CS316: Compilers Lab

Programming Assignment 1 - Git Setup, Submission Instructions, and Working of a Lexical Analyzer
Due: 21/1/2022

The objective of this assignment is to gain a hands-on experience with:

1. A version control system such as Git,
2. `Make` and `makefiles`
3. Using a lexical analyzer generator such as `Flex`.

1 Version Control Systems and `Git`

Have you ever spent hours writing code (or a term paper) only to realize, when your computer crashes, that you haven't saved your progress? Have you ever spent an hour making a change to a program only to realize that all your changes were wrong and you had to go back to the way the code was? Version control helps solve these problems.

Every non-trivial program is written in stages. To write a good program, you need to divide it into stages and finish one stage at a time. After you finish one stage, commit a version so that you have a record. Version control provides many advantages. One of them is a simple way to back up your code. If, for any reason, you want to go back to a previous version, it is very easy. Version control does many more things than backing up your code.

Please understand that you must commit changes often if you want to use version control. If you do not commit, version control cannot help you.

This class uses `git` for version control. Git is a distributed version control system. That means there are two repositories: local and remote. When you commit changes, only the local repository is changed. This makes commits fast and independent of network connections. If your computer is damaged, you still lose the local repository.

To make changes to the remote repository, you need to push the changes done on the local repository. If your computer is damaged, you can retrieve the code from the remote repository.

Please read the guide at github about how to use version control.

Please remember that you must commit and push often to take advantage of version control.

Version control is required in this class. So, the instructors will not accept any excuse like "I accidentally deleted my code and please give me an extension." Neither will "my computer crashed" be accepted as an excuse.

You must commit and push often to demonstrate your progress of the assignments. If there is any doubt about academic dishonesty, your commit history would be an important piece of evidence proving your innocence.

Please read the book about how to use version control: https://git-scm.com/book/en/v2

1.1 Prerequisites

Create a Github account (if you do not already have one). This is the account you should use to create and submit all of your assignments this semester.

Fill the Google form shared with you earlier and inform the TA and Instructors with your GitHub username.

1.2 `Git - setup`

The `Git` setup and submission instructions remain the same for all assignments. Please replace the assignment number in the examples with the number of the assignment you are submitting.

1. Create a Git repository for the assignment.

   (a) Log in to your Github account.

   (b) Visit the Github Teams Discussion page to find the link for the assignment PA1. Click the link. This will create a repository on Github for the assignment (you will follow a similar procedure for all future assignments). Make sure that the repository is called 'IITDhCSE/PA1-<your GitHub username here>'.

   (c) Clone the repository to develop your assignment. Cloning a repository creates a local copy. Change your directory to whichever directory you want to create your local copy in, and type:

   ```
   > git clone git@github.com:IITDhCSE/PA1-<your GitHub username here>.git PA1
   ```

   This will create a subdirectory called `PA1`, where you will work on your code.
   In this command: `git clone` copies a repository.
   `git@github.com:IITDhCSE/PA1-<your GitHub username here>.git` tells `git` where the server (remote copy) of your code is.
   `PA1` tells `git` to place the code in a local directory named `PA1`
   If you change to directory `PA1` and list the contents, you should see the files you will need for this assignment:

   ```
   > cd PA1
   ```

   ```
   > ls
   ```

   And you should see all of the files required to get started with this assignment.

   Sometimes, you may see an error accessing the repository and the clone command may fail to recognize the existance of a repository. One of the reasons could be setting up access credentials:

   Setting up SSH key with GitHub Set up a public SSH key in your GitHub account (if you haven't already). To do this, first generate a new ssh key:

   ```
   > ssh-keygen
   ```

   Hit enter three times (to accept the default location, then to set and confirm an empty passphrase). This will create two files: /.ssh/id_rsa (your private key) and /.ssh/id_rsa.pub (your public key) Then print out your public key:

   ```
   > cat  /.ssh/id_rsa.pub
   ```

   And copy it to the clipboard. Then follow steps at: https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account

2. As you develop your code, you can commit a local version of your changes (just to make sure that you can back up if you break something) by typing:

   ```
   > git add <file name that you want to commit>
   ```

   ```
   > git commit -m "< describe your changes>"
   ```

   `git add <filename>` tells `git` to "stage" a file for committing. Staging files is useful if you want to make changes to several files at once and treat them as one logical change to your code. You need to call `git` add every time you want to commit a file that you have changed.

   `git commit` tells `git` to save a new version of your code including all the changes you staged with `git add`. Note that until you execute `git commit`, none of your changes will have a version associated with them. You can save/commit the changes many times. It is a good habit committing often. It is very reasonable if you commit every ten minutes (or more often).

