CS 211
Fundamentals of Computer Programming II
Winter 2024

Project #01 (v1.4)

Assignment: Scanner for nuPython
Submission: Gradescope
Policy: individual work only, late work *is* accepted
Complete By: Friday January 12th @ 11:59pm CST

Late submissions: see syllabus for late policy… No submissions accepted after Sunday 1/14 @ 11:59pm

Pre-requisites: HW 01

Overview

In CS 211, projects 01 – 04 focus on C programming. We're going to build an execution environment for a subset of the Python language called **nuPython**. Our subset will group statements using { } instead of indentation, and add pointer-based operators * and &.

The first step in building an execution environment is developing a **scanner** that reads characters from a Python program and forms **tokens** --- base elements of the nuPython language. For example, given the input

```
# this is a comment
x = input('Enter a value>')
print(x)
```

The output from the scanner are the following tokens:

```
Token 25 ('x') @ (2, 1)
Token 13 ('=') @ (2, 3)
Token 25 ('input') @ (2, 5)
Token 1 ('(') @ (2, 10)
Token 24 ('Enter a value>') @ (2, 11)
Token 2 (')') @ (2, 27)
Token 25 ('print') @ (3, 1)
Token 1 ('(') @ (3, 6)
Token 25 ('x') @ (3, 7)
Token 2 (')') @ (3, 8)
Token 0 ('$') @ (4, 1)
```

The comment is discarded, and the first token returned is the identifier 'x'; identifiers are names such as

variable and function names. Identifiers are defined as token #25; the "(2, 1)" denotes the line and column where the token 'x' started. The next token is the '=' sign (token #13), followed by an identifier token 'input' denoting the Python input( ) function. The last token is always token #0, which denotes the "end of stream" (EOS) and symbolized by the character '$'.

Your assignment here in project 01 is to build a scanner for nuPython. You will be provided with starter code, and a partially implemented scanner. Here's an example of running the starter code with the following keyboard input:

```
print(x)
x**y
x*y
$
```

```
nuPython input (enter $ when you're done)>
print(x)
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('x') @ (1, 7)
Token 2 (')') @ (1, 8)
x**y
Token 25 ('x') @ (2, 1)
Token 10 ('**') @ (2, 2)
Token 25 ('y') @ (2, 4)
x*y
Token 25 ('x') @ (3, 1)
Token 9 ('*') @ (3, 2)
Token 25 ('y') @ (3, 3)
$
Token 0 ('$') @ (4, 1)
```

For testing purposes, input may come from a file or the keyboard. The example above shows keyboard input which allows for easier testing of your scanner.

Tokens

A **token** is defined as a triple (id, line, col), where **id** is a unique id number (e.g. "(" is defined as 1), **line** is the line number where the token was found, and **col** is the column where the token started. For example, suppose the input is the following input stream spanning 2 lines:

```
print(x)
count = y
```

Your scanner should yield the following token sequence: (25, 1, 1), (1, 1, 6), (25, 1, 7), (2, 1, 8), (25, 2, 1), (13, 2, 7), and (25, 2, 9). Line and column numbers start at 1, identifiers have id = 25, etc. In C, we use a struct ("structure") to define a Token (a structure is like a class in Java or Python). The Token struct is defined in the provided header file "token.h":

```
struct Token
{
  int id;    // token id (see enum below)
  int line;  // line containing the token (1-based)
  int col;   // column where the token starts (1-based)
};
```

