1 Project 1

Due: Mar 8, 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, followed by a discussion of its background. 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 get you to transform a grammar into one which is better suited to implementing a parser.
- To encourage you to use regular expressions to implement a trivial scanner.
- To make you implement a recursive-descent parser for a tiny language.
- To use JSON to represent abstract syntax trees.

1.2 A Tiny Language

A **Tiny Language** TL is specified by the following EBNF grammar

The grammar uses the following notation:

- Terminal symbols are shown in all uppercase or within single quotes. All other symbols are non-terminals.
- The basic grammar notation uses `:` to separate the LHS non-terminal from the first rule, `|` to separate additional rules and `;` to terminate the rules for a LHS non-terminal.
- EBNF extensions include an unquoted `?` to indicate that the previous construct is optional and an unquoted `*` to indicate zero-or-more repetitions of the previous construct. Unquoted parentheses are used for grouping,
- Grammar comments are delimited by the `#` character and end-of-line.

```
program                     # [ (def|expr)* ]
  : def program
  | expr program
  | #empty
```

```
  ;

def
  : 'def' id '(' formals ')' expr   # ast: DEF(id, formals, expr)
  ;

formals
  : id  ( ',' id )*   # ast: FORMALS(id*)
  | #empty            # ast: FORMALS()
  ;

expr
  : expr '?' expr ':' expr   # ast: ?:(expr, expr, expr)
  | expr '<' expr            # ast: <(expr, expr)
  | expr '<=' expr           # ast: <=(expr, expr)
  | expr '>' expr            # ast: >(expr, expr)
  | expr '>=' expr           # ast: >=(expr, expr)
  | expr '==' expr           # ast: ==(expr, expr)
  | expr '!=' expr           # ast: !=(expr, expr)
  | expr '+' expr            # ast: +(expr, expr)
  | expr '-' expr            # ast: -(expr, expr)
  | expr '*' expr            # ast: *(expr, expr)
  | expr '/' expr            # ast: /(expr, expr)
  | '-' expr                 # ast: -(expr)
  | integer                  # ast: integer
  | id                       # ast: id
  | '(' expr ')'             # ast: expr
  | id '(' actuals ')'       # ast: APP(id, actuals)
  ;

actuals
  : expr ( ',' expr )*       # ast: ACTUALS(expr*)
  | #empty                   # ast: ACTUALS()
  ;
integer
  : INT                      # ast: INT() @value: integer value
  ;

id
  : ID                       # ast: ID() @id: lexeme for ID
  ;
```

1.2.1 Lexical Constraints

TL programs need to meet the following lexical constraints:

- Comments start with a `#` character and extend till end-of-line. Comments and whitespace are not significant to TL programs.
- The terminal `INT` must be a sequence of one-or-more digits.
- The terminal `ID` must be a sequence of alphanumerics or `_` but cannot start with a digit.
- All tokens output by the scanner must have `kind` and `lexeme` fields.
  - The `lexeme` field must always contain the substring of the source program corresponding to the token.
  - All tokens except those with `kind ID` and `INT` must have their `kind` field set to the all-uppercase version of the `lexeme` field.

1.2.2 Syntactic Constraints

- The precedence of the operators is listed below from lowest to highest with operators on the same line having the same precedence:

  Conditional operator: `?:`
  Relational operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
  Additive operators: `+`, `-` (binary)
  Multiplicative operators: `*`, `/`
  Unary minus: `-`

  The conditional operator `?:` is right-associative, the relational operators are non-associative, all other operators are left-associative.

- An AST must have at least 2 fields: `tag` and `kids`; the value of the former must be a string and the value of the latter a list (possibly empty) of AST's.
- In tbe above grammar, an AST with tag `T` and kids `K1`, `K2`, ..., `Kn` is shown as `T(K1, K2...,Kn)`.

  Extra fields F added to an AST node are shown as @F. This applies to AST's corresponding to primitive tokens.

  - The AST for the integer 7 should have the following fields:

    `tag`: `INT`
    `kids`: `[]`
    `value`: `7`; note this should be of type integer.

  - The AST for the id xyz should be

    `tag`: `ID`
    `kids`: `[]`
    `id`: `"xyz"`

- The AST for an overall program is a list of `expr` and DEF AST's in the same order as the corresponding source code.
- The AST with tag FORMALS must have a `ID` kid for each formal parameter.
- The AST with tag ACTUALS must have a `expr` kid for each argument.
- The order of an AST's kids must be the same as that of the corresponding symbols in the rule.

1.3 Requirements

Use your favorite programming language to implement a recognizer for TL. 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 a TL program from standard input and output on standard output a JSON list of the tokens corresponding to the TL program followed by a newline. Each token is as specified above. Note that characters which are not legal TL characters should still be recognized as single-character tokens.
3. Typing `./parse.sh` within that directory will read a TL program from standard input and output on standard output a JSON list of the AST's corresponding to the TL program followed by a newline.

   If there are errors in the TL program, the program should terminate after detecting the first syntax error. It should output a reasonably detailed error message on standard error.

The JSON output should not contain any non-significant whitespace except for the terminating newline.

