CMPT 432 - Spring 2025 - Dr. Labouseur

CST and Test Cases for Parser

Tyler DeLorey
tyler.delorey1@marist.edu

March 17th, 2025

Contents

1 Introduction

1.1 How the Parser and CST Work

Before showing the test cases and how I created my Parser for this grammar, I want to briefly explain how I created the Parser in the first place. Below shows the creation of the Parser class and the generation of the Concrete Syntax Tree (CST).

```
63          // PARSER
64          if (errors > 0 || tokens.size() == 0)
65          {
66              log("INFO", "Parsing for Program #" + to_string(i + 1)
      + " skipped due to Lex error(s)");
67              continue;
68          }
69
70          Parser currentParse = Parser(i + 1, tokens, delimiter);
71          currentParse.parse();
72          errors = currentParse.getErrors();
73
74          cout << endl;
75
76          // CST
77          if (errors > 0)
78          {
79              log("INFO", "CST for Program #" + to_string(i + 1) + "
    skipped due to Parse error(s)");
80              currentParse.deleteCST();
81              continue;
82          }
83
84          log("INFO", "CST for Program #" + to_string(i + 1));
85          currentParse.printCST();
86          currentParse.deleteCST();
```

Figure 1.1: Parsing and CST Generation (main.cpp)

After lexing, the program goes straight into parsing the current program. If there were any errors during lex, that means it won't continue to the parsing step and it will skip to the next program if there is any (Lines 64-68). If there were no lex errors, the compiler will continue to the parsing step (Lines 70-72). Since this grammar is a LL(1) grammar (left-to-right reading of the source code, construction of left-most derivation for CST, and looking only token ahead), that means a recursive decent parser is fairly simple to implement here, which is what I do. The recursive decent parser validates the tokens that were lexed in the previous step. I won't explain most of it since it is mostly straight-forward, but there are some parts of it that I will explain in the next section.

If there were no errors in parsing, it will continue to print the valid CST. The CST gets printed in a way where it neatly shows the branches and the leafs of the tree. The tokens themselves will be the leafs of the CST (the terminals), while the branches are the non-terminals that are part of the grammar. The grammar for this language can be found in the grammar.pdf file in this repository.

1.2 Test Files for Parsing

The repository has a ton of test cases in the testFiles folder, mostly from Lex, but there are also some test files specifically for testing the Parsing and the CST generation. More tests will be added when the semantic analysis and code generation steps are completed, but for now, the testFiles folder contains a lot of tests dealing with really dumb edge cases for the Lexer and some cases for the Parser. Section 3 will explore some of these test cases specifically for parsing.

2 Special Comments for Parsing

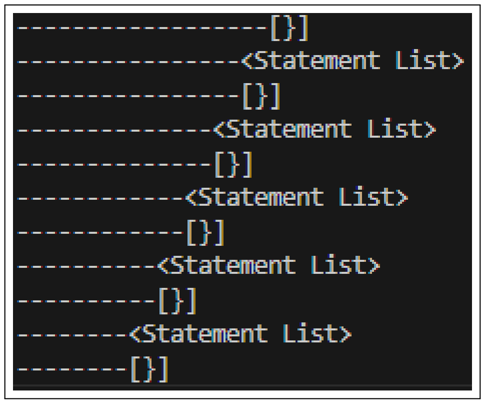
2.1 Recursive Decent Parser

A recursive decent parser is a set of recursive functions that processes the input tokens. Each function corresponds to a non-terminal in the grammar and calls other functions based on the production rules. For an LL(1) grammar, the parser can always decide the next rule to apply by looking ahead one token. This ensures deterministic parsing without any backtracking.

This isn't anything necessarily special about the recursive decent parser for this grammar. Each non-terminal has its own function, and the match() function matches the current token with the set of expected tokens. There are some epsilon productions in this grammar, but they don't cause much of a problem, but it may cause a problem in readable CST generation, which will be explained next.

