CSCI 468, Compilers

Spring 2024

Edward Aldeen, Sam Bierens

Section 1: Program

Source code attached in /capstone/portfolio/source.zip directory.

Section 2: Teamwork

Team member #1 implemented all code in the source according to the tests in **/src/test/java/edu/montana/csci/csci468/**. This took team member #1 16 weeks. Team member #2 created three additional parser tests to help test the code and created the technical documentation. This took team member #2 one week. These tests are in **/src/test/java/edu/montana/csci/csci468/parser/PartnerTest.java**. Team member #1 did 95% of the work.

Section 3: Design Pattern

The design pattern utilized in the source code was memoization. This can be seen in **/src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java**, in the **getListType()** method.

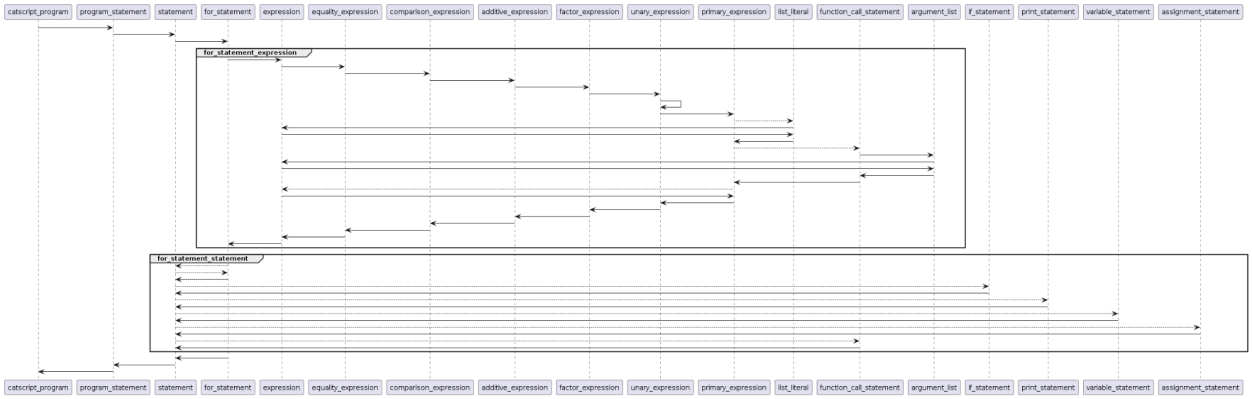
```
37    private static Map<CatscriptType, CatscriptType> listTypeCache = new HashMap<>();
38    public static CatscriptType getListType(CatscriptType type) {
39        if (listTypeCache.containsKey(type)) {
40            return listTypeCache.get(type);
41        } else {
42            ListType listType = new ListType(type);
43            listTypeCache.put(type, listType);
44            return listType;
45        }
46    }
```

This pattern was used to optimize the speed at which the type of a list is determined. This allows us to increase efficiency of the compiler and it optimizes storage. Since getting the type of a list is inefficient, memoization will save processing power and allow time and effort to be redirected to other, more inefficient, methods and functions.

Section 4: Technical Writing

Team member #2 wrote the technical documentation, which can be found in **/capstone/portfolio/CatscriptDocumentation.md**

Section 5: UML



The for loop sequence diagram in this UML diagram represents the flow of control and interactions involved when executing a for loop in CatScript. The sequence begins when a *for statement* is encountered in the CatScript program. The program moves to the expression component, where the condition for the for loop is evaluated. This involves evaluating the expressions within the for loop, starting from *expression* and recursively descending through the grammar.

When the condition specified in the expression evaluates to true, the program moves to the block of statements enclosed within the for loop statement. This represents the execution of the statements inside the loop body. Within the loop body, there can be recursive calls to other statements, such as another *for statement* or any other type of *statement*. This recursion allows for the execution of multiple statements within the loop body. This continues until the loop condition evaluates to false, the program exits the loop, and the sequence for the for loop ends.

This UML diagram removes a significant amount of the potential expression calls within the program, but it makes the sequence diagram unreadable thus only the crucial statement, expression, and recursive calls are left in.

Section 6: Design trade-offs

In Catscript, the parser was implemented using recursive descent parsing rather than a parser generator. This is because recursive descent parsing provides us with greater control over the parsing process. The parsing logic can easily be adjusted to suit the specific needs and

features of Catscript. This allows the parser, and in turn the compiler, to follow the grammar almost exactly. This results in the compiler code that is significantly more legible compared to a parser generator, as each parsing function corresponds directly to a rule in the grammar. Another important reason to use recursive descent parsing over a dedicated parser generator is that a complete implementation of a parser generator in the given time is not feasible and unnecessary. Catscript is a simple language with relatively simple grammar, and to use a parser generator for it would be overkill. Thus recursive descent was used as its simplicity offered competitive performance while remaining straightforward to implement and debug.

Section 7: Software development life cycle model

The model used to develop and test our capstone was test driven development. This model was crucial in making the compiler function as it gave clear cut requirements of what functions must compile correctly, and how they should function. Due to the numerous tests for each function, the tokenizer, the parser, and the bytecode generator, it was easy to determine whether or not each section functioned as it should. These tests also helped catch any regression or side effects due to changing the code. For example a method when changed in the bytecode checkpoint, failed a separate test in the parser, which the tests caught, saving a large amount of time. Test driven development also helps with step by step implementation as instead of writing a hundred lines of untested code, implementation is done test by test. This forces a focus on building on each success, and debugging on each regression or failure. The demerits of the test driven development methodology is that should there not be a test for a function or it not be thorough enough, that function can easily be implemented poorly or not at all causing issues that fail to show up properly when debugging. An example of this would be the *function* related methods in the parser and the bytecode checkpoints, as they lacked thorough tests, or lacked any tests at all. This caused strife as these methods were required for a good majority of the remaining tests, without having dedicated tests for themselves. Overall, if I could use test driven development again, I would, as it is a very simplistic, yet thorough approach to implementing something as complicated as a compiler.