Here is the complete list of tokens that your scanner must recognize. These are defined in “token.h” (enum means “enumerated list of constants”, in this case starting at the value -1 and counting upward by 1):

```
enum TokenID
{
  nuPy_UNKNOWN = -1,  // a character that is not part of nuPython
  nuPy_EOS,           // end-of-stream, denoted by EOF or $
  nuPy_LEFT_PAREN,    // (
  nuPy_RIGHT_PAREN,   // )
  nuPy_LEFT_BRACKET,  // [
  nuPy_RIGHT_BRACKET, // ]
  nuPy_LEFT_BRACE,    // {
  nuPy_RIGHT_BRACE,   // }
  nuPy_PLUS,          // +
  nuPy_MINUS,         // -
  nuPy_ASTERISK,      // *
  nuPy_POWER,         // **
  nuPy_PERCENT,       // %
  nuPy_SLASH,         // /
  nuPy_EQUAL,         // =
  nuPy_EQUALEQUAL,    // ==
  nuPy_NOTEQUAL,      // !=
  nuPy_LT,            // <
  nuPy_LTE,           // <=
  nuPy_GT,            // >
  nuPy_GTE,           // >=
  nuPy_AMPERSAND,     // &
  nuPy_COLON,         // :
  nuPy_INT_LITERAL,   // e.g. 123
  nuPy_REAL_LITERAL,  // e.g. 3.14 or .5 or 89.
  nuPy_STR_LITERAL,   // e.g. "hello cs211" or 'hello cs211'
  nuPy_IDENTIFIER,    // e.g. print or sum or x
  //
  // keywords:
  //
  nuPy_KEYW_AND,      // and
  nuPy_KEYW_BREAK,    // break
  nuPy_KEYW_CONTINUE, // continue
  nuPy_KEYW_DEF,      // def
  nuPy_KEYW_ELIF,     // elif
  nuPy_KEYW_ELSE,     // else
  nuPy_KEYW_FALSE,    // False
  nuPy_KEYW_FOR,      // for
  nuPy_KEYW_IF,       // if
  nuPy_KEYW_IN,       // in
  nuPy_KEYW_IS,       // is
  nuPy_KEYW_NONE,     // None
  nuPy_KEYW_NOT,      // not
  nuPy_KEYW_OR,       // or
  nuPy_KEYW_PASS,     // pass
  nuPy_KEYW_RETURN,   // return
  nuPy_KEYW_TRUE,     // True
  nuPy_KEYW_WHILE     // while
};
```

Most of the tokens are self-explanatory, e.g. "(" and "+". However, some additional clarifications are necessary… First, given a choice, a token always matches the longest possible input sequence. For example, given the input string

```
sum==123
```

This is 3 tokens: an identifier "sum", the equality operator "==", and the integer literal "123". Note the input "==" is not 2 separate tokens "=" and "=", but a single token "==".

Whitespace is used to make the input more readable to other people --- the scanner treats whitespace as the end of a token, and then skips over whitespace in search of the next token. Whitespace is defined as blanks, tabs, and end-of-line characters (\n, \r, and \f). For example:

```
sum    ==
    123
```

and

```
sum==123
```

are identical in terms of the token ids returned (the line and column numbers will differ).

An **identifier** is defined as a letter or underscore, followed by 0 or more letters / digits / underscores. A letter can be lowercase or uppercase. Examples: "X", "_count", "field_X_3". The following are keywords and not identifiers, and should be identified as such: "and", "break", "continue", "def", "elif", "else", "False", "for", "if", "in", "is", "None", "not", "or", "pass", "return", "True", "while". Keywords are case-sensitive, for example the identifier "AND" is not a keyword. Note 3 keywords start with a capital letter: "False", "True", and "None".

A **string literal** starts with a quote, contains 0 or more characters, and ends with the same type of quote. This implies a string literal can start and end with a single quote ', or start and end with a double quote ". Note that a string literal starting with a single quote cannot contain a single quote; likewise a string literal starting with a double quote cannot contain a double quote. Examples: "this is a string literal", and 'they said "hi" '. If the literal is not properly terminated --- either the closing quote is missing or it's the wrong one --- then your scanner must output a warning message of the following format:

```
nuPython input (enter $ when you're done)>
s = "This is a string without a terminator
Token 25 ('s') @ (1, 1)
Token 13 ('=') @ (1, 3)
**WARNING: string literal @ (1, 5) not terminated properly
Token 24 ('This is a string without a terminator') @ (1, 5)
$
Token 0 ('$') @ (2, 1)
```

If not properly terminated, your scanner returns the remainder of the line as the string literal. Note that the value of a string literal is the string without the leading and closing quotes.

An **integer literal** is a sequence of one or more digits. Examples: “000” and “2”. A **real literal** is a sequence of 1 or more digits, followed by a decimal point, followed by 0 or more digits **OR** a sequence of 0 or more digits, followed by a decimal point, followed by 1 or more digits. Examples: “3.14159”, “.525”, and “12345.”.