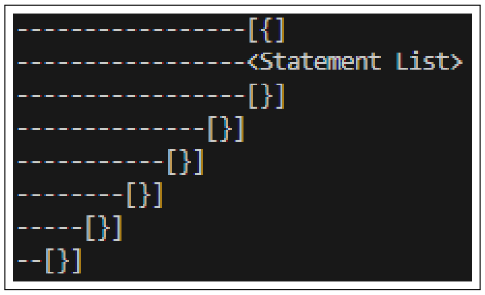
2.2 CST

There isn't much special with the CST generation. A branch node is added every time a non terminal function is called, and at the end of that function, the moveUp() function is called to move the current node pointer up the tree. Also, a leaf node that links to the current token is added every time a successful match occurs. However, there is one special occasion that occurs when adding branches to the CST. Whenever a non-terminal function is called from a separate function (not a recursive call), it will add the new non-terminal as a branch to the CST. However, if it function is recursively called and it becomes an epsilon production, that means the branch will not be added. This is purely for readability for the CST. This only occurs twice in this grammar, in StatementList and CharList. Those are the only two non-terminals that lead to epsilon productions. For example, some programs would end in a lot of StatementList branches that lead to epsilon production if there were a lot of blocks. Below is a snippet of how it would look like originally vs how it would look like with better readability with these branches remove.



```
------------------[}]
----------------<Statement List>
----------------[}]
--------------<Statement List>
--------------[}]
------------<Statement List>
------------[}]
----------<Statement List>
----------[}]
--------<Statement List>
--------[}]
```

Figure 2.1: Example Original Output



```
-----------------[{]
-----------------<Statement List>
-----------------[}]
--------------[}]
-----------[}]
--------[}]
-----[}]
--[}]
```

Figure 2.2: Example New Output

As you can see, I removed the unnecessary branches (besides the branch corresponding to the first call to the non-terminal function, shown in Figure 2.2).

On the next page is the parseCharList function, which is one of the two functions that implement this change.

```
            firstCallChar = true;
            parseCharList();
```

Figure 2.3: Call to parseCharList in parseStringExpr (Parser.h)

```
436         void parseCharList()
437         {
438             log("DEBUG", "Parsing Char List...");
439
440             if (currentTokenType == "CHAR" || currentTokenType == "
    SPACE")
441             {
442                 myCST->addNode("branch", "Char List");
443                 firstCallChar = false;
444                 match(currentTokenType);
445                 parseCharList();
446                 myCST->moveUp();
447             }
448             // Empty branch is only added after initial call to
    parseCharList from parseStringExpr()
449             // also an    regarding parsing and CST
450             else if (firstCallChar)
451             {
452                 myCST->addNode("branch", "Char List");
453                 firstCallChar = false;
454                 myCST->moveUp();
455             }
456             else
457             {
458                 // nothing
459                 // its an
460                 // production
461             }
462         }
```

Figure 2.4: parseCharList (Parser.h)

For this example, if the token type is either a CHAR or SPACE, it is not an epsilon production so it will add the branch normally no matter if it was a recursive call or not (Lines 440-447). If it is an epsilon production, only add the branch if it came from a previous function (Lines 450-455). This is tracked from the firstCallChar variable, which is set to true before the function is called from the previous non-terminal (set on Line 384 in Figure 2.3). If neither of this two cases are true, that means it is a true epsilon production and no branch is added.

