from standard input and output on standard output a single line containing the JSON representation of the parse.

   Different kinds of data literals should be output as a JSON object having two properties:

   `%k`
   : The kind of the literal, as defined below.

   `%v`
   : The value of the literal, as defined below.

   The top-level JSON should consist of a JSON list containing the JSON representations of the top-level **data-literal**'s read from standard input.

   The JSON representation of the different kind of literals should be as follows:

   - An **integer** has the JSON representation `{ "%k": "int", "%v":` *value* `}` where *value* is a JSON integer respresenting the value of the integer. For example, the **integer** `123` should have the JSON representation `{ "%k": "int", "%v": 123 }` .

   - An **atom** has the JSON representation `{ "%k": "atom", "%v":` *value* `}` where *value* is a JSON string spelling out the atom. For example, the **atom** `:_a32` should have the JSON representation `{ "%k": "atom", "%v": ":_a32" }` .

   - A **boolean** has the JSON representation `{ "%k": "bool", "%v":` *value* `}` where *value* is a JSON boolean representing the value of the **boolean**. For example, the boolean `true` should have the JSON representation `{ "%k": "bool", "%v": true }` .

   - A **key** has the JSON representation `{ "%k": "atom", "%v":` *value* `}` where *value* is a JSON string spelling out the key lexeme, but with the `:` moved to the front. For example, the **key** `abc:` should have the JSON representation `{ "%k": "atom", "%v": ":abc" }` .

   - A **list** has the JSON representation `{ "%k": "list", "%v":` *value* `}` where *value* is a JSON list containing the JSON representations of the individual items in the **list**. For example, the list `[ 1, 2 ]` should have the JSON representation:

     ```
     { "%k": "list",
       "%v": [
          { "%k": "int", "%v": 1 },
          { "%k": "int", "%v": 2 }
       ]
     }
     ```

   - A **tuple** has the JSON representation `{ "%k": "tuple", "%v":` *value* `}` where *value* is a JSON list containing the JSON representations of the individual items in the **tuple**. For example, the tuple `{ 1, :k }` should have the JSON representation:

```
{ "%k": "tuple",
  "%v": [
     { "%k": "int", "%v": 1 },
     { "%k": "atom", "%v": ":k" }
  ]
}
```

- A **map** has the JSON representation `{ "%k": "map", "%v":` *value* `}` where *value* is a JSON list containing the 2-element JSON lists representing the individual elements in the **map**. For example, the map `%{ :a => 22, b: 33 }` should have the JSON representation:

```
{ "%k": "map",
  "%v": [
     [ { "%k": "atom", "%v": ":a" },
       { "%k": "int", "%v": 22 }
     ],
     [ { "%k": "atom", "%v": ":b" },
       { "%k": "int", "%v": 33 }
     ]
  ]
}
```

The JSON output should consist of a single line without any whitespace other than the newline terminator. The members of a JSON object may be output in any order.

If there are errors in the content, the program should exit with a non-zero status after detecting the first syntax error. It should output a suitable error message on standard error.

An annotated log of the running project and the provided tests should help clarify the above requirements.

Rationale for the Requirements

The requirements are based on the following rationale:

- The specified language is a simple language containing a subset of Elixir data literals. Implementing a parser for this language allows you to understand the basic principles of scanning and recursive-descent parsing.
- The `make.sh` and `run.sh` scripts allow automatic testing of your project without needing to know the details of your implementation language. The former allows the testing program to run any compilation step required by your implementation language and the latter allows the testing program to run the project.

Provided Files

The prj1-sol directory contains starter shell scripts for the two scripts your submission is required to contain as well as a template README which you must complete and include with your submission.

The extras directory contains auxiliary files associated with the project, including files which can help with testing your project.

tests
A directory containing tests. There are three kinds of test files:

`*.test`
A test input for which your program is expected to succeed.

`*.out`
The expected pretty-printed output for a successful test of the corresponding `*.test` file.