The **EOS** token denotes end-of-stream, and is returned when “$” is encountered in the input stream, or EOF (end-of-file) is reached in an input file. Finally, any other non-whitespace character in the input is returned as an **UNKNOWN** token.

The scanner needs to skip comments (much like it skips whitespace). In nuPython, comments start with "#" and continue to the end of the current line. Example:

```
# This is a comment:
x = 123   # initialize
print(x)  # print
```

In this case the first token returned by the scanner is the identifier “x” on line 2, column 1.

Getting Started --- computer setup

To help you prepare for future classes (e.g. CS 213), we use the EECS Linux computers for our projects. These computers live in the Tech building, and you need an internet connection to access them. The first step is to create an account (or reset your EECS password if you already have an account):

a. If you are working on Windows: install Git Bash, then follow these instructions to install
b. Based on the computer you have, do the following:
    i. Linux: open terminal
    ii. Mac: open terminal
    iii. Windows: open Git Bash

c. At the command prompt type: **`ssh YOUR_NETID@moore.wot.eecs.northwestern.edu`**
    i. Enter your netid in lowercase
    ii. Type your **EECS password** (not your NU password) and press ENTER --- note that nothing appears as you type (for security reasons). If you don’t have an EECS password, skip to the next step.
    iii. If you are new to EECS / don’t remember your EECS password, use this link and obtain a temporary password --- enter your netid (lowercase) and NU password to start. This link only works once; if you need to reset a second time or need help, email EECS IT
    iv. If the “moore” computer is unavailable, here’s a list of other computers to try

d. Once you have successfully logged in, enter the following command: **`hostname`**
e. Confirm the remote computer is moore.wot.eecs.northwestern.edu
f. Logout from the remote computer with this command: **exit**
g. Close the terminal / Git Bash window using the **exit** command (or just close the window)

Note that “moore” is the only computer directly accessible from off-campus; all other EECS computers must be accessed using a VPN.

If you are able to successfully login, then all is well. Here's an **optional** step you can follow that will safely shorten the remote login process if you want. Follow these steps:

1. If you are connected to a remote computer, exit that connection (or just close the window)
2. On your **LOCAL** computer… If you are working on Linux or Mac, open a terminal. If you are working on Windows, open Git Bash. Type the following command into the console window EXACTLY as shown:

```
bash -c "$(curl -fsLS https://bit.ly/3nBxDH3)"
```

3. Follow the prompts… Remember your passphrase if you provide one!
4. When this is done, you'll be able to safely login by simply typing

```
ssh moore
```

Editing and compiling on Linux computers

Login to the remote computer **moore.wot.eecs.northwestern.edu**. We are going to work at the command-line for a few minutes:

1. Make a new directory (folder): **`mkdir test`**
2. Move ("change") into that directory: **`cd test`**
3. List the contents of that directory: **`ls`**

```
hummel@moore$ mkdir test
hummel@moore$ cd test
hummel@moore$ ls
hummel@moore$
```

Let's write a simple C program (shown on the right):

4. Use the micro editor to create main.cpp: **`micro main.c`**
5. Write program to input 2 ints, add, output the sum ---------->
6. Save your work: **`ctrl-S`**
7. Exit (quit) the micro editor **`ctrl-Q`**
8. List the contents of the directory: **`ls`**

```c
#include <stdio.h>

int main()
{
  int x, y, z;

  scanf("%d", &x);
  scanf("%d", &y);

  z = x + y;

  printf("Sum is %d\n", z);

  return 0;
}
```

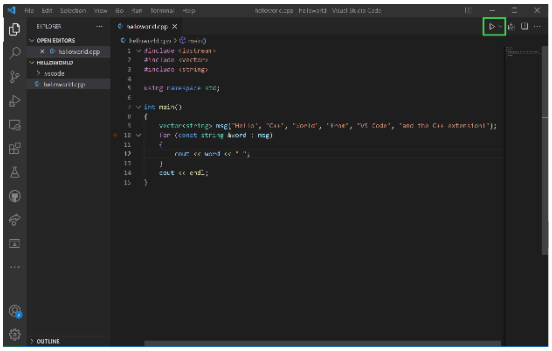
You should now see your "main.c" file listed. The final step is to compile and run your program:

9. Compile with gcc **`gcc -std=c11 -g -Wall main.c`**
10. Run **`./a.out`**
11. Change back to your "home" directory **`cd`**
12. List the contents of your home directory **`ls`**

Your home directory should now contain a "test" directory. This directory (and your work) will remain on the EECS computers after you logout, so you don't need to worry about downloading and saving your work to your local computer unless you want to make a backup copy.

Visual Studio Code

This step is completely optional, but most folks prefer working in VS Code so you can open multiple files at the same time, receive code completion tips, etc. The idea is to use VS Code as a front-end to the EECS servers. You edit locally, but your work is saved on the EECS computers, and you compile and run using the EECS computers. This implies you need an internet connection for this to work.



The 211 course staff have put together a very nice tutorial for setting up VS Code on your local computer to interact with the remote EECS computers. If you're interested, follow the instructions in this PDF. Note that when you are asked to select the type of remote computer, select **LINUX** because the EECS computers are running the Linux OS.

After installing and configuring VS Code, you'll want to install the following extensions so you can program in C and C++. Steps:

1. In VS Code, view extensions (left-most window pane)
2. Install C/C++ from Microsoft (here's a nice post with visuals, just install the extension)
3. Install C/C++ Extension Pack from Microsoft (see post above, it's also shown in the screenshot)
4. Enable auto-save for your work? File >> Preferences >> Settings.

That's it, VS Code is now setup for modern C and C++ programming.

Copying the provided files

Login to the remote computer **moore.wot.eecs.northwestern.edu**. Copy over the provided files to your own account as follows:

1. Make a directory for project 01 — `mkdir project01`
2. Make this directory private — `chmod 700 project01`
3. Move ("change") into this directory — `cd project01`
4. Copy the provided files --- the `.` is needed — `cp -r /home/cs211/w2024/project01/release .`
5. List the contents of the directory — `ls`

As this point you should see the directory "release" shown. Now

6. Move ("change") into release dir — `cd release`
7. List the contents of the directory — `ls`

You should see the following:

```
hummel@moore$ cp -r /home/cs211/w2024/project01/release .
hummel@moore$ ls
release
hummel@moore$ cd release
hummel@moore$ ls
main.c  makefile  scanner.c  scanner.h  test01.py  test02.py  test03.py  token.h
hummel@moore$
```

There are 8 provided files, here's a summary:

Header files:	scanner.h, token.h
C source files:	main.c, scanner.c
Test input files:	test01.py, test02.py, test03.py
Make file:	makefile

The "makefile" contains commands to build, run, and submit for grading. The main( ) function in "main.c" is complete, the scanner in "scanner.c" is partially implemented. To build and run what is provided:

8. Build the program: **`make build`**
9. Run the program (keyboard input): **`./a.out`**
10. Run the program (file input): **`./a.out test01.py`**

Getting started on the assignment…

Your job is to complete the scanner implementation in the file "scanner.c" --- do not modify the other .h or .c files. The .py files are input files for testing your scanner. Appendix A at the end of this document shows the correct output for the provided test files. Appendix B is a copy of the provided code in "scanner.c".

If you installed VS Code, establish a remote connection to one of the EECS computers, and then open the release folder for project 01. From there you'll be able to open the .c and .h files and navigate the provided program. When you want to build and run the program, open a new terminal window in VS Code using the "…" menu, and use the commands shown above in steps 8-10. The workflow is to edit your scanner.c file, save, switch to the terminal window, build, run, test. Repeat.

If you did not install VS Code, then use Git Bash or Terminal to connect to an EECS computer, navigate to the release directory, and use the micro editor to navigate and edit the files. When you want to build and run the program, quit micro, and use the commands shown above in steps 8-10. The workflow is to edit your scanner.c file, save, quit micro, build, run, test. Repeat

The first step is to review the provided code, e.g. look over the main() function in "main.c" to see how it interacts with the scanner. Then review the provided code in "scanner.c" to see the logic behind how the scanner operates --- reading the input stream one character at a time, deciding what to do based on the first character of the token. Note that the provided code runs without error, and recognizes the following tokens: EOF, "$", end-of-line, whitespace, "(", ")", "*", "**", identifiers, and unknown tokens. Newlines and whitespace are also handled.