3 Test Cases

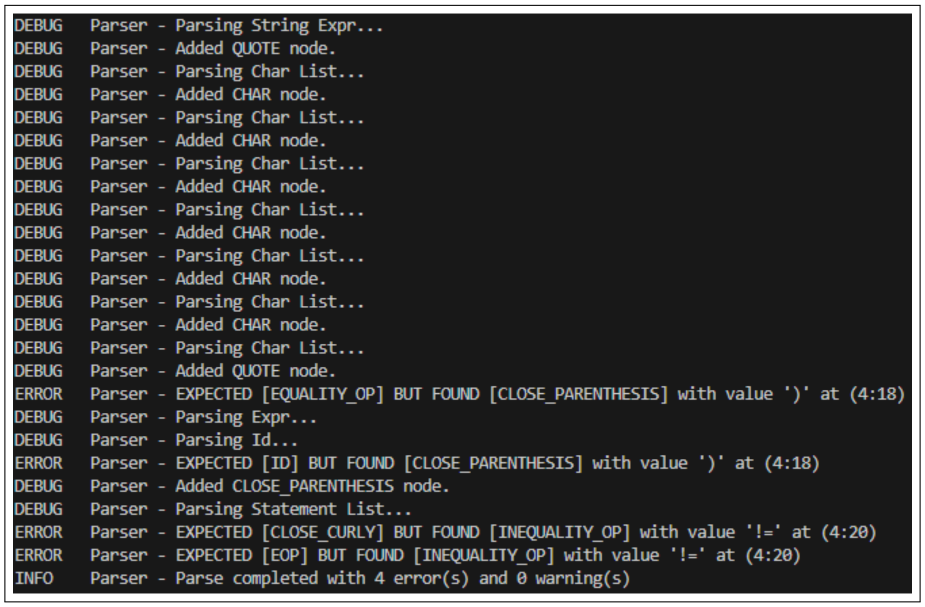
3.1 typeInsideString.txt

The first test case I will be going over is typeInsideString.txt. Most of these test cases just test whether or not the parser works. There are barely any edge cases involving parsing and the CST, which I have explained earlier.

```
/*  Type inside of a String */
{
    boolean d
    d = ("string") != "string")
}$
```

Figure 3.1: typeInsideString.txt

Below is a portion of the Parser output for this program.



```
DEBUG   Parser - Parsing String Expr...
DEBUG   Parser - Added QUOTE node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added CHAR node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added CHAR node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added CHAR node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added CHAR node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added CHAR node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added CHAR node.
DEBUG   Parser - Parsing Char List...
DEBUG   Parser - Added QUOTE node.
ERROR   Parser - EXPECTED [EQUALITY_OP] BUT FOUND [CLOSE_PARENTHESIS] with value ')' at (4:18)
DEBUG   Parser - Parsing Expr...
DEBUG   Parser - Parsing Id...
ERROR   Parser - EXPECTED [ID] BUT FOUND [CLOSE_PARENTHESIS] with value ')' at (4:18)
DEBUG   Parser - Added CLOSE_PARENTHESIS node.
DEBUG   Parser - Parsing Statement List...
ERROR   Parser - EXPECTED [CLOSE_CURLY] BUT FOUND [INEQUALITY_OP] with value '!=' at (4:20)
ERROR   Parser - EXPECTED [EOP] BUT FOUND [INEQUALITY_OP] with value '!=' at (4:20)
INFO    Parser - Parse completed with 4 error(s) and 0 warning(s)
```

Figure 3.2: typeInsideString Output

There is not much to explain here. Since there is invalid syntax (the boolean expression is all set up wrong according to the grammar), it throws a parse error. One thing that I think is important to note is that since the tokens are never re-aligned, there is a chance that there will be more extra error messages that are caused from each other. In order to know what was really wrong, it is a good idea just to look at the first error message that is thrown and work from there.

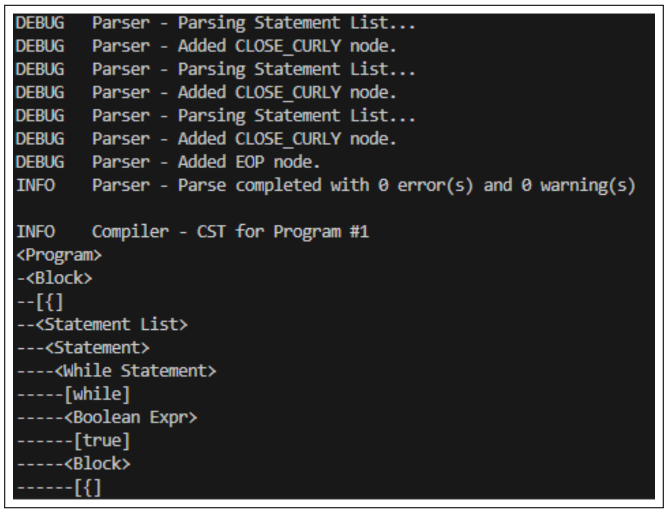
3.2 moveUpVerification.txt

The second test case I will be going over is moveUpVerification.txt. This just proves that the moveUp function works as intended for the CST

```

{
    while true {
        if (a == 2+b){
            a = b
        }
    }


}$
```

