468 Compilers Capstone

Spring 2022

By: Cody Flynn

Partner: Aidan Scallen

Section 1: Program

Path of source code zip file: csci-468-spring2022-private\capstone\portfolio\source.zip

Section 2: Teamwork

In this project, I worked with my partner to enhance our codebase's usability and flexibility in handling edge cases by generating additional tests that we then traded so that each of us could benefit equally. Along with these tests, my partner provided documentation that helped explain how the catscript programming language is really functioning. Below is an example of one of the tests my partner generated

```
//Here is test number one
@Test //checking if you can have one valid function name followed by an error.
void varFunctionMultConflict() {
    assertEquals(ErrorType.DUPLICATE_NAME, getParseError("var x = 10\n"
            + "var y = 25\n" + "function y(){}"));
}
```

This test is beneficial because we did not have a test to check if a variable has the same name as a function. By including this test in our development process, we can be sure that this error is handled by our parser in a way that is useful for our debugging needs.

Working as a partnership for this testing stage of the development process was advantageous for us because we got to experience another side of software engineering that was unique in our computer science curriculum. Typically, tests are provided for us that we must pass in order to assess the correctness of our programs. However, in this project we needed to think of how best to help our partner find hidden bugs through generating additional tests. I think this side of engineering is often overlooked but is an important field in the industry. Through this project and partnership, we gained valuable experience in quality assurance testing.

Section 3: Design pattern

An example of a design pattern implemented in this project is the memoization pattern. This pattern is used to store frequently accessed data in a cache so that it can be retrieved more efficiently at a later time. The code below is located in our CatscriptType class and functions as a getter method. In this case, our cache is being represented as a hash map that keeps track of which list types have been created already. We first look in the cache to see if the current type is already in the map. If it is, we don't need to add it to the cache a second time, the type retrieved from the map can simply be returned by the function. If the type is not yet in the map, it needs to be added and once again returned.

```
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;
    }
```

This pattern improves efficiency over a more basic implementation because a new ListType object is only created once each time the getListType method is called per type, rather than constructing an object with every call.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

Catscript Documentation

Introduction

Welcome! Catscript is a basic scripting language. Here is a simple example of assigning a variable to a string value and printing that variable.

```
var x = "hello world"
print(x)
```

Features

Variable Statements

In catscript, variables can be assigned to any type expression with the standard single equals sign operator (=)

```
variable_statement = 'var', IDENTIFIER,
     [':', type_expression, ] '=', expression;
```

For Statements

For statements require parantheses surrounding the expression and brackets around the statement to be executed

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
                '{', { statement }, '}';
```

If Statements

If statements require parantheses around the expression, brackets around the statement, followed by an optional else conditional statement

```
if_statement = 'if', '(', expression, ')', '{',
                    { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

Function call Statements

Calls to functions are made with a unique identifier followed by a list of arguments of varying length

```
function_call_statement = IDENTIFIER, '(', [ expression , { ',' , expression } ] , ')'
```

Return Statements

Returning from a function is done using the standard return keyword

```
return_statement = 'return' [, expression];
```

Additive Expressions

Order of operations is maintained in catscript by evaluating factor expressions before additive expressions. Standard operators are used for addition and subtraction

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Factor Expressions

A single forward slash is used for division while a single asterisk is used for multiplication

```
factor_expression = unary_expression { ("/" | "*" ) unary_expression };
```

Unary Expressions

The not keyword represents negation in Catscript

```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```

Comparison Expressions

Standard conventions are followed for the greater than, less than, etc operators

```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression
};
```

Equality Expressions

The boolean expressions use '!=' to represent not equal while '==' represents is equal

```
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
```

Primary Expressions

The list of primary expressions in Catscript

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"|
                     list_literal | function_call | "(", expression, ")"
```

Type Expressions

Types that are available in catscript

```
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression,
'>']
```

List Literal Expressions

Lists require open and close square brackets surrounding the expressions they contain

```
list_literal = '[', expression,  { ',', expression } ']';
```

Section 5: UML.