In particular, you'll want to review how "*" and "**" tokens are handled, since they both start with "*". Likewise, review carefully how identifiers are handled, in particular how the **collect_identifier( )** function is defined and implemented; you will be required to create similar functions for handling integer, real, and string literals. See Appendix B for a copy of the code provided in "scanner.c".

When you are ready to start implementing the rest of the scanner, complete the following steps:

1. Recognize the remaining punctuation: +, -, /, %, =, ==, !=, <, <=, etc. Run and test.
2. Recognize keywords by checking the identifier value against the set of given keywords; the cleanest solution is to put the keywords in an array and perform a linear search. See the "TODO" in the code that recognizes identifiers. You can declare the array like this, then use C's *strcmp* function to compare the identifier **value** to a given keyword **keywords[i]**:

   ```
   char* keywords[] = {"and", "break", …, "True", "while"};
   ```

3. Recognize string literals, much like identifiers are handled. Be careful to handle the case where the closing quote or double-quote may be missing. Run and test.
4. Recognize integer literals (ignore real literals for now); the isdigit(c) function is helpful. Run and test.
5. Extend your code for integer literals to look for a '.' that follows, and if so collect more digits and return a real literal instead. Run and test.
6. Turn your code for #4 and #5 above into a single function, such as collect_int_or_real_literal( ), or as separate functions. There's a special case of real literals starting with ".", such as ".525". Can you handle using your existing function(s)?
7. Recognize and discard comments; nuPython supports line comments that start with #.
8. Appendix A shows examples of test input with the corresponding correct output. The test files are "test01.py", "test02.py", and "test03.py".
9. Confirm your program does not contain memory errors by running valgrind, for example:
   - **make valgrind**
   - **make valgrind < test01.py**

Grading and Electronic Submission

You will submit your "scanner.c" file to Gradescope for grading. You should have received an invitation to join Gradescope in your NU email; if not, you can join by visiting Gradescope, signing up for a new account using your NU email and NetID, and providing this entry code: **4GRZ8B**.

When you are ready to submit for grading, do the following:

1. Open a terminal window
2. Change to your release directory
3. make submit

Keep in mind that Gradescope is a grading system, not a testing system. The idea is to give you a chance to improve your grade by allowing you to resubmit and fix errors you may have missed --- Gradescope is not meant to alleviate you from the responsibility of testing your work. For this reason, (1) Gradescope will not open until 2-3 days before the due date, and (2) submissions to Gradescope are limited to **4 submissions per 24-hour period**. A 24-hour period starts at midnight, and after 4 submissions, you cannot submit again until the following midnight; unused submissions do not carry over, you get exactly 4 per 24-hour period (including late days).

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your "Submission History". This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming the following requirements:

1. *You must provide a header comment at the top of "scanner.c", replacing the << WHAT… >> with meaningful text, your name, etc.*
2. *Definition and use of a function to collect integer literals, with header comment*
3. *Definition and use of a function to collect real literals (can be the same function as #2), with header comment*
4. *Definition and use of a function to collect string literals, with header comment*

Use the provided collect_identifier( ) function (see Appendix B) as a template and guide.

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity website. With regards to CS 211, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own,

whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.

