CSC173: Project 2
Recursive Descent and Table-Driven Parsing

The goal of this project is to demonstrate your understanding of the formal model of context-free grammars and parsing by applying the principles of the model to a specific grammar (details below).

The goal of the project is **not** simply for you to write a program that parses strings from the specific required grammar. The goal is for you to demonstrate how the formal model allows you build a parser based on the grammar **mechanically** with almost no thinking required.

Process not product.

Given a parsable grammar, for this project you must first construct a recursive-descent parser for the grammar using parsing functions, as seen in class and in the textbook. These parsing functions come almost directly ("mechanically") from the productions of the (parsable) grammar, as seen in class. You will demonstrate your parser by asking the user for input strings, parsing them, and printing the resulting parse tree (or an error message).

You must also implement a table-driven parser for the grammar, following the description seen in class and in the textbook, and demonstrate its use. Again: it's not about being a creative programmer. It's about understanding and applying the formal model. In this case, your code should be able to parse strings using **any** grammar given a parsing table connected to the productions of the grammar. But you will only need to show it working on one grammar.

Please read the requirements carefully.

Language

For this term, the language that we will be parsing is a very simple programming language.

Here is an informal description of the language:

- An **program** is zero or more **statement**s, each followed by a semicolon.
- There are only two types of statements:
  - Assignment ("="): assign the value of an expression to a variable
  - Output (">"): print the value of an expression on the console
- **Variables** are named with a single letter ("a" to "z").
- **Expressions** use unsigned integers, the four arithmetic operators, and parentheses, as seen in class and in the textbook.
  - The expression "<" (less than) reads and returns a number from the console.
- **Whitespace** (spaces, tabs, newlines) is not allowed. The entire input must be on one line without a newline at the end.

Here are some well-formed expressions of this language (one per line):

```
x=1;
>123+1+2/2*3;
a=<;b=<;>(a+b)/2;
```

The last of these is actually a useful program. You should be able to figure out what it does. If not, please go to study session ASAP.

In order to build a parser, we need a context-free grammar for the language and in order to use the required parsing methods (seen in class and in the textbook), that grammar must be parsable by a recursive descent or table-driven parser.

Think about this yourself.

After you have thought about it yourself, please read Appendix A. Note that you **MUST** use the grammar given in this document.

[1] I know that this makes the strings hard to read and would not be good for a real programming language. However allowing it without making the grammar impossibly complicated involves using a *lexical analyzer* (a.k.a. *scanner* or *tokenizer*). This is mentioned in a footnote on p. 617 of the textbook, and you will see it if you go on to take CSC254.

Requirements

Part 1: Recursive-descent parser (60%)

Implement a recursive-descent parser that produces parse trees for expressions of the required language.

- **You MUST use the grammar given in this document.**
- The style of the parsing functions **MUST** be as seen in class (which is sort of like what is in the textbook; check out the code available on BlackBoard with the project description).
- Your parsing functions **MUST** use functions `lookahead` and `match` as seen in class.
- You should be able to create the parsing functions by reading the productions of the grammar with almost no thinking required.
- The only place where thinking is required is how to use the lookahead symbol, as seen in class and in the textbook.
- This will be long if you do it right. I'm sorry.

You must then demonstrate your parser by reading strings from the user, parsing them, and printing the resulting parse tree.

- Your program must prompt the user for input.
- You may assume that expressions are on a single line of input, and no longer than 255 characters. Note that the empty string is a valid input, so you should use `fgets` rather than `scanf` to read a line of input. See the "C for Java Programmers" guide and/or ask in study session if you need help with this.
- If the input is not well-formed, your parser should print an error message and resume processing at the next line of input.
- Otherwise print the parse tree as described below.
- This phase of the program should terminate when it reads the string "`quit`" (without the quotes).

Part 2: Table-driven parser (40%)

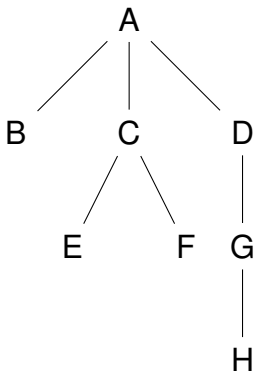
Implement a table-driven parser for expressions of the required language.

