CSCI 468 - Compilers

Garrett Mullings

Jack Hayward

Section 1: Source Code

Our source code can be found in the /capstone/portfolio/source.zip directory, which is the same directory as this paper.

Section 2: Teamwork

Team member 1 created the code to satisfy the tests given to us, as well as those created by team member 2. This included tokenization, parsing, evaluation, and finally compilation into bytecode. The split between creating code and creating tests allowed for a form of testing as close to true black box testing as possible for our situation, which allowed for new bugs to be caught that slipped through the original test suite. Team member 1 contributed approximately 60% of the total time spent on this project.

Team member 2 created the technical documentation for this project, which consists of the catscript user manual, which is found under section 4 of this document. The catscript user manual covers all of the features and capabilities of catscript, such as functions, the type system, try/catch error handling, and control structures. This manual will allow anyone with at least a basic level of programming knowledge to easily learn and write with catscript. The user manual features code snippets everywhere they may be helpful, along with an explanation of the code and the snippets output. This allows users of catscript to better learn the syntax, and better understand how to write with catscript.

Team member 2 also assisted in the test driven development process, creating an additional test suite for my catscript compiler. The goal of this test suite is to go beyond the base test suites provided in order to find bugs that a programmer using our compiler might find. This included else-if statements and nested statements such as nesting for loops and if statements. This ensured the compiler allows some of the more complex tasks programmers frequently must complete. These additional tests were particularly helpful as they pointed out a significant issue- catscript supported if statements, else statements, but not if-else statements, which are essential to any scripting language for the control flow to be complete. Team member 2 contributed approximately 40% of the time spent on this project.

Section 3: design pattern

The design pattern used is known as memoization. Memoizations main benefit is reducing memory usage, improving efficiency of the program. Memoization is implemented in the type system, specifically in the getListType function. This function is used to get the type of a list. Without memoization, the function simply returns a new instance of ListType every call:

```
Return new ListType(type);
```

However, if the programmer uses 2 lists of the same type, the compiler will create 2 identical instances of the ListType. This can be resolved with memoization.

Memoization is essentially the practice of caching common values, objects, etc. to avoid having excess copies of the same thing. This is memoizing the value. In this instance, a hash map is used to store all instances of ListType. When one is needed, the compiler will first look into the hashmap to see if the needed ListType already exists. If it does, it will just reuse it by returning the ListType found in the map. If it does not find the needed list type, that means it is the first time it has been used and a new one will be created. Then, it will be added to the hashmap to allow it to be reused if needed, and then the new ListType will be returned. The full function with memoization is shown below, and it can also be found in the CatscriptType.java file.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>(); //Create the hash map
public static CatscriptType getListType(CatscriptType type) {
    if (cache.containsKey(type)) { //check if the hashmap already contains what we
need
        return cache.get(type); //if so, just return it
    } else { //otherwise, we need a new ListType
        ListType listType = new ListType(type); //create the ListType
        cache.put(type, listType); //add it to the cache (memoize it)
        return listType; //now return it
    }
}
```

Section 4: technical writing

This section contains the user manual for the catscript language.

Catscript Guide

This a short overview of the features of the Catscript Language, and includes some syntax documentation.

Introduction

Catscript is a simple scripting language. Here is an example:

```
var x = "foo"
print(x)
```

Features

CatScript has **embedded type inference** during variable assignments, although it may be more clear to specify your data types, you're not *required* to specify your data types.

Here is an example.

```
var x:int = 1;
var y = 1;
var z = "String Example"
```

The variable X has the explicit type definition using the IDENTIFIER:TYPE syntax. Code written with this syntax may be more readable as the type is explicitly defined. The variables y and z have implicit type definition, just like how python does. The type for this variable is inferred based on what type of expression is being assigned to it. So the var y in this example gets the int type, and the var z gets the String type automatically. This approach is easier to write, it can be really nice to have the option to choose which style fits your project needs the best.

Catscript uses a simple **Statically Typed System**, with the following data types

int - a 32 bit integer

string - a java style string

bool - a boolean value

list - an immutible list of values of any catscript type

null - the empty type

Object - any type of value

void - the empty return type

Catscript uses **Static/Lexical Scoping** to manage its variable accessibility. Here is an example

```
var x:int = 10;
function scopeExample():void {
         var x = 5;
         print("first print statement: " + x)
}
function scopeExample2():void {
         print("second print statement: " + x)
}
scopeExample()
scopeExample2()
```

output

```
first print statement: 5
second print statement: 10
```

The first function's print statement finds a variable x=5 within the immidiate scope of the function definition The second function's print statement can't find a variable x in its local scope, so it looks in the global scope to find the variable x with the value 10.

Control Flow

If Else Statements

Catscript implements standard If/Else style control flow. Which accepts any expression that returns a boolean value to allow for conditional execution.

```
var x :int = 2;
var y :int = 3;
if (x == y){
    funcCall(x);
}
else {
    funcCall(y);
}
```

For Loops

Catscript implements for loops to iterate through a set. We did not integrate the age old C syntax, and instead opted for this format, which is easier to write and read.

