CSC173: Project 2
Grammars and Parsing

Requirements

1. Implement a recursive-descent parser for the grammar of arithmetic expressions seen in class (FOCS Figure 11.2 or an appropriate variant).
   - Your parser should read successive expressions from standard input, construct a parse tree for the expression if it is well-formed, and then print that parse tree to standard output (see description below).
   - If the input is not well-formed, your parser should print an appropriate message and resume processing at the next line of input (*i.e.*, skip to the next newline).
   - You may assume that expressions are on a single line of input if you want, or deal with multi-line input.
   - Your parser should terminate on end-of-file on standard input.
2. Implement a table-driven parser for the grammar of arithmetic expressions. Read expressions from standard input, try to parse the input, print the parse tree or an error message. Most of the infrastructure of the parser will be the same as for Part 1. You must have an explicit parsing table that references explicitly-represented productions (FOCS Figs. 11.31 and 11.32). It may be helpful to produce output like FOCS Fig. 11.34 during debugging.
3. Turn your parser into a calculator by *evaluating* the parse tree produced by your parser. That is, write code to traverse a parse tree computing the value of subexpressions as appropriate and printing the final result. You may use either version of your parser—they should produce equivalent parse trees. The result is a command-line calculator like the UNIX program `bc`.

You must implement all three parts of the project as a single program named `expr`. This program should read the input, call both parsers and the evaluator (if you do all three parts), and print the answers informatively. It is your responsibility to make it clear to us what your program is doing.

Note: You can, of course, implement the parts one at a time. That's probably a good strategy. In your main loop, only call the parts that you have implemented so far.

Extra Credit Ideas (10% each; max 20% extra credit)

1. Add variables, assignment statements, and the use of variables in expressions to your system.
2. Add builtin functions (`sin`, *etc.*).
3. Add user-defined functions.

FYI: *The UNIX Programming Environment* by Kernighan and Pike, Chapter 8, describes building such a program using a compiler-generator (`yacc`, now superceded by `bison`). Of course you're writing the parser by hand, but some of the ideas might be useful for these extra credit parts of the project.

Grammars for Arithmetic Expressions

Here is FOCS Fig 11.2, written more concisely:

$$
\begin{aligned}
<D> &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
<N> &\rightarrow <D> \mid <N>\ <D> \\
<E> &\rightarrow <N> \mid \texttt{(}\ <E>\ \texttt{)} \mid <E>\ \texttt{+-}\star\texttt{/}\ <E>
\end{aligned}
$$

Ask yourself if this grammar is parsable by a recursive descent parser. You'd want to be able to figure this out, so give it a try. If it is parsable, then you are all set. If not, think about what you have to do to make it parsable. See FOCS p. 624 and the section "Making Grammars Parsable" pp. 631–633. The appendix at the end of this document has more help, but **don't read it until until you have tried to figure this out for yourself.**

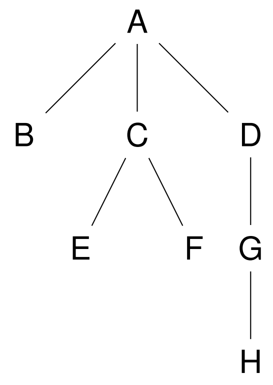
Parse Trees and Printing Parse Trees

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 if you read the chapter for this unit I promise you that you will find all kinds of useful code.

For parts 1 and 2 of 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.

Here is an example:



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

As for the code, printing a tree involves doing a pre-order tree traversal. So it's a recursive function, right? First you print the node, then you print its children in order.

You also need to keep track of the current indentation level. So this will be a parameter to your 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 an internal function with that parameter, called from the toplevel function with indentation 0 to get the ball rolling.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) How to build your project
   (b) How to run your project's program(s) to demonstrate that it/they meet the requirements
   (c) Any collaborators (see below)

   (d) Acknowledgements for anything you did not code yourself (you should avoid this, other than the code we've given you, which you don't have to acknowledge)

2. All source code for your project, including a `Makefile` or shell script that will build the program per your instructions. Do not just submit Eclipse projects without this additional information.

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 grade your will be.**

Programming Policies

You must write your programs using the "C99" dialect of C. This means using the "`-std=c99`" option with `gcc` or `clang`. For more information, check 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 should be able to write code without warnings in this course.

