1 Project 1

Due: Feb 18, before midnight.

Important Reminder: As per the course *Academic Honesty Statement*, 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.

1.1 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 token lists and abstract syntax trees.

1.2 Requirements

Use any programming language available on `remote.cs` to implement a recognizer for a ifdef language with directives similar to the conditional compliation directives in the C-preprocessor. Specifically, update your github repository with a directory `submit/prj1-sol` such that:

1. Typing `./make.sh` within that directory will build any artifacts needed to run your program.
2. Typing `./scan.sh` within that directory will read text from standard input and output on standard output a JSON list of tokens for the ifdef language followed by a newline. The tokens of the ifdef language are defined below.
3. Typing `./parse.sh` within that directory will read text from standard input and output on standard output a JSON list of AST's as defined below.

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 error message on standard error.

1.2.1 Lexical Requirements

The lexical requirements for the ifdef language is defined by the following rules:

- A **line** is a maximal sequence of characters not containing a newline `'\n'` character.
- **Linear whitespace** is a maximal sequence of characters consisting of space `' '` or tabs `'\t'`.
- `IFDEF`, `IFNDEF`, `ELIF`, `ELSE` and `ENDIF` tokens are `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif` occurring at the start of a line optionally preceeded by linear whitespace. A line starting with such a token is referred to as a **directive line**.
- A `SYM` is a maximal sequence of characters starting with an alphabetic character or underscore and continuing on with zero-or-more alphanumerics or underscores. They are only recognized after linear whitespace following `IFDEF`, `IFNDEF` and `ELIF` tokens.
- The rest of a directive line following the starting token and possible `SYM` token is ignored.
- `TEXT` tokens consists of a maximal sequence of lines none of which are directive lines.

The `scan.sh` script you are required to implement should output a JSON list of tokens on standard output followed by a newline. Each token should be output as a pair: a 2-element JSON list whose first element is the token kind as defined above and second element is the token *lexeme*, i.e. the text which the token matched.

For example, the following input from scan0.defs:

```
  Some text
  more text
    #else stuff ignored
    #ifndef _123 ignored
    # ifdef text line
    #elif 2222 note 2222 not a sym
  more text
  and more
```

should result in the output:

```
[
   [
      "TEXT",
      "  Some text\n  more text\n"
   ],
   [
      "ELSE",
      "#else"
   ],
  [
    "IFNDEF",
    "#ifndef"
  ],
  [
    "SYM",
    "_123"
  ],
  [
    "TEXT",
    "    # ifdef text line\n"
  ],
  [
    "ELIF",
    "#elif"
  ],
  [
    "TEXT",
    "  more text\n  and more\n"
  ]
]
```

Note that the above is a pretty-printed version of the output; the actual output should not contain any non-significant whitespace other than the terminating newline.

1.2.2 Syntactic Requirements

The `parse.sh` script should extract the structure defined by the following EBNF grammar from its standard input:

```
source
  : (TEXT | ifdef)*
  ;
ifdef
  : (IFDEF | IFNDEF) SYM
    source
    (ELIF SYM source)*
    (ELSE source)?
    ENDIF
  ;
```

The grammar uses the usual EBNF notation covered in class.

The output of `parse.sh` should be the JSON representation of an AST defined as follows:

- A `source` non-terminal should result in a JSON list of ASTs corresponding to its `TEXT` or `ifdef` consituents.
- The JSON corresponding to a `TEXT` token should be a JSON object `{ "¬ tag": "TEXT", "text":` *TXT* `}` where *TXT* is the lexeme for the `TEXT` token.
- The JSON corresponding to a `ifdef` non-terminal should consist of a JSON object `{ "tag":` *TAG*`, sym:` *SYM*`, xkids:` *KIDS* `}` where *TAG* is either `IFDEF` or `IFNDEF` depending on the first token, *SYM* is the lexeme of the `SYM` token, and *KIDS* is a JSON list which is a concatenation of:
  1. The list of AST's corresponding to the first `source` non-terminal on the RHS of the rule for `ifdef`.
  2. A list of the ASTs for the `ELIF` sections where the AST for a `ELIF` section is a JSON object of the form `{ "tag": "ELIF", sym:` *SYM*`, xkids:` *ELIF_KIDS* `}` where *SYM* is the lexeme of the `SYM` token, and *ELIF_KIDS* is the list of AST's for the second `source` non-terminal on the RHS of the rule for `ifdef`.
  3. If `ELSE` is present, then a single element list containing the AST for the `ELSE` section which consists of `{ "tag": "ELSE", xkids:` *ELSE_KIDS* `}` where *ELSE_KIDS* is the list of AST's for the third `source` non-terminal on the RHS of the rule for `ifdef`.

