Section 1 Program: 1 sentence

The sources included in the file source.zip is in this directory.

Section 2 Teamwork: 2 paragraphs

During the semester, I was responsible for the majority of the code while my partner was responsible for the documentation as well as a series of tests for me to complete within my project. The semester was laid out to where we as individuals complete the entirety of the code base for the project while at the end of our semester, our partner writes some tests for me to complete. On top of that, my partner sent me his codebase for me to writeup a few pages of documentation explaining his thought process and ideas throughout the semester.

I received 3 tests to complete, the first test was a test for the tokenization, which made sure that the program would execute if EOF was at the start. The second test I received was a parsing test that made sure my program defaulted to the minus over the plus in the equation below. My third test was a parse print statement test to make sure strings correctly outputted.

```java
public void unterminatedStrings2(){
    assertTokensAre(EOF);
    assertTokensAre("\"asdf\"\"asdf", STRING, ERROR, EOF);
    assertTokensAre("\"asdf \"asdf\"", STRING, IDENTIFIER, ERROR, EOF);
}
```

```java
public void parseSubtractionExpressionWorks2() {
    AdditiveExpression expr = parseExpression("1 -+ 1");
    assertTrue(expr.isAdd());
}
```

```java
public void printStringStatement() {
    PrintStatement expr = parseStatement("print(“test”)");
    assertNotNull(expr);
}
```

Section 3 Design Pattern:

In computing, memorization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. A memoization function remembers the results corresponding to some set of specific inputs. It is used to lower a function's time cost in exchange for space cost, this means memorized functions become optimized for speed In exchange for a higher use of computer memory space.

As you can see below, we used a memoization technique for storing out results in a function call within our project. This topic was not talked about until towards the end of the class, but I enjoyed learning about the information about memoization and how it ultimately fits into our project.

-
  ```
  static Map<CatscriptType, CatscriptType> CACHE = new HashMap<>();
  public static CatscriptType getListType(CatscriptType type) {
      CatscriptType potentialMatch = CACHE.get(type);
      if(potentialMatch != null) {
          return potentialMatch;
      } else {
          ListType listType = new ListType(type);
          CACHE.put(type, listType);
          return new ListType(type);
      }
  ```

Section 4: Technical Writing:

My partner didn't send me his project code in time therefor I will be writing the documentation for my own project.

Expressions

For the expressions portion of the project, we created a hierarchy of expression classes that all had individual importance to complete and operate their own tasks. There are classes for every expression type within this folder such as Additive, Boolean, Equality, Factor, ListLiteral, Unary and many more. Each class contains variables, callers, validate and evaluate functions that are used in the main classes such as CatScriptParser in order to link and operate the expression and Literals portion of the project. These classes are all binded to the Statements portion by the ParseElement class.

Statements

For the statements portion of the project, there is a hierarchy of statement classes that are all called and maintained by the parent Statement class. These individual classes help our loops and iterative statements work correctly, for without them the language would not work very well.

- Forstatment: Contains series of get functions, as well as a compile, execute and bytecode compile. These are called within the CatScriptParser.
- IfStatement: Contains a series of Expression and list variables as well as get, set, validate, and execute functions. These are called within the CatScriptParser and other locations with the purpose of being able to properly use if statements within the language.
- FunctionCall and FunctionDefinition: Contains a List, String, CatscriptType and LinkedList variables with a series of get, execute, and validate functions. These are called all throughout the program and without we could not create or call functions in CatScript.
- VariableStatment: Contains an Expression, String, and a couple CatscriptType variables as well as some get, set validate and execute functions. This class is handled within the parseVarStatement function within the CatScriptParser class with the object of successfully parsing variable statements used within the program.

CatScriptParser

This class was the most important and time consuming throughout the project, inside it contains successful parsing for the Print, If, Var, For, FunctionCall, Identifier and every other Statement class. On top of that it parses the Expressions throughout the project. It is the main calling center for the other classes within the project and Is roughly 60% of why every test works throughout the project.

Tokenizer

The tokenizer portion of the project successfully tokenizes data within the project into a random string of characters, this serves as a reference to the original data but cannot be guessed. Within the project we are tokenizing numbers, strings, syntax, keywords, list literals and many more. I had the most fun on this portion of the project due to the fact that I have never worked with tokenization before, and It was an interesting learning experience to see how it works within our project and why it is important in the real world.

