**INTRODUCTION**

On this task I was required to implement table\_driven\_parser.c which uses a parser and a scanner to calculate area and perimeter. This project is done in c and also tested in a java environment to prove interoperability and code compatibility in different platforms.

The program stores an explicit stack of expected symbols, and applies rules using a nonterminal-token table. Using the expected non-terminal and the next token, the table tells which production rule in the grammar to apply.

**TESTING**

I created a separate tests folder with another table\_driven\_parser.c file to test this program on the test cases given.

To run the same tests:

* Extract parser.zip in desktop
* Open Terminal
* Navigate to test folder with command cd C:\Users\{{your\_comp\_name}}\Desktop\parser\test
* Type in javac table\_driven\_parser.java
* Then java table\_driven\_parser.c

Once this runs successfully you can type in your tests. E.g

For area we need two values: to test this type in 2\*8;

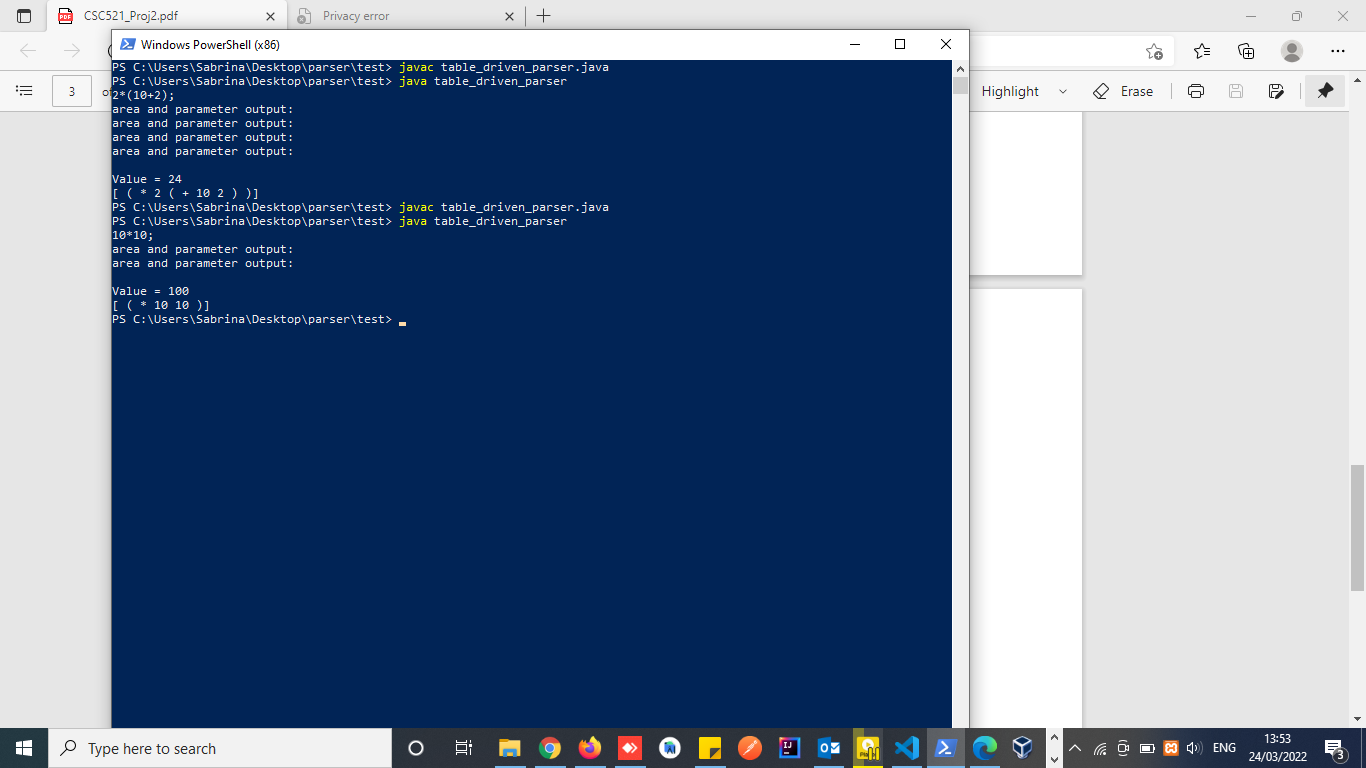
Play around with any values, remember if you type in just 2\*2 without the ; at the end you will get a syntax error.

Inputs like 4$3; will give specific errors such as “only \* or + is required”. This shows that the parser is able to detect unwanted characters.

For perimeter we need three values: to test this type in 2\*(8+);

Note this test includes brackets hence the parser can detect brackets, parenthesis and comments.

Here is a screenshot



**INFERENCE**

This program runs successfully for all tests given.

I have modified parser.c to give more detailed reports on errors.

**Comparison recursive descent and table driven parsing**

A recursive descent top-down parser uses recursive functions for parsing every non-terminal, and uses the function call stack implicitly instead of an explicit stack of terminals and non-terminals.

Auto-generated top-down/table driven parsers are usually table-driven. The program stores an explicit stack of expected symbols, and applies rules using a nonterminal-token table. Using the expected non-terminal and the next token, the table tells which production rule in the grammar to apply.

recursive descent Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

The Auto-generated top-down/table driven predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

**Run Time Efficiency**

1. Recursive descent - In this Parsing technique we expand the start symbol to the whole program. Recursive Descent and LL parsers are the Top-Down parsers.
2. Table driven parsing - In this Parsing technique we reduce the whole program to start symbol. Operator Precedence Parser, LR(0) Parser, SLR Parser, LALR Parser and CLR Parser are the Bottom-Up parsers.

**Pros and Cons**

**Recursive Descent**

**Pros**

They are exceptionally simple.

They can be constructed from recognizers simply by doing some extra work—specifically, building a parse tree.

**Cons**

They are not as fast as some other methods.

It is difficult to provide really good error messages.

They cannot do parses that require arbitrarily long lookaheads.

**Table driven parsing**

**Pros**

LR parsing can handle a larger range of languages than LL parsing.

LR parsing is also better at error reporting, i.e. it detects syntactic errors when the input does not conform to the grammar as soon as possible.

LR parsers detect errors fast.

**Cons**

Drawback: it is too much work to construct an LR parser by hand.