For example, the input:

```
Some text
  more text
#ifdef sym123
  # ifdef text
  #elif _123
     elif text 1
  #ifndef some_sym
  # ifndef text
  #endif
  #elif _345
     elif text 2
  #else ignored text
    # else text
  #endif
  Some more text
```

should result in the output:

```
[
   {
      "tag" : "TEXT",
   "text" : "Some text\n  more text\n"
},
{
   "sym" : "sym123",
   "tag" : "IFDEF",
   "xkids" : [
      {
         "tag" : "TEXT",
         "text" : "  # ifdef text\n"
      },
      {
         "sym" : "_123",
         "tag" : "ELIF",
         "xkids" : [
            {
               "tag" : "TEXT",
               "text" : "     elif text 1\n"
            },
            {
               "sym" : "some_sym",
               "tag" : "IFNDEF",
               "xkids" : [
                  {
                     "tag" : "TEXT",
                     "text" : "  # ifndef text\n"
                  }
               ]
            }
         ]
      },
      {
         "sym" : "_345",
         "tag" : "ELIF",
         "xkids" : [
            {
               "tag" : "TEXT",
               "text" : "     elif text 2\n"
            }
         ]
      },
      {
         "tag" : "ELSE",
         "xkids" : [
            {
               "tag" : "TEXT",
               "text" : "    # else text\n"
```

```
          }
        ]
      }
    ]
  },
  {
    "tag" : "TEXT",
    "text" : "  Some more text\n"
  }
]
```

Note that the above is a pretty-printed version of the output; the actual output should not contain any non-significant whitespace other than the terminating newline. Note also that the order of the fields in a JSON object is undefined; hence output where the fields are in a different order is also acceptable.

Further examples are available in the extras/tests directory.

1.3 Provided Files

The prj1-sol directory contains starter shell scripts for the three scripts your submission is required to contain as well as a template README.

The extras directory contains a `tests` directory and the following testing scripts:

do-test.sh This must be invoked from the directory containing your `scan.sh` and `parse.sh` scripts with two arguments the first of which must be either `scan` or `parse` and the second must specify the name of a `*.defs` text file containing ifdef content.

The expected result for the test must be in a file having name formed from the input file by replacing the `.defs` with either `.scan.json` or `.parse¬.json`.

The script will run the appropriate `scan.sh` or `parse.sh` script on the input file and compare the standard output with the expected output. It will print out errors if the comparison fails or if the `scan.sh` or `parse.sh` script returns a non-zero exit status. It will succeed silently if the test succeeeds.

run-all-tests.sh If run without any arguments, it will run `do-test.sh` on all the tests in the tests directory, printing a message for each test.

If it is provided any argument at all, it will only run the scan tests. This is convenient for testing the lexer thoroughly before working on the parser.

1.4 Git

- Always ensure that your local copy of the `cs471` course repository is up-to-date (this manual step is particularly necessary if you are connecting to an existing x2go session):

  ```
  $ cd ~/cs471
  $ git pull
  ```

- You will likely be submitting multiple labs while working on this project. To avoid having updates to the labs and project stepping over each other, it is imperative that you create a separate branch for this project and for each lab.