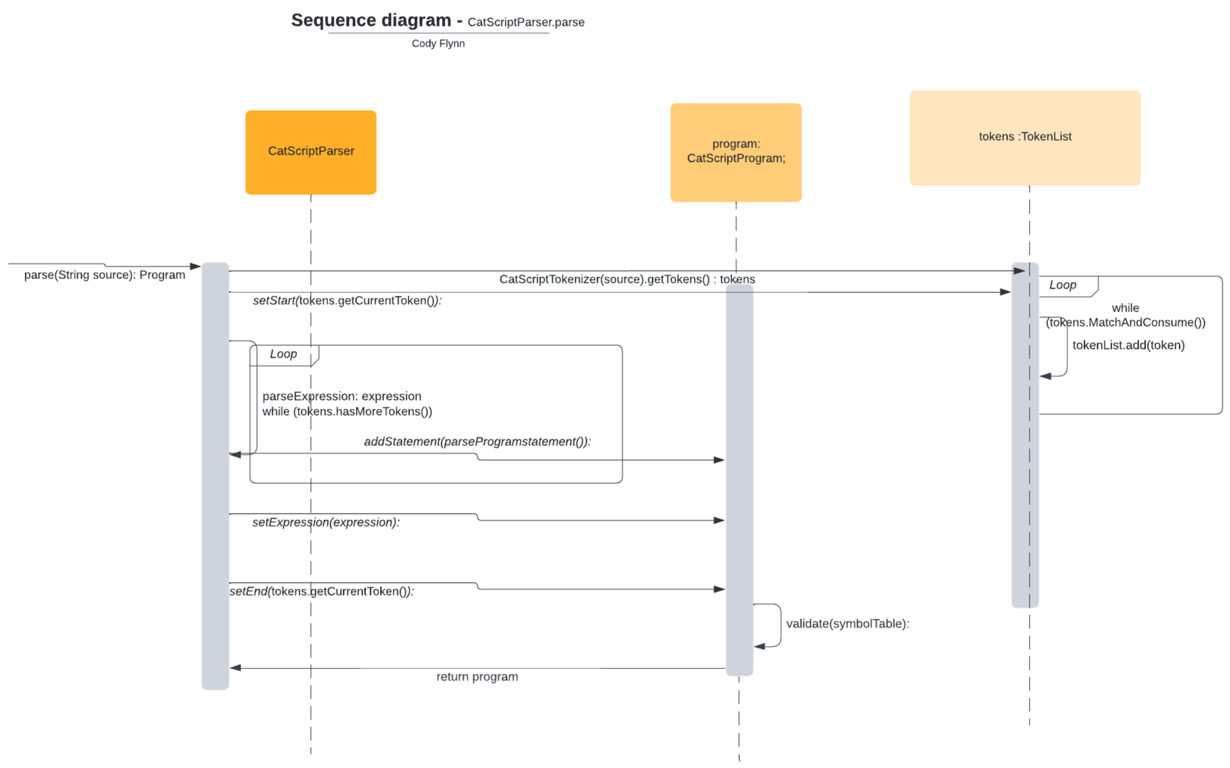
This UML sequence diagram represents a high level view of the parse functionality in our compiler. The code that this diagram is translated from starts with the snippet below.

```
public CatScriptProgram parse(String source) {
        tokens = new CatScriptTokenizer(source).getTokens();

        // first parse an expression
        CatScriptProgram program = new CatScriptProgram();
        program.setStart(tokens.getCurrentToken());
        Expression expression = null;
        try {
            expression = parseExpression();
        } catch(RuntimeException re) {
            // ignore
        }
        if (expression == null || tokens.hasMoreTokens()) {
            tokens.reset();
            while (tokens.hasMoreTokens()) {
                program.addStatement(parseProgramStatement());
            }
        } else {
            program.setExpression(expression);
        }

        program.setEnd(tokens.getCurrentToken());
        return program;
    }
```

Essentially, during the parsing phase, the source string will be split into a list of tokens from the tokenizer. Then, while there are still tokens remaining, we will parse expressions from them in a recursive fashion. If an expression has not been identified by the parser, we will begin looking for statements within the list of tokens until there are no tokens remaining.