- As for Part 1: read expressions from the user, try to parse the input, print the parse tree or an error message, until "`quit`".
- Most of the infrastructure of the parser will be the same as for Part 1.
- **You MUST use the grammar given in this document.**
- You **MUST** use an explicit parsing table that references explicitly-represented productions (FOCS Figs. 11.31 and 11.32). That means the table must contain either indexes of productions in a list or array of productions (as seen in the textbook), or references (pointers) to the productions themselves.
- You **MUST** have a function that creates and returns an instance of this table for the grammar of the required language. This function does **not** need to **translate** a grammar into a parsing table. It just needs to build and return the parsing table for the grammar that you are using. So work it out by hand, then write the code to produce it.
- You **MUST** have a parsing function that takes a parsing table and an input string as arguments and does a table-driven parse. (You may have helper functions also.) Note that this function should be able to parse using **any** grammar, given its parsing table and productions. That is, it is **independent** of the grammar that it is using (but you only need to show it working with the required grammar).
- It may be helpful to produce output like FOCS Fig. 11.34 during debugging.

The next step would be to convert the parse trees produced by your parser(s) to *expression trees* and then either *evaulate* the expressions to compute their values or *generate code* to compute the values later. It's not hard to do that once you've built the parse trees, but you do **not** have to do it for this project. You will see it in CSC254 if you take that course.

Your project **MUST** be a single program. This program should read the input and then call each parser (assuming you do both parts) and print the results, as described above. It is your responsibility to make it clear to us what your program is doing.

There is no opportunity for extra credit in this project.




```
A
  B
  C
    E
    F
  D
    G
      H
```

Figure 1: Example parse tree and corresponding printed output

Parse Trees and Printing Parse Trees

A parse tree is a dynamic data structure. You have seen trees in Java and they're the same in C. The textbook has an entire chapter on trees (Chapter 5), and the chapter for this unit has useful code also (also available on BlackBoard with the project description).

For this project, your program must print the resulting parse tree to standard output. There are many ways to do this, but for this project you will produce output in an *indented, pretty-printed format*. What this means is:

- Each node is printed on a separate line.
- The children of a node are indented relative to their parent.
- All children of a node are at the same level of indentation.
- The empty string symbol epsilon "ε" can be printed in C using by using a Unicode escape: `"\u03B5"`. If that doesn't work for you, just print the word "`empty`".

Printing a tree involves doing a tree traversal, right? Traversing a tree is a recursive procedure, right? You print nodes and you recursively print their children, in the right order. So you can implement it using a recursive function. It's elegant *and* practical.

You also need to keep track of the current indentation level. So this will be a parameter to your pretty-printing function. In C, which does not have function overloading, this usually means two functions: a toplevel pretty-print function with no indentation parameter, and a helper function with that parameter, called from the toplevel function with indentation 0 to get the ball rolling.

If this did not make sense to you, please get to study session ASAP.

Additional Requirements and Policies

The short version:

- You **must** use the following C compiler options:
  `-std=c99 -Wall -Werror`
- If you are using an IDE, you **must** configure it to use those options (but I suggest that you take this opportunity to learn how to use the command-line).
- You **must** submit a ZIP including your source code and a README by the deadline.
- You **must** tell us how to build your project in your README.
- You **must** tell us how to run your project in your README.
- Projects that do not compile will receive a grade of **0**.
- Projects that do not run or that crash will receive a grade of **0** for whatever parts did not work.
- Late projects will receive a grade of **0** (see below regarding extenuating circumstances).
- You will learn the most if you do the project yourself, but collaboration is permitted in teams of up to three (3) students.
- Do not copy code from other students or from the Internet.

Detailed information follows...

Programming Requirements

C programs **must** be written using the "C99" dialect of C. This means using the "`-std=c99`" option with `gcc` or `clang`. For more information, see Wikipedia.

You **must** also use the options "`-Wall -Werror`". These cause the compiler to report all warnings, and to make any warnings into errors that prevent your program from compiling. You **must** be able to write code without warnings in this course.

With these settings, your program should compile and run consistently on any platform. We will deal with platform-specific discrepancies as they arise.

If you are using an IDE (Eclipse, XCode, VSCode, CLion, *etc.*), you **must** ensure that it will also build as described above. The easiest way to do that is to setup the IDE with the required compiler options. There are some notes about this in the C Programming Resources (for CSC173 and beyond) area.