  Create a branch for this project in your working copy of your github respository:

  ```
  $ cd ~/i471?               #go to clone of github repo
  $ git checkout main        #ensure in main branch
  $ git pull                 #ensure main up-to-date
  $ git checkout -b prj1     #create a new branch for this project
  ```

  Whenever you restart work on this project, it is **imperative** to ensure that you are on the correct branch. You can use commands like the following to ensure that you are in your `prj1` branch:

  ```
  $ cd ~/i471?
  $ git branch -l              #list all branches;
                               #current branch marked by a *.
  $ git checkout prj1          #checkout project branch
  $ cd submit/prj1-sol         #go to project dir
  ```

1.5 Standard Input, Standard Ouput, Standard Error

When a program starts up under any current OS, three I/O streams are always initially open:

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.

1.5.1 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(...)` will write `...` to standard error.

1.5.2 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).

1.5.3 Ruby

- The "constants" `STDIN`, `STDOUT` and `STDERR` refer to the three streams.
- STDIN.read will read from standard input until EOF.
- `print(...)` or `STDOUT.print(...)` will write `...` to standard output. `STDERR.print(...)` will write `...` to standard error.

1.5.4 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.

1.5.5 C++

`cin`, `cout` and `cerr` from `iostream` can be used for the three streams.

1.5.6 Basic C

- `<stdio.h>` defines `stdin`, `stdout` and `stderr` for the three streams.
- `getchar()`, `scanf()` will read from standard input.
- `putchar()`, `printf()` will write to standard output.
- `fwrite(..., stderr)`, `fprintf(stderr, ...)` will write to standard error.

1.5.7 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:

```
$ ./parse.sh
#ifdef sym
some text
#else
else text
#endif
some ending text
^D
```

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:

```
$ ./parse.sh < devel1.defs > devel1.parse.json
```

The `< devel1.def` redirects the contents of `devel1.defs` to the standard input of `parse.sh`; the `> devel1.parse.json` redirects the standard output of `parse.sh` to `devel1.parse.json`.

Note that `parse.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 `parse.sh` is calling a python parser, then the python parser can continue using `sys.stdin` and `sys.stdout`.

1.6 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:
  - Does not require any explicit memory management. This would rule out lower-level languages like C, C++, Rust.
  - Support regex's either in the language or via standard libraries.
  - Easy support for JSON, ideally via standard libraries.

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

- It is probably a good idea to have both `scan.sh` and `parse.sh` call a single program with that program producing a scan or parse result depending on how it was invoked.

- 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 `json_pp` which is available on `remote.cs`. Unfortunately, it seems to output the keys of an object in sorted order by name. This is irritating but not a show-stopper, especially since the choice of the name `xkids` forces the children of an AST node to be printed after the other properties.

- 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/cs471/projects/prj1/extras
  $ $extras/do-test.sh scan $extras/tests/scan0.defs
  ```

- The exit status of the last shell command is available in the shell variable `$?`. You can examine it using the command `echo $?`.

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

You may proceed as follows:

1. Review the material covered in class on regex's, scanners, grammars and recursive-descent parsing. 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.

2. Read the project requirements thoroughly.

3. Start your project in a manner similar to how you start a lab. Specifically, copy over the provided files and commit them to github:

   ```
   $ cd ~/i471?/submit
   ```