Appendix A

test01.py

```
print(x)
{
  a = b
}
```

```
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('x') @ (1, 7)
Token 2 (')') @ (1, 8)
Token 5 ('{') @ (2, 1)
Token 25 ('a') @ (3, 3)
Token 13 ('=') @ (3, 5)
Token 25 ('b') @ (3, 7)
Token 6 ('}') @ (4, 1)
Token 0 ('$') @ (5, 1)
```

test02.py

```
print(count)

if x<y:
{
  z = 123
  _var = 0.5
}
else:
{
  s1 = 'this is a string'
  s2 = "and this has 123"
  s3 = "and I quote 'hi'"
  s4 = 'and "quote" within'
}

print("The loop yields:")

while x<y:
{
  print(x)
  print(y)
}
```

```
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('count') @ (1, 7)
Token 2 (')') @ (1, 12)
Token 34 ('if') @ (3, 1)
Token 25 ('x') @ (3, 4)
Token 16 ('<') @ (3, 5)
Token 25 ('y') @ (3, 6)
Token 21 (':') @ (3, 7)
Token 5 ('{') @ (4, 1)
Token 25 ('z') @ (5, 3)
Token 13 ('=') @ (5, 5)
Token 22 ('123') @ (5, 7)
Token 25 ('_var') @ (6, 3)
Token 13 ('=') @ (6, 8)
Token 23 ('0.5') @ (6, 10)
Token 6 ('}') @ (7, 1)
Token 31 ('else') @ (8, 1)
Token 21 (':') @ (8, 5)
Token 5 ('{') @ (9, 1)
Token 25 ('s1') @ (10, 3)
Token 13 ('=') @ (10, 6)
Token 24 ('this is a string') @ (10, 8)
Token 25 ('s2') @ (11, 3)
Token 13 ('=') @ (11, 6)
Token 24 ('and this has 123') @ (11, 8)
Token 25 ('s3') @ (12, 3)
Token 13 ('=') @ (12, 6)
Token 24 ('and I quote 'hi'') @ (12, 8)
Token 25 ('s4') @ (13, 3)
Token 13 ('=') @ (13, 6)
Token 24 ('and "quote" within') @ (13, 8)
Token 6 ('}') @ (14, 1)
Token 25 ('print') @ (16, 1)
Token 1 ('(') @ (16, 6)
Token 24 ('The loop yields:') @ (16, 7)
Token 2 (')') @ (16, 25)
Token 43 ('while') @ (18, 1)
Token 25 ('x') @ (18, 7)
Token 16 ('<') @ (18, 8)
Token 25 ('y') @ (18, 9)
Token 21 (':') @ (18, 10)
Token 5 ('{') @ (19, 1)
Token 25 ('print') @ (20, 3)
Token 1 ('(') @ (20, 8)
Token 25 ('x') @ (20, 9)
Token 2 (')') @ (20, 10)
Token 25 ('print') @ (21, 3)
Token 1 ('(') @ (21, 8)
Token 25 ('y') @ (21, 9)
Token 2 (')') @ (21, 10)
Token 6 ('}') @ (22, 1)
Token 0 ('$') @ (23, 1)
```