1.4 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 example TL programs, and some sample JSON outputs. Note that the JSON outputs are quite unreadable because of the lack of whitespace but that is easily remedied:

```
$ cd ~/cs471/projects/prj1/extras
```

```
$ cat simple-asts.json | json_pp
```

It also contains a backend for translating your JSON AST's into working C code which can then be run:

```
$ cd ~/cs471/projects/prj1/extras/backends
$ mkdir -p ~/tmp
$ ./c-target.mjs < ../example-asts.json > ~/tmp/example.c
$ cd ~/tmp
$ gcc -g -Wall example.c -o example
$ ./example
1
24
720
1
24
720
1
2
21
144
1
2
21
144
```

1.5 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 in 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
```

1.6 Hints

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

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. Choose the implementation language for your project. Ideally it 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.
4. Start your project in a manner similar to how you start a lab.
   (a) 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
```

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

6. The requirements forbid extraneous whitespace in the JSON output which makes the output quite hard to read. To get around this, 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, which means that `kids` print before `tag`. This is irritating but not a show-stopper.

7. Start work on your lexer. The tokens it needs to recognize are particularly simple:

   - The reserved word `def` which is recognized as a token with `kind` set to `DEF`.
   - Multiple character operators like `<=`, `>=`, `==`, `!=`.
   - Integer literals with `kind` set to `INT`.
   - Identifiers with `kind` set to `ID`.
   - All other single character operators.

   Additionally, the lexer needs to be set up to ignore whitespace and `#` to end-of-line comments.

   [These requirements are simple enough that it is quite simple to implement the lexer without using regex's, but it is even simpler using regex's.]

   Decide whether your lexer will deliver tokens one at a time or all at once in a list (the latter is a simpler choice).

   Your lexer should read the **entire** standard input into a string. (this makes the subsequent recognition of tokens easier). The lexer should be set up as a loop which loops while you do not have a token.

   (a) At the start of the loop check whether the unprocessed portion of the input starts with whitespace. If so, continue with the loop with the whitespace discarded from the input.

   (b) Check whether the unprocessed portion of the input starts with a `#` comment. If so, continue with the loop with the comment discarded from the input.

   (c) Check whether the unprocessed portion of the input starts with any of the multiple character tokens like `ID`, `INT` or any of the multi-

ple character operators. If so, produce the corresponding token and discard the matching lexeme from the input.

(d) If none of the above cases apply, produce a single character token with both `kind` and `lexeme` set to the character. Discard the character from the input.

It is key that you set up your regex's to match only at the **start** of the unprocessed portion of the input. In most regex engines this can be achieved by using the `^` anchor.

8. Transform the provided grammar into an EBNF grammar suitable for recursive-descent parsing. These transformations will be needed for the operator part of the grammar.

   - Use separate non-terminals for each precedence level (repeated here, in order of increasing precedence):

     Conditional operator: `?:` (right-assoc)
     Relational operators: `<`, `<=`, `>`, `>=`, `==`, `!=` (non-assoc)
     Additive operators: `+`, `-` (binary) (left-assoc)
     Multiplicative operators: `*`, `/` (left-assoc)
     Unary minus: `-`

   - The non-terminal for a precedence level will be defined using the non-terminal for the next higher precedence level.
   - If the operators for a level are right-associative, then use right-recursive rules.
   - If the operators for a level are left-associative, then use the Kleene-closure operator `( ... )*`.
   - If the operators for a level are non-associative, then the rules for that level will not be recursive.
   - The highest precedence level should contain a right-recursive rule for unary minus plus rules for parenthesized expressions, function calls and primitive expressions like identifiers and integers.

9. Set up your parser 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.

10. Write a `check()` function which returns true iff the `kind` of the `lookahead` token matches the `kind` provided as its argument.

11. Write a `match()` function which sets `lookahead` to the next token from the lexer if the `lookahead` token matches the `kind` provided as its argument. If that is not the case, it should set things up to output a detailed error message to standard error and terminate the program.

12. Write some kind of `Ast class` which supports standard `tag` and `kids` fields as well as an optional `value` field when `tag` is `INT`'s and an optional `id` field when `tag` is `ID`.

13. Develop your parser in the following order so that you can test at each stage:

    (a) Primitive integer and identifier expressions.

    (b) Unary minus expressions.

    (c) Multiplicative expressions.

    (d) Additive expressions.

    (e) Relational expressions.

    (f) Conditional expressions.

    (g) Parenthesized expressions.

    (h) Function call expressions. This will require you to define a parsing function for the `actuals` non-terminal in the grammar specification.

    (i) Function definitions.

    Adjust your top-level parsing function at each stage to call the current function which is under development. Output the results of your top-level parsing function as JSON (pipe the result through `json_pp` to pretty-print the result).

    It is important to keep in mind that calling `match()` destroys the current value of `lookahead`. Hence if you need part of the `lookahead` for subsequent building of the AST, then it is **imperative** that you squirrel away that information into a local variable before calling `match()`.

    The provided devel.tl is set up with test cases which you can uncomment in successive stages of development.

14. 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

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

```
$ cd ~/i471X
$ 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()`.