```
$ cp -pr ~/cs471/projects/prj1/prj1-sol .
$ cd prj1-sol
$ git add .
$ git commit -m 'started prj1'
$ git push -u origin prj1 #push prj1 branch to github
```

4. Fill in your details in the `README` template. Commit and push your changes.

5. Start work on your lexer. It is easiest to read the entire contents of standard input into a string and produce all the tokens in a list rather than one-at-a-time.

   Since the language is line oriented, split the input string into lines. Keep some kind of `text` variable used for accumulating consecutive text lines.

   Look at each line to classify it as either a text line or a directive line.

   - If it is a text line, simply append the lexeme to the `text` variable.
   - If it is a directive line:

     (a) If the `text` variable is non-empty add a suitable `TEXT` token pair to the output token list and reset the `text` variable to empty.

     (b) Emit the token for the starting directive to the output token list.

     (c) If the starting directive was a `#ifdef`, `#ifndef` or `#elif`, look for a `SYM`. If found, add it to the output token list.

     (d) Discard the rest of the directive line.

   Be sure to emit any leftover text to the token list when you finish with all the lines.

   Set up the `scan.sh` script to drive your lexer program. Set up the `make.sh` script too if your implementation language requires a separated compilation step.

   Test your lexer using the provided scanner tests:

   ```
   $ $extras/do-test.sh scan $extras/tests/scan.defs
   ```

   Once you feel that you have the lexer working, run all the scanner tests:

   ```
   $ $extras/run-all-tests.sh scanner-tests-only
   ```

6. Implement your parser. Initialize it with a list of all the tokens for standard input. It may be a good idea to add an artificial `EOF` token to the list

produced by the scanner, avoiding the need for the parser to check for an empty list.

(a) Set it up to maintain a `lookahead` token as some sort of "global" or instance variable. Make sure that when your parser is initialized, it primes the `lookahead` with the first token read from the lexer.

(b) Write a `peek(kind)` function which returns true iff the current lookahead has the same `kind`. Note that since this parser is quite straightforward, it is possible to avoid writing a `peek()` and merely code the necessary tests into the rest of the parser code.

(c) Write a `match(kind)` function which advances the `lookahead` to the next token from the lexer if the `lookahead` token has the same `kind`. It should output an error message to standard error and terminate the program with a non-zero exit code if the `kind`'s differ.

(d) Write the parser following the recipe provided in class for EBNF grammars.

- Since the grammar has only two non-terminals, you will need only two parsing functions `source()` and `ifdef()`. You should set up each parsing function to return the AST for the tokens which it recognizes.
- The code for `source()` should initialize an empty result list. Use a **while** loop for its body since its RHS consists of zero-or-more repetitions. Within the loop, check the lookahead:

  i. If it is a `TEXT` token, then add the corresponding AST to the result list and continue with the loop.

  ii. If it is a `IFDEF` or `IFNDEF` token, call the `ifdef()` parsing function and add its result to the result list and continue with the loop.

  iii. If it is not one of the previous two cases, terminate the loop and return the result list.
- The code for `ifdef()` should match the starting `IFDEF` or `IFNDEF` token and then match a `SYM` token. It should then call `source()` to obtain the list of ASTs for the nested source. This returned list will be the kids of the AST being constructed. Then **while** the lookahead is `ELIF`, construct AST's for `ELIF` as per the grammar, appending them to the kids list. If the lookahead after the `ELIF` loop is an `ELSE`, add an AST for the `ELSE` to the kids list. At that point, the lookahead must match a `ENDIF` token. Finally, construct and return the overall AST for the `IFDEF`/`IFNDEF` AST by build an object containing the tag, sym and kids.

Test your parser using the provided scanner tests:

```
    $ $extras/do-test.sh parse $extras/tests/parse0.defs
```

Once you feel that you have the parser working, run all the scanner and parser tests:

```
    $ $extras/run-all-tests.sh
```

7. Iterate until you meet all requirements.

It is always a good idea to keep committing your project to github periodically to ensure that you do not accidentally lose work.

1.7 Submission

Before submitting your project, update your README to specify the status of your project. Document any known issues.

Submit using a procedure similar to that used in your labs:

```
$ cd ~/i471?
$ git branch -l          #list all branches;
                          # current branch has *, should be prj1.
$ git checkout main       #goto main branch
$ git pull                # pull changes (if any)
$ git checkout prj1       #back to prj1 branch
$ git merge -m 'merged main' main   # may not do anything
$ git status -s          #should show any non-committed changes
$ git commit -a -m 'completing prj1'
$ git push                #push prj1 branch to github
$ git checkout main       #switch to main branch
$ git merge prj1 -m 'merged prj1'  #merge in prj1 branch
$ git push                #submit project
```

1.8 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 `check()` and `match()`. The semantics of the routines are slightly different: they get the next token in `accept()`, whereas we get the next token in `match()`.

- *Grammars and Parsing*, discusses building ASTs. The `peek()` and `consume()` routines are exactly equivalent to our `check()` and `match()`.