test03.py

```
#
# this is a comment
#
print(_count)
+-*/%**=!===
<=<>>=&:#123
123<123.456
(+123)
if(123==_123count456):{hi}#random (123) comment
{-9}
[ +0.5 123 .5 . ]  # the .5 is a real number, the . by itself is unknown
-12.
x=123456.7890012435
y=.5
z=5.
"Hey, the closing quote is missing!
and break continue def elif else false False for FOR if in is
None NONE none not or pass Pass return True while while123 _while
```

```
Token 25 ('print') @ (4, 1)
Token 1 ('(') @ (4, 6)
Token 25 ('_count') @ (4, 7)
Token 2 (')') @ (4, 13)
Token 7 ('+') @ (5, 1)
Token 8 ('-') @ (5, 2)
Token 9 ('*') @ (5, 3)
Token 12 ('/') @ (5, 4)
Token 11 ('%') @ (5, 5)
Token 10 ('**') @ (5, 6)
Token 13 ('=') @ (5, 8)
Token 15 ('!=') @ (5, 9)
Token 14 ('==') @ (5, 11)
Token 17 ('<=') @ (6, 1)
Token 16 ('<') @ (6, 3)
Token 18 ('>') @ (6, 4)
Token 19 ('>=') @ (6, 5)
Token 20 ('&') @ (6, 7)
Token 21 (':') @ (6, 8)
Token 22 ('123') @ (7, 1)
Token 16 ('<') @ (7, 4)
Token 23 ('123.456') @ (7, 5)
Token 1 ('(') @ (8, 1)
Token 7 ('+') @ (8, 2)
Token 22 ('123') @ (8, 3)
Token 2 (')') @ (8, 6)
Token 34 ('if') @ (9, 1)
Token 1 ('(') @ (9, 3)
Token 22 ('123') @ (9, 4)
Token 14 ('==') @ (9, 7)
Token 25 ('_123count456') @ (9, 9)
Token 2 (')') @ (9, 21)
Token 21 (':') @ (9, 22)
Token 5 ('{') @ (9, 23)
Token 25 ('hi') @ (9, 24)
Token 6 ('}') @ (9, 26)
Token 5 ('{') @ (10, 1)
Token 8 ('-') @ (10, 2)
Token 22 ('9') @ (10, 3)
Token 6 ('}') @ (10, 4)
Token 3 ('[') @ (11, 1)
Token 7 ('+') @ (11, 3)
Token 23 ('0.5') @ (11, 4)
Token 22 ('123') @ (11, 8)
Token 23 ('.5') @ (11, 12)
```

```
Token -1 ('.') @ (11, 15)
Token 4 (']') @ (11, 17)
Token 8 ('-') @ (12, 1)
Token 23 ('12.') @ (12, 2)
Token 25 ('x') @ (13, 1)
Token 13 ('=') @ (13, 2)
Token 23 ('123456.7890012435') @ (13, 3)
Token 25 ('y') @ (14, 1)
Token 13 ('=') @ (14, 2)
Token 23 ('.5') @ (14, 3)
Token 25 ('z') @ (15, 1)
Token 13 ('=') @ (15, 2)
Token 23 ('5.') @ (15, 3)
**WARNING: string literal @ (16, 1) not terminated properly
Token 24 ('Hey, the closing quote is missing!') @ (16, 1)
Token 26 ('and') @ (17, 1)
Token 27 ('break') @ (17, 5)
Token 28 ('continue') @ (17, 11)
Token 29 ('def') @ (17, 20)
Token 30 ('elif') @ (17, 24)
Token 31 ('else') @ (17, 29)
Token 25 ('false') @ (17, 34)
Token 32 ('False') @ (17, 40)
Token 33 ('for') @ (17, 46)
Token 25 ('FOR') @ (17, 50)
Token 34 ('if') @ (17, 54)
Token 35 ('in') @ (17, 57)
Token 36 ('is') @ (17, 60)
Token 37 ('None') @ (18, 1)
Token 25 ('NONE') @ (18, 6)
Token 25 ('none') @ (18, 11)
Token 38 ('not') @ (18, 16)
Token 39 ('or') @ (18, 20)
Token 40 ('pass') @ (18, 23)
Token 25 ('Pass') @ (18, 28)
Token 41 ('return') @ (18, 33)
Token 42 ('True') @ (18, 40)
Token 43 ('while') @ (18, 45)
Token 25 ('while123') @ (18, 51)
Token 25 ('_while') @ (18, 60)
Token 0 ('$') @ (19, 1)
```