Figure 3.3: moveUpVerification.txt

Below is my Parse and CST output for this program. I won't show the entire output since it all works out correctly.



```
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Added EOP node.
INFO    Parser - Parse completed with 0 error(s) and 0 warning(s)

INFO    Compiler - CST for Program #1
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<While Statement>
-----[while]
-----<Boolean Expr>
------[true]
-----<Block>
------[{]
```

Figure 3.4: moveUpVerification Output

The output shows the end of the parsing debugging statements and the beginning of the CST creation. The tree format is very readable and the debugging statements can be turned off by turning off verbose mode in the Verbose.h header file.

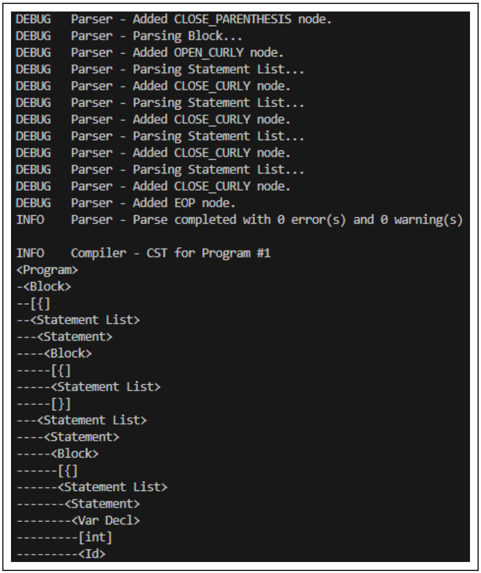
3.3 parseorama.txt

The third test case is parseorama.txt. This is a test case taken from a previous student online and is just an overall proof that parsing works as intended.

```

{/*this caused my compiler to go into an infinite loop*/
    {}
    {int astringb}
    {
    print(a)print(2)print(0+1)print(false)print(true)print("test")
    print((a==b))print((a!="tes t"))
    a = aa=bb=bb=1b=1+bb="te st"b=""b=falseb=true
    while true{
    if (5+a=="test"){

    }
    }

    }
    }$
```

Figure 3.5: parseorama.txt

Below is my Parse and CST output for this program.



```
DEBUG   Parser - Added CLOSE_PARENTHESIS node.
DEBUG   Parser - Parsing Block...
DEBUG   Parser - Added OPEN_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Added CLOSE_CURLY node.
DEBUG   Parser - Added EOP node.
INFO    Parser - Parse completed with 0 error(s) and 0 warning(s)

INFO    Compiler - CST for Program #1
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Block>
-----[{]
-----<Statement List>
-----[}]
---<Statement List>
----<Statement>
-----<Block>
------[{]
------<Statement List>
-------<Statement>
--------<Var Decl>
---------[int]
---------<Id>
```

Figure 3.6: parseorama Output

Even though the formatting in the file looks very messy, it is all valid syntax, so the parsing is successful and the CST is created (sorry that is image is small).