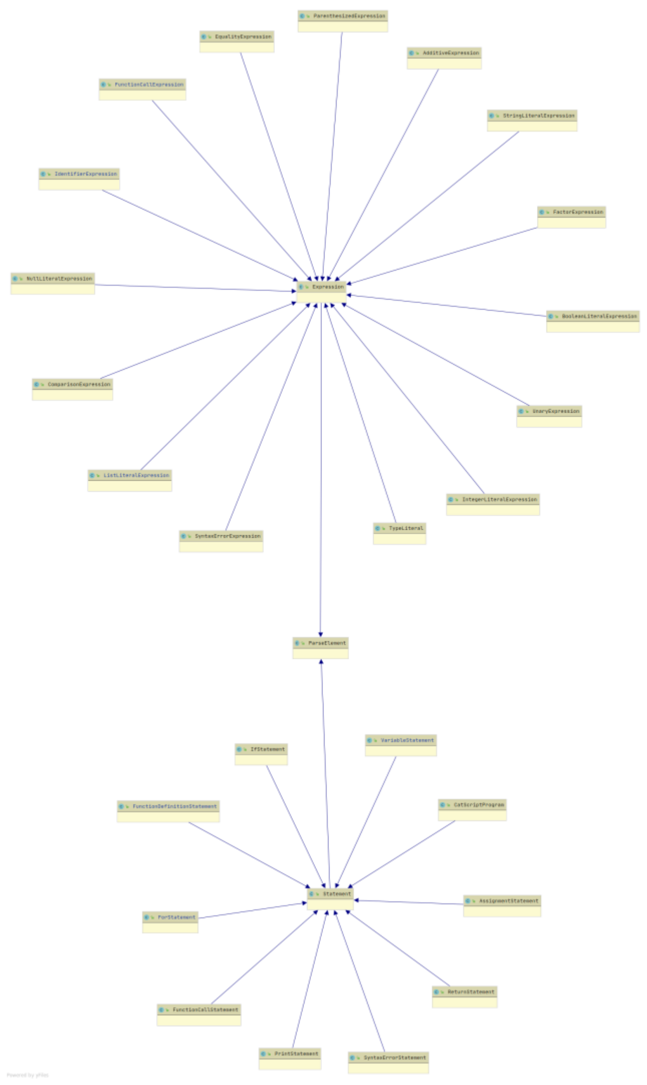
Bytecode

We used bytecode toward the end of the project in order to take our source code and compile it into low level code. I found this the most confusing part of the project and still don't really understand much about it, except for the fact that the code is recognized by the processor.

Section 5 UML:

UML was not a main focus within this class, but we did receive some UML overview images containing detailed layout of our program that we developed throughout the semester. As you can see below, we have a series of child classes all pointing towards either the Expression class or the Statement class. These two classes are used as parent callers that are binded via the ParseElement class.

Going a little bit more in-depth into the matter, we can see that the Expression and Statement classes are necessarily the brain of the overview, these two classes are what root variables and callings use in order to make the child classes operate and run correctly. This is the barebones of my project this semester, without everything individually linked the project would not work. I never saw the importance of main class linking until I took compilers. I see how Carson laid out the project in this manner and I think that this idea of throwing a heap of classes and codebase at us was a great learning experience as I have heard this is mostly how work will be like after college.


ParenthesizedExpression
EqualityExpression
AdditiveExpression
FunctionCallExpression
StringLiteralExpression
IdentifierExpression
FactorExpression
NullLiteralExpression
Expression
BooleanLiteralExpression
ComparisonExpression
UnaryExpression
ListLiteralExpression
IntegerLiteralExpression
SyntaxErrorExpression
TypeLiteral
ParseElement
IfStatement
VariableStatement
FunctionDefinitionStatement
CatScriptProgram
Statement
ForStatement
AssignmentStatement
ReturnStatement
FunctionCallStatement
PrintStatement
SyntaxErrorStatement
Powered by yFiles

Section 6: Design Tradeoff

A Parsing generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar. The generated code is a parser, which takes a sequence of characters and tries to match the sequence against the grammar. When it comes to a parser generator, there are its obvious pros and cons. The advantages are that it is quite easy to write a specification, in particular if the input format isn't too odd. We end up with an easy maintainable piece work that is easily understood. The downside of using a parser generator is that they are not very user friendly. This would mean that the system that we spent all this time working on would be very difficult for the consumer to use.

Using our hand written recurse descent parser, it was the better choice over the parser generator because while it was very tedious and complicated to produce, I have complete control over my parser that allows me to do all sorts of stuff that I couldn't do with a parser generator. After doing some research on the topic, I found hand-written parsers usually perform better than generated ones.

In my own opinion, I feel that Carson having us create our own hand-written parser rather than using a generated one was the better option for many reasons. Firstly, I feel that I have learned a lot more information than I ever would have if we skipped over that function of the project. Secondly and lastly, we had to option to make our recursive descent parser as powerful or well-written as time permitted. I feel that going down the generated parser route would be very linear, making the room for commitment and pride small.

Section 7: Software Development life cycle model.

Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before software is fully developed and tracking all software development by repeatedly testing the software against all test cases. Test structure of this process was laid out in the order of setup, execution, validation, and cleanup.

The benefits of using TDD is that programmers who wrote more tests tended to be more productive. Programmers using pure TDD on new projects reported they only rarely felt the need to use the debugger. TDD offers more than just simple validation of correctness but can also drive the design of the program. By focusing on the test cases first, one must imagine how the functionality is used by clients, therefor the programmer is concerned with the interface before the implementation.

How this project was developed was that Carson was the test-case creator while it was our job as the students to successfully complete these tests. I feel that this method of teaching throughout the semester was very productive. I never have had a class layout such as this one and I feel that throughout the trial and error of creating code to satisfy each individual test was cool and exciting. Overall, I feel that Carson did an amazing job with this teaching style due to the fact that I was told this is how the majority of the real workforce projects will be and I feel that it was great preparation for me before starting my career.