```
var strList :List = ["Some", "String", "Elements", "in", "a", "List",
"Literal"]
for (str in strList){
```

```
    print(str);
}
```

Output

```
Some
String
Elements
in
a
List
Literal
```

Function Definitions and Calls

Functions are a useful control flow mechanism that is widely used to manage jumps to specific instructions based on their functionality. Functions are implemented in a similar fashion to Java. Here is an example of the syntax

```
function funcExample(arg1:list<string>, arg2:string):bool {
         for (str in arg1) {
                if (str == arg2){
                    return true;
                }
         }
         return false;
}
```

The above example function takes in two parameters, a list of strings and a string, and it has the return type of bool. This example function iterates through the list and looks for the provided string. If it is found, it returns true, if it is not found in the list, it returns false. This is a good example of the control flow in catscript as it displays the functions, looping, and conditional checks available in the catscript language.

Try/Catch Implementation

Catscript allows exception throwing, which can be checked and caught with a try catch block. A try catch block will run the statements in the try part of the block, and if an specific exception is identified by the catch part of the block, the catch statements will be included. Ideally, the 'try' statements will eventually trigger the exception to activate the catch block. Here is a Catscript Example which uses a try catch block

```
var x:int = null;
if (x == null){
    try{
        x = 5;
        var nonNullException:Object = Throw x = 5;
```

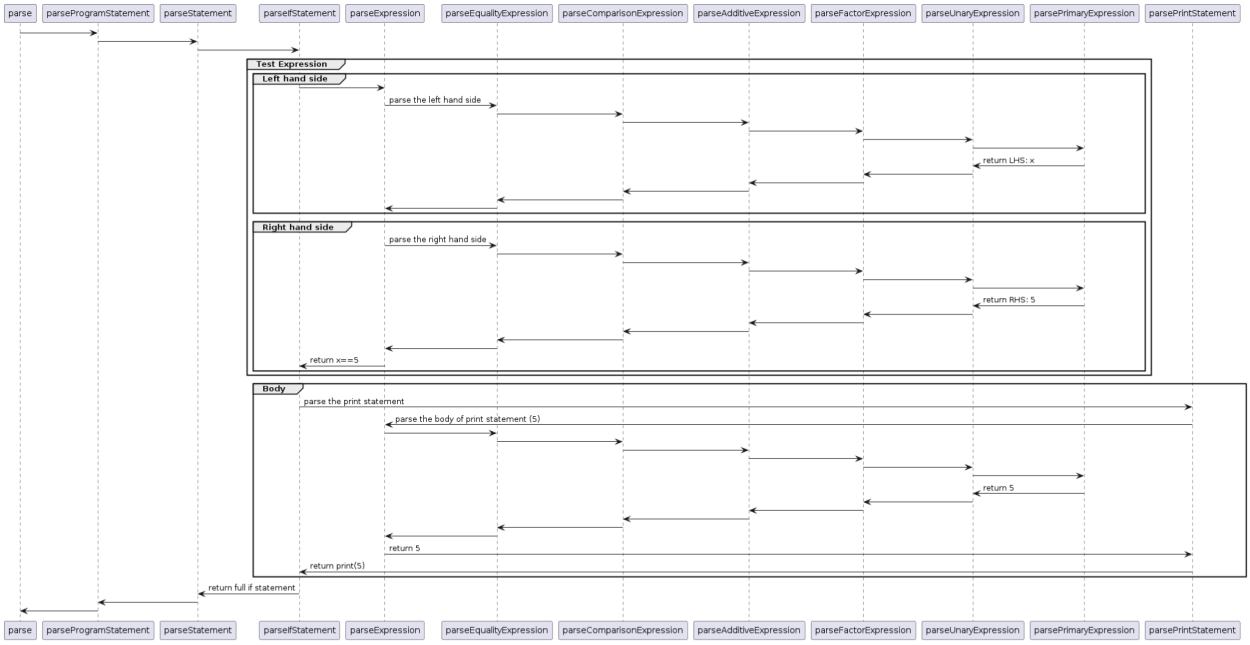
```
    } catch (nonNullException){
        x = 6;
    }
}
```

Section 5: UML

For a UML diagram, it was decided that a sequence diagram would best showcase the project. Here, A sequence diagram representing the parsing of the statement is shown:

```
if (x == 5) {
    print(5)
}
```

The sequence diagram showcases the recursive descent algorithm, where each grammar rule gets its own own function, which recursively calls other functions just like grammar rules recursively move through other grammar rules until they reach a terminal value. In this diagram, each arrow represents a function call, and it can be seen how the algorithm moves up and down the parse tree to parse programs. Both expression and statement parsing are shown. The if statement and print statement are both used.



Section 6: design tradeoffs

At the beginning of the development of the catscript parser, the decision was made to use recursive descent as opposed to the more popular and widely taught use of parser generators. There are many advantages to the recursive descent algorithm. It is a simple, elegant algorithm which mirrors the grammar extremely closely. When writing the algorithm, I found myself constantly looking at the grammar to ensure my code was structured correctly. Essentially, each grammar rule has its own function, and whenever the grammar recursively references a grammar rule from within another grammar rule, the corresponding function will do the same. This allows significant advantages when it comes to debugging.

Firstly, writing the algorithm will make sure any issues with your grammar are uncovered before they become actual bugs, as code that is missing the connecting recursive calls will not ever run. Secondly, the separation of each grammar rule into individual functions fits our software development lifecycle (discussed in section 7) extremely well due to the modular nature of this approach. It is also very easy to add new grammar rules to support new features in the future, as all you need to do is add one new function per rule.

The main disadvantage of parser generators that turned us away from them is the unnecessary complexity they bring. It is true that they have the potential to remove much of the code writing, they require entirely new skills to be learned. Many parser generators require learning a domain specific language for the particular parser generators. Now, instead of focusing on the actual development, developers have to take the time to learn a separate language. Parser generators typically generate messy, complicated, and hard to read code. This makes fixing any bugs or issues in the generated parser much more difficult. Parser generators also can not allow the same level of customization that recursive descent offers, due to the limited control over the generated code. Recursive descent allows complete control over all aspects of the parser.

Our overall philosophy when it came to this decision, essentially, was why learn the extra steps and complexities of parser generators when you could instead learn one algorithm that creates simple, clean, and easy to understand code? We feel this ensured we had a very strong understanding of both the grammar and the parser, overall significantly improving the learning experience.

Section 7: software development lifecycle

The development lifecycle chosen was test driven development. Test driven development involves creating suites of automated tests before writing any actual code. The test driven development cycle follows the ensuing steps:

1. Write a test or a suite of tests, depending on the module/component being tested.
   The initial test writing is a big part of the appeal of test driven development, but it can also be a downside depending on your timeline, budget, and team size. This is because it adds a sometimes very large upfront development cost as you have to create your tests before code, as opposed to directly beginning the actual coding. The advantage of this is that the tests can often reflect the project's requirements and goals, which allows the development team a highly focused development path. This can make the code writing process very fast and efficient, and avoids any unnecessary decision making regarding next steps.
2. Run the tests - expect them to fail, as no code is written (yet).

This initial test run is to have a baseline to compare against, as well as ensure there are no unexpected bugs in your tests. If you are writing a second, third, etc. round of tests, this checks to see if your current code base allows any tests to pass. If written correctly, all should fail with the expected error or failure message. If these are the initial tests, all will obviously fail as no code exists.

3. Write the code necessary to make the tests pass.
   Now, the code writing begins. Here, the goal is to write the minimum amount of code to satisfy your tests. This enforces principles of clean, efficient code, as well as streamlines the development process for maximum efficiency. As a developer, another clear benefit to test driven development becomes clear here. Developers receive very fast feedback, and receive frequent positive feedback as they get more tests to pass. The fast feedback loop is beneficial, as it keeps developers on track as they move through the process of writing and debugging code.
4. Run all tests, verify all tests are passing including old tests previously implemented.
   Now, all tests can be run to ensure all code was implemented properly. It is important to note that here you run all test suites for the entire project, and not just the tests that new code was written for. This is to ensure there are no unexpected side effects of the new code, and all tests that previously passed were not affected by the new changes.
5. Debug/fix any failing tests.
   Here, developers will evaluate the results of the testing. After Identifying any failing tests, they will start the debugging process, repeating the failing tests until they are all passing. Afterwards, it is important to remember to repeat step 4, in order to once again check for unexpected side effects.
6. Repeat!
   After each round of new test suites, revisit step 1. Ensure there are no requirements that are left unmet. If there are, start again with a new round of tests to address the missing requirements, moving through this cycle until all requirements are met.

How test driven development affected the development of the catscript compiler

The Utilization of test driven development was very effective, especially in guiding the development of the compiler. Firstly, test driven development provided clear goals, as well as a clear order in which to accomplish these goals. Secondly, as The goals were completed, test driven development provided a very fast feedback loop. This made sure any problems were addressed early, and did not become larger problems. This sped up debugging massively, as it nearly removed the need to track down the bug entirely. This is due to the fact that test driven development mainly focuses on individual

components, keeping bugs small. Lastly, test driven development provided a simple way to track progress simply by keeping track of what tests are completed and what tests are still needed.

While test driven development provided many benefits, one of its shortcomings made itself apparent during development. With test driven development, It can be very easy to miss some big picture issues, as the tests only focus on the small individual parts. Issues pertaining to interaction between components, integration, or other larger issues can sometimes fall through the gaps in the test suites. One major issue that was missed during development pertained to the parsing of the print statement. The tests checked for printing literals, however, a separate test was failing for an unexpected reason: attempting to print the value of an identifier was not working. Identifiers and print statements both worked separately, however, when combined they produced an error. This gap in the tests cost significant development time, when other testing techniques may have caught this issue sooner.