Appendix B

```
/*scanner.c*/

//
// << WHAT IS THE PURPOSE OF THIS FILE??? >>
//
// << WHAT IS YOUR NAME >>
// << WHAT SCHOOL IS THIS >>
// << WHAT COURSE IS THIS >>
// << WHAT QUARTER IS THIS >>
//
// Starter code: Prof. Joe Hummel
//

#include <stdio.h>
#include <stdbool.h>  // true, false
#include <ctype.h>    // isspace, isdigit, isalpha
#include <string.h>   // strcmp
#include <assert.h>   // assert

#include "scanner.h"


//
// collect_identifier
//
// Given the start of an identifier, collects the rest into value
// while advancing the column number.
//
static void collect_identifier(FILE* input, int c, int* colNumber, char* value)
{
  assert(isalpha(c) || c == '_');  // should be start of an identifier

  int i = 0;

  while (isalnum(c) || c == '_')  // letter, digit, or underscore
  {
    value[i] = (char)c;  // store char
    i++;

    (*colNumber)++;  // advance col # past char

    c = fgetc(input);  // get next char
  }

  // at this point we found the end of the identifer, so put
  // that last char back for processing next:
  ungetc(c, input);

  // turn the value into a string:
  value[i] = '\0';  // build C-style string:
```

```
  return;
}


//
// scanner_init
//
// Initializes line number, column number, and value before
// the start of processing the input stream.
//
void scanner_init(int* lineNumber, int* colNumber, char* value)
{
  assert(lineNumber != NULL);
  assert(colNumber != NULL);
  assert(value != NULL);

  *lineNumber = 1;
  *colNumber = 1;
  value[0] = '\0';  // empty string
}


//
// scanner_nextToken
//
// Returns the next token in the given input stream, advancing the line
// number and column number as appropriate. The token's string-based
// value is returned via the "value" parameter. For example, if the
// token returned is an integer literal, then the value returned is
// the actual literal in string form, e.g. "456". For an identifer,
// the value is the identifer itself, e.g. "print" or "y". For a
// string literal such as 'hi class', the value is the contents of the
// string literal without the quotes.
//
struct Token scanner_nextToken(FILE* input, int* lineNumber, int* colNumber, char* value)
{
  assert(input != NULL);
  assert(lineNumber != NULL);
  assert(colNumber != NULL);
  assert(value != NULL);

  struct Token T;

  //
  // repeatedly input characters one by one until a token is found:
  //
  while (true)
  {
    //
    // Get the next input character:
    //
    int c = fgetc(input);

    //
```

```
// Let's see what we have...
//

if (c == EOF || c == '$')  // no more input, return EOS:
{
  T.id = nuPy_EOS;
  T.line = *lineNumber;
  T.col = *colNumber;

  value[0] = '$';
  value[1] = '\0';

  return T;
}
else if (c == '\n')  // end of line, keep going:
{
  (*lineNumber)++;  // next line, restart column:
  *colNumber = 1;
  continue;
}
else if (isspace(c))  // other form of whitespace, skip:
{
  (*colNumber)++;  // advance col # past char
  continue;
}
else if (c == '(')
{
  T.id = nuPy_LEFT_PAREN;
  T.line = *lineNumber;
  T.col = *colNumber;

  (*colNumber)++;  // advance col # past char

  value[0] = (char)c;
  value[1] = '\0';

  return T;
}
else if (c == ')')
{
  T.id = nuPy_RIGHT_PAREN;
  T.line = *lineNumber;
  T.col = *colNumber;

  (*colNumber)++;  // advance col # past char

  value[0] = (char)c;
  value[1] = '\0';

  return T;
}
else if (c == '_' || isalpha(c))
{
  //
```

```
  // start of identifier or keyword, let's assume identifier for now:
  //
  T.id = nuPy_IDENTIFIER;
  T.line = *lineNumber;
  T.col = *colNumber;

  collect_identifier(input, c, colNumber, value);

  //
  // TODO: is the identifier a keyword? If so, return that
  // token id instead.
  //

  return T;
}
else if (c == '*')
{
  //
  // could be * or **, let's assume * for now:
  //
  T.id = nuPy_ASTERISK;
  T.line = *lineNumber;
  T.col = *colNumber;

  (*colNumber)++;  // advance col # past char

  value[0] = '*';
  value[1] = '\0';

  //
  // now let's read the next char and see what we have:
  //
  c = fgetc(input);

  if (c == '*')  // it's **
  {
    T.id = nuPy_POWER;

    (*colNumber)++;  // advance col # past char

    value[1] = '*';
    value[2] = '\0';

    return T;
  }

  //
  // if we get here, then next char did not
  // form a token, so we need to put the char
  // back to be processed on the next call:
  //
  ungetc(c, input);

  return T;
```

```
    }
    //
    //
    // TODO: all the remaining tokens (punctuation, literals), and
    // also need to handle line comments.
    //
    //
    else
    {
      //
      // if we get here, then char denotes an UNKNOWN token:
      //
      T.id = nuPy_UNKNOWN;
      T.line = *lineNumber;
      T.col = *colNumber;

      (*colNumber)++;  // advance past char

      value[0] = (char)c;
      value[1] = '\0';

      return T;
    }

  }//while

  //
  // execution should never get here, return occurs
  // from within loop
  //
}
```