`*.err`
: A test input for which your program should fail.

do-tests.sh
: A shell script which can be used for running the above tests. It can be invoked from any directory and takes up to two arguments:

1. The path to your `run.sh` shell script.
2. An optional argument giving the path to a single test to be run.

If invoked with only a single argument, the script will run all tests. If invoked with a second argument, then it will run only the test specified by that argument.

LOG
: A log file illustrating the operation of the project.

Standard Input, Standard Output, Standard Error

This project requires your program to read from standard input and write its output to standard output and write error messages to standard error. These are the three I/O streams which are initially available when a program starts up under any current OS:

Standard Input
: An input stream, initially set up to read from the console. This often corresponds to file descriptor 0.

Standard Output
: An output stream, initially set up to output to the console. This often corresponds to file descriptor 1.

Standard Error
: Another output stream, initially set up to output to the console. This often corresponds to file descriptor 2.

So you can use these streams without needing to open any file, as they are already open.

All popular languages provide access to these streams.

Python

- `sys.stdin`, `sys.stdout` and `sys.stderr` refer to the three streams.
- `sys.stdin.read()` will read from standard input until EOF.
- `print(...)` or `sys.stdout.write(...)` will print `...` to standard output (the former adds a newline).
- `sys.stderr.write(...)` or `print(..., file=sys.stderr)` will write `...` to standard error.

JavaScript nodejs

- `0`, `1` and `2` refer to the three streams and can be used wherever a file path is expected.
- `fs.readFileSync(0, 'utf8')` will read from standard input until EOF.
- `console.log(...)` or `fs.writeFileSync(1, ...)` will write `...` to standard output (the former adds a newline and has additional functionality).
- `console.error(...)` or `fs.writeFileSync(2, ...)` will write `...` to standard error (the former adds a newline and has additional functionality).

Java

Java defines `System.in`, `System.out` and `System.err` for the three streams; you can then use the smorgasbord of `java.io.*` classes to read/write the streams. The newer `java.nio.file` package provides more convenient APIs.

Using stdin within the Unix Shell

If a program is reading interactively from standard input, then it will freeze and wait for input to be provided on the terminal:

```
$ ./run.sh
%{a: 22 }
^D   #indicate EOF
[{"%k":"map","%v":[[{"%k":"atom","%v":":a"},{"%k":"int","%v":22}]]}]
$
```

The control-D is used to indicate EOF to the terminal controller.

It is much more convenient to use I/O redirection in the shell:

```
$ ./run.sh \
     < ~/cs571/projects/prj1/extras/tests/50-compound-data.test \
   | jq - S . > compound-data.json
```

The `\` escapes newlines to combine multiple physical lines into a single logical line; the `< .../tests/50-compound-data.test` redirects the contents of `50-compound-data.test` to the standard input of `run.sh`; the `| jq -S .` pipes the single line output of the program to `jq -S .` which pretty-prints the json on its standard output (`-S` sorts all object keys); finally, the `> compound-data.json` redirects the standard output of `jq` to `compound-data.json`.

Note that `run.sh` is totally unaware of the redirection; the shell takes care of setting up the standard input and output streams so that they are redirected to the files. For example, if `run.sh` is calling a python parser, then the python parser can continue using `sys.stdin` and `sys.stdout`.

Before Starting Your Project

Before starting this project, set up a symlink to the course repository under your home directory on `remote.cs`:

```
$ cd ~                      #ensure you are in your home directory
$ ln -s ~umrigar/cs571 .    #set up symlink
$ ls cs571                  #list files
```

You should see the top-level files and directories in the course repository.

Note that the above symlink will always reflect the current contents of the course repository.

It is also probably a good idea to set up a work directory on `remote.cs`.

```
$ mkdir -p ~/i571/submit
```

Then use the `~/i571` directory for any personal work for the course and the `~/i571/submit` directory for work to be submitted. For example, you might do the work for this project in `~/i571/submit/prj1-sol`.

If you are familiar with git, it is probably a good idea to create a git respository for your `~/i571` directory.

Hints

This section is not prescriptive in that you may choose to ignore it as long as you meet all the project requirements.

The following points are worth noting:

- Ideally, the implementation language for your project should support the following:
    - Support regex's either in the language or via standard libraries.
    - Easy support for JSON, ideally via standard libraries.