With these settings, your program should compile and run consistently on any platform.

Furthermore, your program must pass `valgrind` with no error messages. Of course this might differ for different executions, but it should be clean for any fixed examples requested in the project description. Any questions, ask the TAs *early*.

Projects in this course will be evaluated using `gcc` on Fedora Linux (recent version, like 25 or 26). Here are several ways you can setup this environment for yourself:

- Visit `getfedora.org` and download and install Fedora on a spare computer or separate partition of your main computer.
- Don't have a spare computer? No problem. Visit `virtualbox.org` and download VirtualBox. Then setup a new virtual machine running Fedora Linux (you may need to download the installer as above). VMWare is a commercial virtualization app if you want to buy something.
- Not confident setting up a VM or don't have space? No problem. Visit `docker.com` and download and install Docker for your computer. You can easily setup an image with Fedora Linux and `gcc`, or see the appendix below for more.

- Don’t want to install anything? No problem. You need an account on the Computer Science Department’s undergraduate cycle servers. Ask the TAs about it. Do not wait until the last minute. This will not happen overnight.

Mac and Windows users especially beware: You must test your code under Fedora Linux, or expect problems (and low grades).

Late Policy

Don’t be late. But if you are: 5% penalty for the first hour or part thereof, 10% penalty per hour or part thereof after the first.

Collaboration Policy

You will learn the most if you do the projects YOURSELF.

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

- Groups of no more than 3 students, all currently taking CSC173.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor’s or TA’s discretion.
- One member of the group should submit on the group’s behalf and the grade will be shared with other members of the group. Other group members should submit a short comment naming the other collaborators.
- All members of a collaborative group will get the same grade on the project.

Appendix: Using Docker

“Docker provides operating-system-level virtualization [. . .] to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.” (Wikipedia)

Getting Started with Docker

I have built a docker image based on Fedora linux and containing `gcc`, `make`, `gdb`, `valgrind`, and `emacs`, as well as manpages. You can download it from here:

http://www.cs.rochester.edu/u/ferguson/csc/docker/

Once you have installed Docker, load the image to create the `fedora-gcc` image in your Docker installation:

```
docker load -i fedora-gcc-docker.tgz
```

Then you can do, for example:

```
docker run -it --rm -v "`pwd`":/home -w /home fedora-gcc bash
```

This runs `bash` (the command shell) using the `fedora-gcc` image. The `-it` option means to run interactively; `-rm` tells Docker to remove the container when the process exits; `-v` and the bit with the colon says to make your current working directory (outside docker) available on `/home` within docker; and `-w` tells docker to make `/home` the initial working directory for the `bash` process. Whew!

You can do a lot with Docker. You'll have to read the docs but it's a great option. Please report any problems with the image ASAP.

Appendix: R-D-Parsable Grammar of Arithmetic Expressions

You should try to figure this out for yourself before reading this section.

Really: Go think about it. You should know how to make a grammar parseable by a recursive-descent parser.

Then go to the next page if you still can't figure it out by yourself.

FOCS Fig 11.2:

$$\begin{aligned}
<D> &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
<N> &\rightarrow <D> \mid <N>\ <D> \\
<E> &\rightarrow <N> \mid \texttt{(}\ <E>\ \texttt{)} \mid <E>\ \texttt{+-*/}\ <E>
\end{aligned}$$

Equivalent factored, right-recursive grammar that is parsable by a recursive-descent parser:

$$\begin{aligned}
<E> &\rightarrow <T>\ <TT> \\
<TT> &\rightarrow \texttt{+-}\ <T>\ <TT> \mid \epsilon \\
<T> &\rightarrow <F>\ <FT> \\
<FT> &\rightarrow \star\texttt{/}\ <F>\ <FT> \mid \epsilon \\
<F> &\rightarrow <N> \mid \texttt{(}\ <E>\ \texttt{)} \\
<N> &\rightarrow <D>\ <NT> \\
<NT> &\rightarrow <N> \mid \epsilon \\
<D> &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9}
\end{aligned}$$

Note that evaluating the expressions produced by this grammar is a bit more involved than the nice almost-expression trees produced by the original grammar. But you should be able to walk the tree and evaluate it anyway. Try a couple of expressions and look at the parse trees.