You may **NOT** use `#pragma`'s in your programs. This includes `#pragma`'s added by an IDE or any other way that they might get into your code.

You **SHOULD** test your program with the memory checking program `valgrind`. If you don't know what `valgrind` is or why it is A Good Thing, read C for Java Programmers Chapter 11: Debugging a C Program.

Programs that do not receive a clean report from valgrind have problems that **should be fixed** whether or not the program sometimes runs properly on some platforms. The only exception is unfreed memory errors (so-called "memory leaks"). We will not penalize you for those in CSC173.

If your program does not work for us, the first thing we're going to do is run valgrind on it. If valgrind reports errors: you have errors in your program. Period.

It is easy to run valgrind in a virtual machine or Docker container if you cannot install it natively. Mac and Windows users can install Docker Desktop. See How to run Linux (including valgrind) in Docker for more information about Linux and Docker.

For help with `valgrind`, please go to study session **well before** the project deadline.

Submission Requirements

You **must** submit your project as a ZIP archive of a folder (directory) containing the following items:

1. A file named `README.txt` or `README.pdf` (see below)
2. The source code for your project (do not include object files or executables in your submission)
3. A completed copy of the submission form posted with the project description.

The name of the folder in ZIP **must** include "CSC173", "Project 1" (or whatever), and the NetID(s) of the submitters. For example: "`CSC173_Project_1_aturing`"

Your README **must** include the following information:

1. The course: “CSC173”
2. The assignment or project (*e.g.*, “Project 1”)
3. Your name and email address
4. The names and email addresses of any collaborators (per the course policy on collaboration)
5. Instructions for building your project (with the required compiler options)
6. Instructions for running your project

The purpose of the submission form is so that we know which parts of the project you attempted and where we can find the code for some of the key required features.

- **Projects without a submission form or whose submission form does not accurately describe the project will receive a grade of 0**.
- If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

Project Evaluation

You **must** tell us in your README how to build your project and how to run it.

Note that we will NOT load projects into Eclipse or any other IDE. We **must** be able to build and run your programs from the command-line. If you have questions about that, go to a study session.

We **must** be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better your grade will be.** It is **your** job to make the building of your project easy and the running of its program(s) easy and informative.

For C projects, the most common command for building a program from all the C source files in the directory (folder) is:

```
gcc -std=c99 -Wall -Werror -o EXECUTABLE *.c
```

where `EXECUTABLE` is the name of the executable program that we will run to execute your project.

You may also tell use to build your project using `make`. In that case, be sure to include your `Makefile` with your submission. You **must** ensure that your `Makefile` sets the compiler options appropriately.

If you expect us to do something else, you **must** describe what we need to do in your `README` file. This is unlikely to be the case for most of the projects in CSC173.

Please note that we will **NOT** under any circumstances edit your source files. That is your job.

Projects that do not compile will receive a grade of 0. There is no way to know if your program is correct solely by looking at its source code (although we can sometimes tell that is incorrect). This is actually an aspect of a very deep result in Computer Science that we cover in CSC173.

We will then run your program by running the executable, or as described in the project description. If something else is required, you **must** describe what is needed in your `README` file.

Projects that do not run or that crash will receive a grade of 0 for whatever parts did not work. You earn credit for your project by meeting the project requirements. Projects that do not run **do not** meet the requirements.

Any questions about these requirements: go to study session **BEFORE** the project is due.

Late Policy

Late projects will receive a grade of 0. You **must** submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

I assume that you are in this course to learn. You will learn the most if you do the projects **yourself**.

That said, collaboration on projects is permitted, subject to the following requirements:

- Teams of no more than 3 students, all currently taking CSC173.
- You **must** be able to explain anything you or your team submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the team should submit code on the team's behalf in addition to their writeup. Other team members **must** submit a README (only) indicating who their collaborators are.
- All members of a collaborative team will get the same grade on the project.

Working in a team only works if you actually do all parts of the project together. If you only do one part of, say, three, then you only learn one third of the material. If one member of a team doesn't do their part or does it incorrectly (or dishonestly), all members pay the price.

Academic Honesty

I assume that you are in this course to learn. You will learn nothing if you do not do the projects **yourself**.

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

The use of generative AI tools is **NOT** permitted in this course.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.