  Scripting languages like Python or JavaScript will probably make the development easiest.

- The requirements forbid extraneous whitespace in the JSON output which makes the output quite hard to read. To get around this, you can pipe the output through a JSON pretty-printer like `jq -S .` which is available on `remote.cs`.

- While developing the project, you will probably be running tests provided in the extras directory. It may be convenient to set up a shortcut shell variable in the shell you are using for developing your project.

```
$ extras=$HOME/cs571/projects/prj1/extras
# run a particular test
$ $extras/do-tests.sh ./run.sh $extras/tests/12-single-int.test
# run all tests
$ $extras/do-tests.sh ./run.sh
```

- The exit status of the last shell command is available in the shell variable `$?`. You can examine it using the command `echo $?`. This can be used to verify that your program exits with a non-zero status when given erroneous inputs.

- Note that calling `consume()` changes the lookahead token. So if you need the lexeme for a token, it should be grabbed from the lookahead before `consume()`ing that token.

You may proceed as follows:

1. Review the material covered in class on regex's, scanners, grammars and recursive-descent parsing. Specifically:
    a. Review the online parser to make sure you understand the gist of how arith.mjs works without getting bogged down in the details of JavaScript.
    b. Review specific arithmetic expression to JSON parsers implemented in different programming languages.
2. Read the project requirements thoroughly.
3. Ensure that you have set up your `remote.cs` account for as specified in the Before Starting Your Project section.
4. Copy over the starting code for your project (this assumes you have created a `~/i571/submit` directory):

```
$ cd ~/i571/submit
$ cp -r ~/cs571/projects/prj1/prj1-sol .
$ cd prj1-sol
$ ls  # list starting files
```

5. If using git, set up a .gitignore file suitable to your implementation language. Set it up to ensure that you do not commit binaries or cached files to git. Note that the project allows you to set up the `make.sh` script to have the TA build those files when grading your project.
6. Fill in your details in the `README` template. Commit and push your changes if using git.

7. Write an EBNF grammar for the data literals language. You should be able to do so by structuring your grammar based on the description of the language provided in the Requirements section.

   Once you are happy with the grammar, paste it in as a comment in one of your implementation files. Use the grammar to drive your code as per the recipes discussed in class.

8. Start work on your lexer. It is easiest to simply read the entire standard input into a string variable.

   You need to decide whether your lexer will accumulate all tokens into a list, or deliver them one-by-one as needed by the parser.

   - The former organization facilitates using unlimited lookahead in the parser; i.e. the parser can look ahead by several tokens in order to make parsing decisions.
   - The latter organization will require having the lexer track its position within the input text.

   As mentioned in class, minimally a token should have the following fields:

   `kind`
   : specifying the kind of token.

   `lexeme`
   : specifying the matched text.

   Additionally, you may want to track the position of the token within the input stream to facilitate error reporting.

   Depending on the implementation language used for your project, making the **`kind`** field a string equal to the **`lexeme`** field for all tokens having only one possible lexeme will make your parser more self-documenting.

   To produce the next token, the scanner can work as follows:

   a. Ignore whitespace and **`#`**-to-end-of-line comments, if any. Note that there could be a sequence of alternating whitespace and **`#`**-comments.

   b. Check whether the prefix of the text after the whitespace/comments matches a possible multiple character token. If yes, accumulate that token.

   c. Otherwise return the first character in the text as a single character token. This works particularly well if these tokens have the token **`kind`** set to the **`lexeme`**. This will allow any illegal characters to be delivered to the parser which has better context to report errors.

9. Use the techniques discussed in class to write a recursive descent parser for your constructed grammar. Note that the recipe provided for writing recursive descent parsers requires maintaining a "global" variable **`tok`** which contains the current lookahead token and a **`consume(kind)`** function which sets **`tok`** to the next token if its **`kind`** matches the parameter, and reports an error if that is not the case.

   [If using python3 as your implementation language, you will need to declare any "global" variable **`nonlocal`** in order to refer to it from within your parsing functions.]