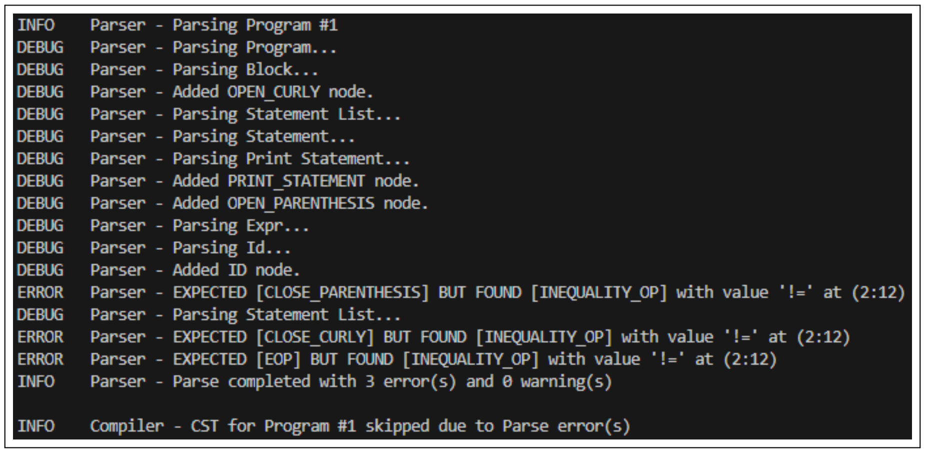
3.4 test3.txt

The last test case I will be going over is stupidTestCase.txt. This is a test case that I used for Lex, but it can also be used for parse because since Lex succeeds.

```
{
    print(a!/*Comment in between inequality*/=0)
}$
```

Figure 3.7: stupidTestCase.txt

Below is my Parse output for this program.



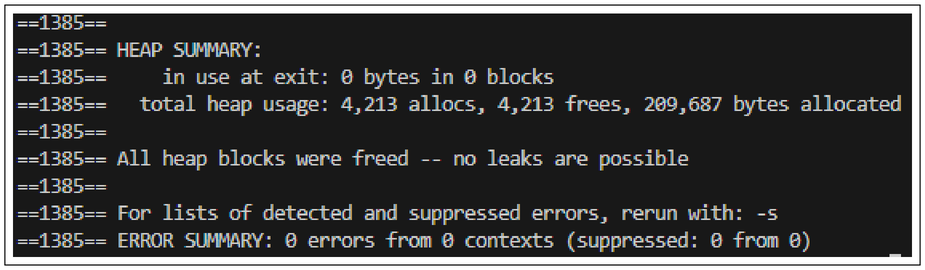
```
INFO    Parser - Parsing Program #1
DEBUG   Parser - Parsing Program...
DEBUG   Parser - Parsing Block...
DEBUG   Parser - Added OPEN_CURLY node.
DEBUG   Parser - Parsing Statement List...
DEBUG   Parser - Parsing Statement...
DEBUG   Parser - Parsing Print Statement...
DEBUG   Parser - Added PRINT_STATEMENT node.
DEBUG   Parser - Added OPEN_PARENTHESIS node.
DEBUG   Parser - Parsing Expr...
DEBUG   Parser - Parsing Id...
DEBUG   Parser - Added ID node.
ERROR   Parser - EXPECTED [CLOSE_PARENTHESIS] BUT FOUND [INEQUALITY_OP] with value '!=' at (2:12)
DEBUG   Parser - Parsing Statement List...
ERROR   Parser - EXPECTED [CLOSE_CURLY] BUT FOUND [INEQUALITY_OP] with value '!=' at (2:12)
ERROR   Parser - EXPECTED [EOP] BUT FOUND [INEQUALITY_OP] with value '!=' at (2:12)
INFO    Parser - Parse completed with 3 error(s) and 0 warning(s)

INFO    Compiler - CST for Program #1 skipped due to Parse error(s)
```

Figure 3.8: stupidTestCase Output

Because there is error in the syntax, again regarding the boolean expressions (needs an extra set of parenthesis), the CST is never printed for this program and it ends.

4 Conclusion

There are many more test cases that I have in the testFiles folder, but these are some I decided to chose to show. Parsing was a lot easier than Lex for me, since it either worked or it didn't. Below shows some valgrind output that proves that the program has no memory leaks and all bytes are allocated correctly



```
==1385==
==1385== HEAP SUMMARY:
==1385==     in use at exit: 0 bytes in 0 blocks
==1385==   total heap usage: 4,213 allocs, 4,213 frees, 209,687 bytes allocated
==1385==
==1385== All heap blocks were freed -- no leaks are possible
==1385==
==1385== For lists of detected and suppressed errors, rerun with: -s
==1385== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4.1: Valgrind Output