Do not type `git add *` because you will likely add unnecessary files to the repository. When your repository has many unnecessary files, committing becomes slower. If the unnecessary files are large (such as executables or core files), committing can take several minutes and your assignments may be considered late.

3. The changes you saved by executing `git commit` in the previous step are local to your development environment i.e. they are saved in a local repository. To copy your changes back to Github (to make sure they are saved if your computer crashes, or if you want to continue developing your code from another machine, or you want to make your code visible to a collaborator), type

   ```
   > git push
   ```

   If you do not push, the teaching staff cannot see your solutions.

2 Write a Makefile

Makefiles let you define complicated sets of commands to build your projects. Makefiles consist of a series of rules:

```
[target] :  [dependences]
[TAB]   command 1
[TAB]   command 2
...
```

A rule *target* is the name of the rule. The *dependences* are the files the rule depends on. The *commands* are what to do when the rule is "fired". Note: there must be a tab before each command.

A rule is fired in one of two ways: (i) it is directly invoked (by calling "make [target]") or (ii) it is invoked by another rule that is fired.

When a rule is fired, it goes through the following process:

1. If a *dependence* has a rule in the Makefile, fire that rule (using this same process)
2. Once all dependences have been fired, check to see if *target* is "out of date": interpret *target* as a filename, and see if the timestamp on the file is older than the time stamp on any of its dependences. If it is, the target is "out of date". If there is no file named *target*, the target is always assumed to be out of date. If there are no dependences and *target* exists, target is assumed to be up to date.
3. If the target is out of date, execute the list of commands

You can use Makefiles to orchestrate complicated build processes.
If you type "make" without a target, make will fire the first rule in the Makefile.
We usually define a target called "clean" whose job it is to clean up any intermediate files generated during the build process. This can also be used to remove all generated targets to force recompiling everything.

Makefiles also let you define macros to reuse the same commands over and over. For example, we can define GCC as a macro that invokes gcc the way we want:

```
DEBUG = -DDEBUG
CFLAGS = CFLAGS = -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror
GCC = gcc $(CFLAGS) $(DEBUG)
```

Note that we use $(MACRO_NAME) to insert the macro into other places, including commands. Makefiles can get much more complicated than this, but their full power is beyond the scope of this course.

Edit the `Makefile` given to you in the main directory of your repository (we will be using the `Makefile` to drive the building and testing process of all of your code.

Create a target called `dev` that first prints out your name:

1. Full name (first and last)
2. IITDh email ID

For example, if the instructor typed:

`make dev`

it would print the following:

```
Nikhil Hegde
nikhilh@iitdh.ac.in
```

3 Scanner

The working of a scanner—sometimes called the tokenizer or lexical analyzer—forms the first phase of the compilation process. A scanner is a program that takes a sequence of characters (the source file of the program) and produces a sequence of tokens that will be used to feed the compiler's parser, the next phase of the compilation process. So, for example, the input

`A := B + 4`

Would translate into the following tokens:

`IDENTIFIER (Value = "A")`

`OPERATOR (Value = ":=")`

`IDENTIFIER (Value = "B")`

`OPERATOR (Value = "+")`

`INTLITERAL (Value = "4")`

The way that we define tokens in a programming language is with *regular expressions*. For example, a regular expression that defines an integer literal token looks like:

`[0-9]+` (read: "1 or more digits"),

while a regular expression that defines a float literal token looks like:

`[0-9]+[̇0-9]* | [̇0-9]+` (read: "Either 1 or more digits followed by a decimal followed by 0 or more digits; or a decimal followed by 1 or more digits")

While you can write a scanner by hand, it is very tedious. Instead, we typically use tools to help us automatically generate scanners. The tools we recommend you to use are either flex (if you're planning on writing your compiler in C or C++) or ANTLR (if you're planning on writing your compiler in Java). flex is available on most Unixes/Linux (including the ecegrid machines), while ANTLR requires a download. If you want to use other tools to generate your scanner, feel free, but we will be able to provide less help.

3.1 Token definitions

We will be building a compiler for a simple language called MICRO in this class. The token definitions (written in plain English) are as follows:

```
an IDENTIFIER token will begin with a letter, and be followed by any number of
    letters and numbers.
IDENTIFIERS are case sensitive.

INTLITERAL: integer number
            ex) 0, 123, 678

FLOATLITERAL: floating point number available in two different format
```

```
                yyyy.xxxxxx or .xxxxxxx
            ex) 3.141592 , .1414 , .0001 , 456.98

STRINGLITERAL: any sequence of characters except '"'
            between '"' and '"'
            ex) "Hello world!" , "************" , "this is a string"

COMMENT:
      Starts with "--" and lasts till the end of line
      ex) -- this is a comment
      ex) -- any thing after the "--" is ignored

Keywords:

PROGRAM,BEGIN,END,FUNCTION,READ,WRITE,
IF,ELSE,FI,FOR,ROF, BREAK, CONTINUE,
RETURN,INT,VOID,STRING,FLOAT

Operators

:= + - * / = != < > ( ) ; , <= >=
```

3.2 What you need to do

You should build a scanner that will take an input file and output a list of all the tokens in the program. For each token, you should output the token type (e.g., OPERATOR) and its value (e.g., +).

