

Jared Weiss Capstone Submission

Section 1: Program

I have included a .zip (source.zip) file of the source files that I used to complete the final project for CSCI 468: Compilers at Montana State University.

Section 2: Teamwork

My partner in the teamwork section of this course was Kristoff Finley. Because we wrote this project mostly by ourselves, with little collaboration, our “teamwork” portion was submitting some new tests that our version of the language had to pass. We decided to implement a Logical Expression.

The logical expression consists of two expressions, a left hand side and a right hand side, and an operator. Both the expressions must evaluate to booleans, and the operator will be either “&&” or “||”. For example, “a && b” evaluates to true if and only if a is true and b is true.

The features we had to implement in order to get this to work were: tokenizing the “&&” operator, or the “||” operator, parsing a logical expression from those tokens, evaluating the expression and finally, compiling the expression. Each of these features were extracted into 4 different tests. You can find them all in the src/test directory in the repo.

Section 3: Design Pattern

One pattern that we used in this project was the Fly Wheel, or Memoization Pattern. The example can be found in /src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java in “getListType()”. Here is the source code before this pattern:

```
public static CatscriptType getListType(CatscriptType type) {  
    return new ListType(type);  
}
```

And source code after this pattern:

```
private static final Map<CatscriptType, ListType> cacheType = new HashMap<>();  
public static CatscriptType getListType(CatscriptType type) {  
    ListType listType = cacheType.get(type);  
    if (listType == null) {  
        cacheType.put(type, new ListType(type));  
        return cacheType.get(type);  
    }  
    return listType;  
}
```

First, some background information. getListType is a function that is used a lot. Because CatScript’s list types are distinct from just normal types, it is helpful to have this function to get a list of ints (for example), without writing some gronky code. Without the Memoization pattern, we would have to allocate a new CatScriptType every time we called the function, by basically just mapping Type to ListType. Here, we use a HashMap to cache types that we’ve already

queried before. This way, we're not allocating more instances of Type and ListType than absolutely necessary, and reusing memory we've already allocated.

Section 4: Technical Writing

The documentation written for CatScript is included in this project submission (CatScript-documentation.pdf). It goes over the basics of the language as well as a few features of this pretty simple scripting language.

Section 5: UML

The UML Diagram for the Parse Elements in CatScript is included with this project submission (CatScript-UML.pdf). In the UML you can see the general structure of how we parsed and worked with tokens in our compiler. It has a very nice tree-like structure to it. The architecture of this compiler is done very well. It made working within the project a nice and enjoyable experience.

As you can see, all expressions and statements inherit from ParseElement, so we can all expect them to have shared functionality, and we can expect all of them to do things that we want. The behaviour is *standardized* across the project, and it made it really easy to work within it. Also, by doing it this way, it makes for easy error collection.

Section 6: Design Trade-offs

The most important part of our design is the use of the Recursive Descent algorithm and parsing tokens in the language by hand rather than using a parser generator. Professor Carson Gross pointed out that most of our peers at other colleges are being taught compilers by learning a parser generator. While it would probably work, it would not be nearly as debuggable, and it would almost be like learning a new programming language to make our own programming language.

The advantages of learning the recursive descent algorithm were that we could learn how to write other parsers (not just programming languages), and adding a new feature (i.e. the logical expression) was as simple as adding a new function in the parser. It really was that easy. We didn't have to make some convoluted Regular Expression to parse the expression, or employ some radical new tools to do it. It's really simple, and it just works.

The problem with using a parser generator is you have no idea what happens behind the scenes, and it's impossible to debug. There's no way you'd be able to step through all the code that it generates, as a recent lecture showed us. It will generate 1000+ lines of code, just for a simple additive expression. I would highly recommend watching Carson's lecture on parser generators, it's quite enlightening. In my humble opinion, debuggability is key. Debuggable code is often easier to write, read and maintain. When writing debuggable code, it's easy to see your process, and where you're going. Debuggable code is readable, all the steps are laid out in front of you, and it's easier to maintain because you can *debug* it.

Parser generators are extremely convoluted and they *have* to be that way to generate code that achieves the users' goals, but it's so much more valuable to be able see what exactly the code is doing, and why it's doing it. The recursive descent algorithm is pretty and simple and it makes a lot of sense.

Section 7: Software Development Life Cycle Model

The model that we used was Test Driven Development (TDD). TDD is great. TDD is (in essence) writing all the tests that you expect the project to pass before you write *any* code. I had no idea that this was a development life cycle that was used at all. I was almost disappointed that I hadn't encountered it yet in my career. I have done a few internships and all of them wrote tests AFTER they wrote code.

While writing tests afterwards is all well and good, adding features is harder when you do it this way because the tests you wrote earlier may not encompass that new feature, and you have to adjust/write new tests. While the tests in our project may be flawed and miss some edge cases, it is usually easy to remedy the edge cases by adding a few more lines of code. TDD outlines what we expect the language to behave like, and generally makes sure it's rigorous. The way the language is supposed to behave is defined by the tests, we just needed to make sure that our compiler met those specs.

TDD also gives a real sense of progress throughout the project, because it feels good to get the tests to pass, and to know that the behaviour is (mostly) watertight. I used IntelliJ while developing CatScript, and seeing a cute little green checkmark pop up next to the test is extremely satisfying, and often kept me going throughout the project. All of Carson's classes have done it this way and it's extremely satisfying to get all the tests passing.

And just to quickly appeal to authority and in hopes that you may change some classes, I really think this is the way to go from a teaching standpoint. TDD gives metrics that professors can measure in assignments (just check if the test is passing), and it also precisely lays out what the project is supposed to be able to do (so students know what they need to do). I know this won't work in all classes, but, I'm a huge fan of the way that Carson does things and I think that the CS department would do well to take a leaf out of his book.