A Parsable Grammar of Simple PL Expressions

You need an unambiguous grammar of Simple PL expressions and your grammar needs to be parsable by recursive descent.

Here is the informal definition of the language again:

- An **program** is zero or more **statement**s, each followed by a semicolon (like Java).
- There are only two types of statements:
  - Assignment (“=”): assign the value of an expression to a variable
  - Output (“>”): print the value of an expression on the console
- **Variables** are named with a single letter (“a” to “z”).
- **Expressions** use unsigned integers and the four arithmetic operators, as seen in class and in the textbook.
  - The expression “<” (less than) reads and returns a number from the console.
- **Whitespace** (spaces, tabs, newlines) is not allowed. The entire input must be on one line without a newline at the end.

Try to write a grammar for this language. Don’t worry about whether it is parsable by recursive descent. Just try to get the productions right, following the definitions as closely as possible.

For programs, you need to figure how to recursively describe “zero or more statements each of which is followed by a semicolon.”

For expressions and numbers, use the approach seen in class and in the textbook.

DO IT NOW BEFORE GOING ON TO THE NEXT PAGE.

A program (P) is zero or more statements (S), each followed by a semicolon. So a program can be the empty string (zero statements), or "S `;`" (one statement), or "S `;` S `;`" (two statements), and so on.

Thinking recursively: a program can be the empty string, or it can be a statement followed by a semicolon followed by another program (which is zero or more statements):

$$P \rightarrow S \texttt{;} P \mid \varepsilon$$

Note that I have written the two bodies in the opposite order from the English description since empty productions are traditionally written last (sort of like a "default" or "otherwise" clause in a `switch` statement).

And then a statement (S) is either an assignment statement or an output statement:

$$S \rightarrow V = E \mid > E$$

You should be able to convince yourself that this part of the grammar is unambiguous and parsable by a recursive descent or table-driven parser.

The definitions of expressions and numbers are just like those seen in class and in the textbook. A variable name is also a legal expression in this language.

$$\begin{aligned}
E &\rightarrow E + E \mid E - E \mid E * E \mid E \,/\, E \mid (\, E \,) \mid N \mid V \\
N &\rightarrow N\ D \mid D \\
D &\rightarrow \texttt{0} \mid \cdots \mid \texttt{9} \\
V &\rightarrow \texttt{a} \mid \cdots \mid \texttt{z}
\end{aligned}$$

Ask yourself if this part of the grammar is unambiguous and parsable by a recursive descent or table-driven parser. How do you know? What do you need to be able to do for each syntactic category for a recursive descent parser? What about for a table-driven parser?

If it is not, try to make it so. See FOCS sections "Unambiguous Grammars for Expressions" (pp. 613–615) and "Making Grammars Parsable" (pp. 631–633).

DO THAT NOW BEFORE GOING ON TO THE NEXT PAGE.

The grammar of expressions (E) is **ambiguous**. It does not enforce the precedence of the operators. It is also **left-recursive**.

The productions for numbers (N) are left-recursive.

You should know why those are bad things for a recursive-descent or table-driven parser. If it's not clear: please go to study session.

TRY TO FIX THEM YOURSELF.

Really: Go think about it. Read the textbook. Try it. Then go to the next page.

For the expressions, use the unambiguous grammar of arithmetic expressions shown in the textbook (FOCS Fig. 11.22). We introduce new syntactic categories for each level of precedence, yielding expressions (E), terms (T), and factors (F). Terms are combined using $+$ and $-$ (lowest precedence). Factors are combined using $*$ and $/$ (highest precedence).

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T \,/\, F \mid F \\ F &\rightarrow (\, E\,) \mid N \mid V \end{aligned}$$

For the numbers (N), eliminate the left recursion by rearranging the order of the constituents in the recursive production as seen in FOCS Example 11.14:

$$N \rightarrow D\,N \mid D$$

You should understand why this will not change the interpretation of numbers, although it will require different code to convert the parse tree for a number into the number that was parsed.

Does that make the grammar of expressions parsable by a recursive descent or table-driven parser?

THINK ABOUT IT BEFORE YOU GO TO THE NEXT PAGE.

For the expressions, the productions are still left-recursive.

For the numbers, as noted in FOCS (p. 631): “Unfortunately, the productions of Example 11.14 are still not parsable by our methods.”