You are provided a set of sample inputs and outputs as part of the starter files of the repository. These are the only inputs we will test your compiler on. Your outputs need to match our outputs exactly (we will be comparing them using diff, though we will ignore whitespace).

3.3 Hints

Note that even though our sample outputs combine together a bunch of different tokens as a single type (e.g., all keywords have the token type KEYWORD), you will be better served by defining every keyword and operator as a *different* token type (so your scanner will have different tokens for, say, := and <), and then writing a little bit of extra code to print the output we expect for that token type.

While it might seem weird, you *will* need to define a token that eats up any whitespace in your program (recall that your compiler really only sees a list of characters; it has no reason to think that a tab character isn't an important character). Make sure that when you recognize a whitespace token, you just silently drop it, rather than printing it out.

3.4 What you need to submit

- All of the necessary code for your compiler/scanner that you wrote *yourself.*
- A Makefile with the following targets:
  1. compiler: this target will build your compiler
  2. clean: this target will remove any intermediate files that were created to build the compiler
  3. dev: this target will print the same developer information that was explained in Section 2.
- A shell script (this must be written in bash) called `runme` that runs your compiler/scanner (note that there are no extensions like `.sh`). This script should take in two arguments: first, the input program file to be compiled and second, the filename where you want to put the compiler's/scanner's output. E.g. `./runme nested.micro myoutput.txt` should execute your scanner with the input `nested.micro`

and produce output that gets written in `myoutput.txt`. You can assume that we will have run `make compiler` before running this script.

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, your Makefile and `runme` should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation.

4 Submitting your code

You will use `git`'s "tagging" functionality to submit assignments. Rather than using any submission system, you will use `git` to *tag* which version of the code you want to grade. To tag the latest version of the code, type:

```
> git tag -a <tagname> -m ‘‘<describe the tag>’’
```

This will attach a tag with name <tagname> to your latest commit. Once you have a version of your program that you want to submit, run the following commands:

```
> git tag -a cs316pa1submission -m "Submission for PA1"
> git push --tags
```

We would be using a tag name cs316pa1submission. So, the above commands will create a tag named "cs316pa1submission" and push it to the remote server. The grading system will check out whichever version you have tagged "cs316pa1submission" and grade that. If you want to update your submission (and tell the grading system to ignore any previous submissions) type:

```
> git tag -a -f cs316pa1submission -m "Submission for PA0"
> git push -f --tags
```

`> git tag -a -f cs316pa1submission -m "Submitting PA1"` overwrites the tag on the local repository.

`git push -f -tags`, pushes the updates and overwrites the tag on the remote repository (on Github).

These commands will overwrite any other tag named `cs316pa1submission` with one for the current commit. Please be careful about the following rules:

1. For each assignment, you should tag only one version with `cs316pa1submission`. It is your responsibility to tag the correct one. You CANNOT request regrading if the grading program retrieves the version that you do not want to submit.

2. After tagging a version `cs316pa1submission`, any modifications you make to your program WILL NOT BE GRADED (unless you update the tag, as described above).

3. The grading program starts retrieving soon after the submission deadline of each assignment. If your repository has no version tagged `cs316pa1submission`, it is considered that you are late.

4. The grading program checks every student's repository 120 hours after the submission deadline. If a version tagged `cs316pa1submission` is found, the grading program retrieves and grades that version.

5. The grading program uses only the version tagged `cs316pa1submission`. It does NOT choose the higher score before and after the submission deadline. If a later version has the `cs316pa1submission` tag, this later version will be graded with the late discount. Thus, you should tag a late version with `cs316pa1submission` only if you are confident that the new score, with the late discount, is higher.

6. The time of submission is the time when you push the code to the repository, not the time when the grading program retrieves your code. If you push the code after the deadline, it is late. Even though you push before the grading program starts retrieving your program, it is still considered late.

7. You should push at least fifteen minutes before the deadline. Give yourself some time to accommodate unexpected situations (such as slow networks).

8. You are encouraged to tag partially working programs for submission early. In case anything occurs (for example, your computer is broken), you may receive some points. Please remember to tag improved version as you make progress.

Do not send your code for grading. The only acceptable way for grading is to tag your repository.

Under absolutely no circumstance will the teaching staff (instructors and teaching assistants) debug your programs without your presence. Such email is ALWAYS ignored. If you need help, go to office hours, or post on the discussion forum.