CSC173: Project 2
Grammars and 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** 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 **entirely mechanically** (almost no thinking required).

Process not product.

Given a grammar, you will first need to ensure that it is parsable by recursive descent. If not, you will need to convert it into a parsable grammar.

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.

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 as long as it is provided as a parsing table connected to the productions of the grammar. But you will only need to show it working on one grammar.

$$
\begin{aligned}
\langle E\rangle &\rightarrow \langle E\rangle \ \texttt{\&} \ \langle E\rangle \ \mid \ \langle E\rangle \ \texttt{|} \ \langle E\rangle \ \mid \ \texttt{\textasciitilde} \ \langle E\rangle \ \mid \ \texttt{(}\,\langle E\rangle\,\texttt{)} \ \mid \ \langle S\rangle \\
\langle S\rangle &\rightarrow \langle S\rangle \ \langle B\rangle \ \mid \langle B\rangle \\
\langle B\rangle &\rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

Figure 1: Grammar of simple bitstring expressions

Grammar

The grammar in Figure 1 describes expressions of a simple algebra for bitstrings (strings of bits: `0`s and `1`s).

- An expression $\langle E\rangle$ may use one of three operators, as well as parentheses:
  - The operator "`~`" is the unary bitwise logical NOT operator. It has the highest precedence.
  - The operator "`&`" is the binary bitwise logical AND operator.
  - The operator "`|`" (vertical bar) is the binary bitwise logical OR operator. It has the lowest precedence. Do not confuse the operator "`|`" with the meta-symbol "|" used to separate the bodies of productions with the same head.
  - FYI: These operators actually exist in C.
- A bitstring $\langle S\rangle$ is a non-empty sequence of bits.
- A bit $\langle B\rangle$ is either a `0` or a `1`.

Here are some well-formed expressions of this grammar:

```
1
101010
~010
~~100
10&01
1|00&111
(101|001)&(11|00)
~(101|010)
```

Requirements

Part 1: Recursive-descent parser (60%)

Implement a recursive-descent parser that produces parse trees for a grammar of bit-string expressions.

- The style of the parsing functions **must** be as seen in class (which is sort of like what is in the textbook). This is not a creative programming exercise. I'm sorry.
- 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 boring 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.
- If the input is not well-formed, your parser should print an appropriate 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 a grammar of bitstring expressions.

- 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 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 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 your grammar of bitstring expressions. 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 can parse *any* grammar given its parsing table and productions.
- 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 *evaulate* the expressions to compute the value of the expression. It's not hard to do that if you've built the parse trees properly, but you do **not** have to do it for this project.

Your project **MUST** be a single program named `bits`. 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.

FYI: *The UNIX Programming Environment* by Kernighan and Pike, Chapter 8, describes building a similar program (for arithmetic expressions) 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 of interest, if not immediately useful.

There is no opportunity for extra credit in this project.

Grammars for Bitstrings

Here is the grammar of bitstrings again:

$$\begin{aligned}
\langle E\rangle &\rightarrow \langle E\rangle \;\texttt{\&}\; \langle E\rangle \;\mid\; \langle E\rangle \;\texttt{|}\; \langle E\rangle \;\mid\; \texttt{\~{}}\,\langle E\rangle \;\mid\; \texttt{(}\,\langle E\rangle\,\texttt{)} \;\mid\; \langle S\rangle \\
\langle S\rangle &\rightarrow \langle S\rangle\ \langle B\rangle \mid \langle B\rangle \\
\langle B\rangle &\rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}$$

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 sections "Unambiguous Grammars for Expressions" (pp. 613–615) and "Making Grammars Parsable" (pp. 631–633).

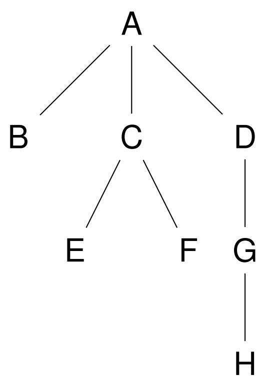
The appendix at the end of this document has more help, but **don't read it until until you have tried to figure this out for yourself**.

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 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.



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

Figure 2: Example parse tree and corresponding printed output

Printing a tree involves doing a tree traversal, right? What kind of traversal? Think about it. . . Traversing a tree is a recursive procedure, right? You print nodes and you 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.

