CSCI468 Compilers

Spring 2024

Elyse Dalager | Mason Watamura

Section 1: Program

The source code zip file is included in this directory as source.zip.

Section 2: Teamwork

Team member 1, myself, worked on 100% of the source code implementation of the recursive descent parser for the CatScript scripting language. Team member 2, Mason Watamura, was responsible for 100% of the documentation for the CatScript language and three tests. The documentation can be found in Section 4: Technical writing. The tests are included in the test/java/edu/montana/csci/csci468/demo/PartnerTests.java directory.

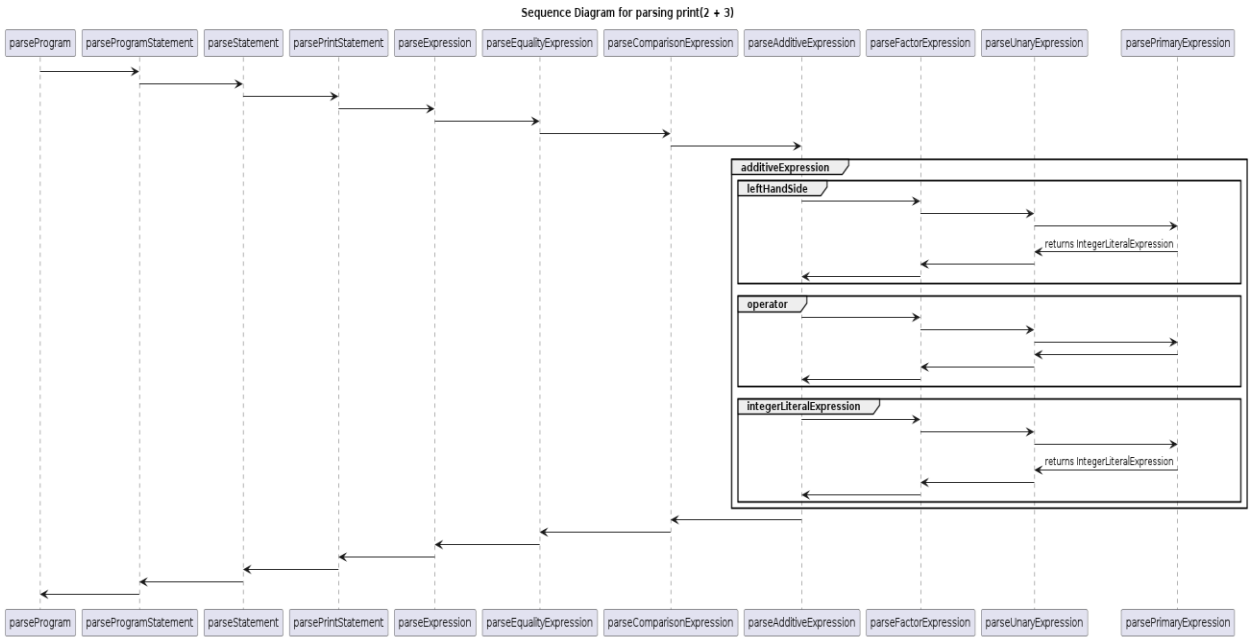
Section 3: Design pattern

The Memoization Pattern was used to memoize the getListType() function in main/java/edu/montana/csci/csci468/parser/CatscriptType.java. This pattern was used to optimize the execution of the parser by storing known list types in a hash map called cache, so we don't store elements and their respective data types more than necessary. The execution of the parser is faster with this technique and frees up memory to sufficiently implement the Memoization Pattern.

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

Section 4: Technical writing

Section 5: UML



Section 6: Design trade-offs

An alternative approach to creating a parser would be a parser generator. Parser generators are programs that take a grammar and grammar rules to code generate a language. Typically, it takes a lexical grammar containing Regular Expressions and a language grammar using Extended Backus-Naur Form (EBNF) as its two inputs. One may use tools like lex (a lexer generator) and yacc (yet another compiler compiler) to aid in this concept of method generation. A common parser generator in use today is known as Another Tool for Language Generation (ANTLR), which is very popular in the Java community.

In contrast, a recursive descent parser is a simple and obvious approach to creating a parser. It begins by taking an EBNF grammar that outlines the rules and regular expressions of the language. Developers are responsible for creating a method for each rule within the grammar, then calling each other method defined on the right-hand side of the production and matching strings as needed. The GNU Compiler Collection (GCC), C#, and Python compilers all use this approach to parsing!

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project. TDD is a software development methodology that requires tests to be written before the code is implemented. Its goal is to ensure that the code is running as expected and meets any specified requirements. TDD aided in building this recursive descent parser by creating an efficient and accurate way to test the code; the test base was immediately available to test any progress in the parser's

implementation. It also establishes expectations defined by each test to ensure the code is working properly and behaves as expected. For these reasons, I enjoyed using Test Driven Development to implement a recursive descent parser.