Fixing both of these problems involves introducing a new syntactic category for the “tail” of a string in some syntactic category, as seen in FOCS Example 11.15.

For the numbers, we need to introduce a new syntactic category for “the tail of a number,” which is the part that comes after the first digit. I will use the category “M” for this:

$$\begin{aligned} N &\rightarrow D\ M \\ M &\rightarrow N \mid \varepsilon \end{aligned}$$

You should be able to see how to select the right production for M based on a single lookahead symbol. If not: please go to study session.

To make the unambiguous grammar of expressions parsable, we need to realize that an expression is “one or more terms, separated by $+$ or $-$” and a term is “one or more factors, separated by $\star$ or $/$.” In other words, an expression is a term followed by zero or more occurrences of $+$ or $-$ with another term. Similarly, a term is a factor followed by zero or more occurrences of $\star$ or $/$ with another factor.

TRY DOING IT YOURSELF BEFORE YOU GO TO THE NEXT PAGE.

We introduce syntactic category Z for "the tail of an expression," which is the part that comes after the first term. Similarly with U for "the tail of a term," which is the part that comes after the first factor:

$$
\begin{aligned}
E &\rightarrow T\ Z \\
Z &\rightarrow \texttt{+}\ T \mid \texttt{-}\ T \mid \varepsilon \\
T &\rightarrow F\ U \\
U &\rightarrow \texttt{*}\ F \mid \texttt{/}\ F \mid \varepsilon
\end{aligned}
$$

The productions for factor (F), number (N), and digit (D) are unchanged.

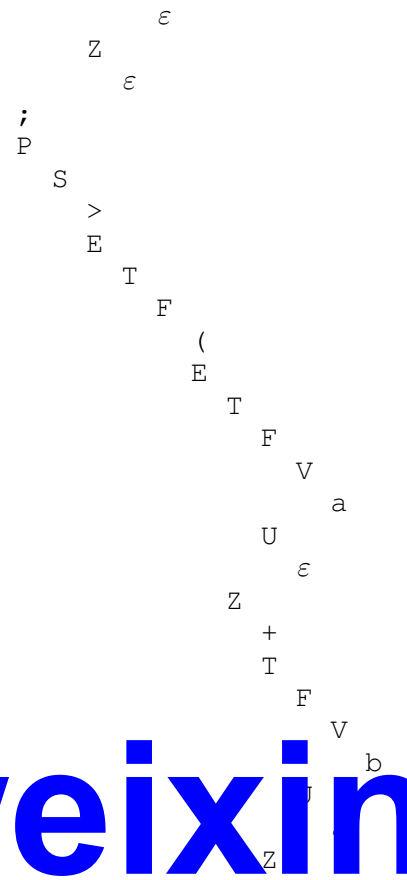
The final parsable grammar for Simple PL is:

$$
\begin{aligned}
P &\rightarrow S\ \texttt{;}\ P \mid \varepsilon \\
S &\rightarrow V\texttt{=}E \mid \texttt{>}\ E \\
E &\rightarrow T\ Z \\
Z &\rightarrow \texttt{+}\ T \mid \texttt{-}\ T \mid \varepsilon \\
T &\rightarrow F\ U \\
U &\rightarrow \texttt{*}\ F \mid \texttt{/}\ F \mid \varepsilon \\
F &\rightarrow \texttt{(}\ E\ \texttt{)} \mid V \mid N \\
N &\rightarrow D\ M \\
M &\rightarrow N \mid \varepsilon \\
D &\rightarrow \texttt{0} \mid \cdots \mid \texttt{9} \\
V &\rightarrow \texttt{a} \mid \cdots \mid \texttt{z}
\end{aligned}
$$

You MUST use this grammar for your project.

B Example Transcript

```
Enter input to test ("quit" to quit): x=1;
Testing "x=1;"
P
  S
    V
      x
    =
    E
      T
        F
          N
            D
              1
            M
              ε
        U
          ε
      Z
        ε
  ;
  P
    S
      V
        a
      =
      E
        T
          F
            <
          U
            ε
      Z
        ε
  ;
  P
    S
      V
        b
      =
      E
        T
          F
            <
          U
```




```
Enter input to test ("quit" to quit):
Testing ""
P
  ε
Enter input to test ("quit" to quit): quit
Bye
```