Additional Requirements and Policies

The short version:

- You **must** use 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 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 any 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.

Furthermore, your program should pass `valgrind` with no error messages. If you don’t know what this means or why it is A Good Thing, look at the C for Java Programmers document which has a short section about it. Programs that do not receive a clean report from `valgrind` have problems that **should be fixed** whether or not they appear to run properly. If your program does not work for us, the first thing we’re going to do is run valgrind on it.

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 (details below).

The name of the folder in ZIP **must** include "CSC173", "Project 1", 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 don't run don't 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 don't 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.

There is code out there for all these projects. You know it. We know it.

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.

Appendix: Parsable Grammar of Bitsring Expressions

Here is the original grammar of bitstring expressions:

$$
\begin{aligned}
\langle E\rangle &\rightarrow \langle E\rangle \;\texttt{\&}\; \langle E\rangle \;\mid\; \langle E\rangle \;\texttt{|}\; \langle E\rangle \;\mid\; \texttt{\~{}}\;\langle E\rangle \;\mid\; \texttt{(}\;\langle E\rangle\;\texttt{)} \;\mid\; \langle S\rangle \\
\langle S\rangle &\rightarrow \langle S\rangle\;\langle B\rangle \mid \langle B\rangle \\
\langle B\rangle &\rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}
$$

The productions for $\langle E\rangle$ do not enforce the precedence of the operators (the grammar is ambiguous). They will need to be re-written. See "Unambiguous Grammars for Expressions" (FOCS pp. 613–615).

Some of the productions for $\langle E\rangle$ and $\langle S\rangle$ are left-recursive. That will have to be eliminated if the grammar is to be parsable by recursive descent. See "Making Grammars Parsable" (FOCS pp. 631–633) which includes Examples 11.14 and 11.15.

Try to do it yourself.

Really: Go think about it. Read the textbook. Try it.

Go to the next page if you want some help.

Remove the ambiguity by introducing new syntactic categories for each level of precedence:

$$\begin{aligned}
\langle E\rangle &\rightarrow \langle E\rangle\ \texttt{|}\ \langle T\rangle \mid \langle T\rangle \\
\langle T\rangle &\rightarrow \langle T\rangle\ \texttt{\&}\ \langle F\rangle \mid \langle F\rangle \\
\langle F\rangle &\rightarrow \texttt{!}\ \langle F\rangle \mid \texttt{(}\ \langle E\rangle\ \texttt{)} \mid \langle S\rangle \\
\langle S\rangle &\rightarrow \langle S\rangle\ \langle B\rangle \mid \langle B\rangle \\
\langle B\rangle &\rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}$$

That looks after the precedence of the operators.

Now eliminate the left recursion by rearranging the order of constituents in those productions and then left-factor by introducing "tail" categories $\langle ET\rangle$, $\langle TT\rangle$, and $\langle ST\rangle$:

$$\begin{aligned}
\langle E\rangle &\rightarrow \langle T\rangle\ \langle ET\rangle \\
\langle ET\rangle &\rightarrow \texttt{|}\ \langle E\rangle \mid \epsilon \\
\langle T\rangle &\rightarrow \langle F\rangle\ \langle TT\rangle \\
\langle TT\rangle &\rightarrow \texttt{\&}\ \langle T\rangle \mid \epsilon \\
\langle F\rangle &\rightarrow \texttt{\~{}}\ \langle F\rangle \mid \texttt{(}\ \langle E\rangle\ \texttt{)} \mid \langle S\rangle \\
\langle S\rangle &\rightarrow \langle B\rangle\ \langle ST\rangle \\
\langle ST\rangle &\rightarrow \langle S\rangle \mid \epsilon \\
\langle B\rangle &\rightarrow \texttt{0} \mid \texttt{1}
\end{aligned}$$

Note that eliminating the left recursion by rearranging the constituents will change the way expressions involving those operators are grouped. This is not a problem for bitwise AND and OR.

The result is an $LL(1)$ grammar[1] of simple bitstring expressions that is parsable by recursive descent.

Note that using the parse trees produced by this grammar is more involved than the nice almost-expression trees produced by the original grammar. But it can be done, although you don't have to do it for this project.

[1]See the solutions for Homework 2.3 when they are available for more about parsable grammars.