   Most of the provided grammar can be handled easily using the recipe provided for recursive descent parsers.

   - A utility predicate which checks if the current lookahead token can start a data literal may be useful.

  - Your parser should attempt a non-terminal only if the current lookahead token can start that non-terminal, or if that non-terminal is the only remaining possibility.
  - When a parsing function returns successfully, ensure that the `tok` lookahead contains the token immediately after the tokens recognized by that function.
  - One advantage of hand-written recursive descent parsers is that it is possible to use arguments and return values of parsing functions. Specifically, have each parsing function return a value representing the phrase parsed by that function. Ensure that the return value can easily be converted to the required JSON output.

10. Convert the value returned by your parser to a JSON string without any whitespace and output to standard output followed by a newline.
11. Test your parser using the provided scanner tests:

    ```
    $ ~/cs571/projects/prj1/extras/do-tests.sh run.sh
    ```

    Debug any failing tests. Note that you can run a single test by adding an additional argument to the above command providing the path to the failing test.
12. Iterate until you meet all requirements.

If using git, it is always a good idea to keep committing your project periodically to ensure that you do not accidentally lose work.

Submission

You are required to submit a zip-archive to gradescope (you can access gradescope from the **Tools** menu on the brightspace navigation for this course).

Unpacking the archive should result in at least the following files:

```
prj1-sol/README
prj1-sol/elixir-data.ebnf
prj1-sol/make.sh
prj1-sol/run.sh
```

The unpacked `prj1-sol` directory should contain all other source files needed to build and run your project using `sh make.sh` followed by `sh run.sh`.

For example, a sample zip file for a project implemented in python contains:

```
$ unzip -l py/prj1-sol.zip
Archive:  py/prj1-sol.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
     1127  2024-01-08 03:54   prj1-sol/README
      406  2024-02-10 17:25   prj1-sol/elixir-data.ebnf
     5305  2024-01-08 14:33   prj1-sol/elixir-data.py
      198  2024-01-08 03:54   prj1-sol/make.sh
      242  2024-01-24 20:37   prj1-sol/run.sh
---------                     -------
     7278                     5 files
```

A do-zip.sh script has been created to facilitate creating the zip file. Please read the comments at the start of the script. A suitable .zipignore has been added to the directory containing the starter files. This `.zipignore` is suitable for a python or javascript project. If you have used java, then one is available in java-zipignore.

You should verify your zip file before submission. Simply unzip it into a `tmp` directory:

```
$ mkdir -p ~/tmp
$ cd ~/tmp
$ unzip PATH_TO_ZIP_FILE  # maybe something like ~/i?44/submit/prj1-sol.zip
```

This should **create** a `prj1-sol` directory in `~/tmp`, Go into `~/tmp/prj1-sol` and you should be able to successfully run `sh make.sh` followed by `sh run.sh`.

When you submit your zip file to gradescope, it will run automated tests:

1. Verify that all required files have been included.
2. Build your submission using `sh make.sh`.
3. Use your `run.sh` to Run some tests (note that the actual project grading may use additional tests).

If a step fails, then subsequent steps are aborted.

Important Note: Unfortunately, gradescope removes execute permissions when unpacking the zip archive. So you cannot set up your `run.sh` script to run interpeter files directly. So for example, a `run.sh` which contains the line `$dir/elixir-literal.mjs` will not work, you will need to use `node $dir/elixir-literal.mjs`. Similarly, for Python use `python3 $dir/elixir-literal.py` instead of `$dir/elixir-literal.py`.

References

- Example Parser from Wikipedia article on *Recursive descent parser*. Note that the grammar notation is slightly different:
  - `{` X `}` is used to indicate 0-or-more repetitions of X instead of `X*`.
  - `[ X ]` is used to indicate an optional `X` instead of `X?`.

  The parser uses `accept()` and `expect()` instead of our `peek()` and `consume()`. The semantics of the routines are slightly different: they get the next token in `accept()`, whereas we get the next token in `consume()`.
- Grammars and Parsing, discusses building ASTs. The `peek()` and `consume()` routines described there are exactly equivalent to our `peek()` and `consume()` routines.