Sequence diagram - CatScriptParser.parse
Cody Flynn
CatScriptParser
program: CatScriptProgram;
tokens :TokenList
parse(String source): Program
CatScriptTokenizer(source).getTokens() : tokens
setStart(tokens.getCurrentToken()):
Loop
while (tokens.MatchAndConsume())
tokenList.add(token)
Loop
parseExpression: expression
while (tokens.hasMoreTokens())
addStatement(parseProgramstatement()):
setExpression(expression):
setEnd(tokens.getCurrentToken()):
validate(symbolTable):
return program

Section 6: Design trade-offs

In this project, the main design trade-off was deciding which type of parser to implement. The two main options were a parser generator or a recursive descent parser.

An advantage of parser generators is that they require less coding on the developer's part. The generator creates most of the code based on the grammar it is provided. Therefore, using a parser generator can save time initially. However, a major drawback to parser generators is the readability and debuggability of the code that is created. If you look at the parser code that is generated, it is quite difficult for humans to comprehend. The variables are not intuitively named and the structure of the code is extremely complex and convoluted. Another disadvantage of parser generators is that they are not often used in industry. Parser generators tend to be more of an academic exercise rather than a method applicable to a wide variety of tasks. Finally, building a parser generator does not give a student a good sense of the recursive nature of grammars the in the way that a recursive descent parser does. The generator does not create code in a logically recursive, top-down sequence that a human would. This makes identifying the recursive patterns in the language's grammar much harder to identify and is not as useful for a beginner.

Recursive descent parsers have the advantage of being much easier to understand for students trying to learn how compilers really operate. Because recursive descent parsers are written entirely by hand, the readability of the code is much improved over what a parser generator creates. The recursive descent parser allows for more nuanced and customizable programming languages because they can be tweaked at a more granular level. This is opposed to what a parser generator creates because that code largely must be accepted as is. The recursive descent parser provides a better window into the code while a parser generator creates more of a black box. Additionally, recursive descent parsers are widely used in industry. Most major programming languages including Java implement a recursive descent parser because of the customizability they provide. This makes learning how to create a recursive descent parser much more valuable to a student who could possibly be required to come up with one during their career. The major downside of the recursive descent parser is the amount of code that a developer has to write themselves. To implement a recursive descent parser from scratch, the coder must write not only each function within the parser, but all of the infrastructure code required for the program to run. For someone new or unfamiliar with parsing, this amount of code can be quite intimidating and time-consuming. Depending on the grammar of the languge you are parsing, building a recursive descent parser could take several times longer than using a parser generator.

Weighing these pros and cons between parsing techniques, the recursive descent parser was chosen for this project. While every project has trade-offs in the implementation of design elements, being able to analyze the strengths and weaknesses of each element is an important skill that this project helped develop.

Section 7: Software development life cycle model

The software development life cycle model used for this project was test driven development. This style of software development focuses on writing tests that satisfy the requirements of the project, then writing code to make those tests pass. If bugs are found, additional tests will be developed so that code can be written to fix the bug. These steps can then repeat, forming a cyclical model.

There are several benefits to using test driven development to build a compiler like this. First, the tests provide a clear development path to follow. There is much less ambiguity over whether or not progress is being made when you can observe tests passing that were failing before changes to the code were made. Second, test driven development allows for a more manageable point of entry into a large codebase such as this project. Programmers can often be overwhelmed by the size of a new program, but following the tests and getting them to pass one at a time makes becoming familiar with the codebase much easier. Finally, I believe gaining experience with test driven development will be beneficial to our careers. Many, if not all, large scale projects implement some sort of testing in their development process. However, with most school projects the tests are hidden from the student until their project is submitted and graded by an instructor. Incorporating tests directly into the development process is a more practical and applicable method for real-world programming.

A downside to the test driven development model is the slight lack of freedom that comes with coding specifically to get tests to pass. If the tests were not there, it may become more likely that a student would come up with a unique solution to a problem since they aren't as pigeonholed by individual tests. Another downside to this development model is the narrower sense of understanding of the project the programmer may have. While focusing on one test at a time may be beneficial, it can also be detrimental to one's ability to grasp what is going on in the program as a whole.