

CSCI 468 Compilers Portfolio

Spring 2023

Shreya Deb | Max Kuttner

SECTION 1: PROGRAM

A zip file of the source code is included in this directory named [source.zip](#).

SECTION 2: TEAMWORK

Our team developed separate recursive descent parsers using test driven development for the capstone project. I am the Primary Engineer, and my partner is the Documentation and Testing Engineer. Both partners wrote the technical documentation for the other partner's project. We each wrote tests to assist each other in writing a fully complete parser. My partner's tests are provided in:

[src/test/java/edu/montana/csci/csci468/demo/CapstoneFinalThreeTests.java](#)

SECTION 3: DESIGN PATTERN

The Memoization Pattern is used to memoize calls to `CatScriptType#getListType()`. If we are creating a list we don't want to store the type of each element with the element itself more than we have to, so instead we implement our design pattern. A hash map is used as a cache to store different list variants. This frees up memory and effectively implements the Memoization Pattern. But there is a drawback, in a multithreaded environment this code would be difficult since the `HashMap` is not thread safe for our code is in a singly threaded environment. Here is the code used in our project:

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

SECTION 4: TECHNICAL WRITING

Introduction

Catscript is a simple scripting language built on top of Java. It includes basic mathematical and logical operations, as well as support for iterative statements, conditionals, functions and recursion. Here's an example program that uses a string variable to print hello world.

```
var str = "Hello World!"  
print(str)
```

Comments:

Catscript has single line comments that work similarly to Java or C derived languages. They are initialized with two forward slashes (//) and all text between them and the end of the line is ignored by the compiler.

```
// this is a Catscript comment :)
```

Typesystem

Catscript is a statically typed language which means the variable types are known at compile time and cannot be changed. Catscript variables have flexible type declarations and can be set explicitly or inferred based on their value. The data types within Catscript are broken down into two categories, simple and complex. Additionally, there is a void type that is not assignable to anything, but is associated with functions that do not return a value.

Simple:

Catscript has five primitive types baked into the language, including integers, strings, booleans, objects, and null types. Assignability with primitives is straightforward; any type can be assigned from null and all other type comparisons reference their underlying Java classes.

```
int    // a 32 bit integer  
string // a java-style string  
bool   // a boolean value  
null   // the null type  
object // any type of value
```

Complex:

Catscript contains one native complex type, called a list. Lists are immutable and have the unique ability to carry multiple types. The relationship between simple types is preserved for complex ones, so ensuring list assignability means referencing its component types.

```
list // list<x> is a list of values with type 'x'
```

Features

Print Statement:

Print statements is a method used to visually output data to the console or other output device. It works by taking some input value such as a string or integer and sending it to the terminal. The following example shows how the string “Hello Catscript!” is printed:

```
print("Hello Catscript!")
```

```
> Hello Catscript!
```

Variable Statements:

Variables in Catscript are how values become associated with identifiers. Catscript only allows for one variable to be declared under a specific name at a time. As noted previously, Catscript types can be declared explicitly or inferred.

To demonstrate how variables work in catscript let's examine the following program.

```
var height : int = 3
var width = 2
width = 3
var area = width * height
print(area)
```

```
> 9
```

The height variable is declared explicitly, this is done by following the identifier with a (:) and the appropriate (<type>).

```
var height : int = 3
```

Conversely, the width variable is assigned implicitly. By setting it equal to three, the compiler associates that variable's type with an integer.

```
var width = 2
```

Variables are also able to be reassigned. Notice that width is immediately changed to three. Reassignment is legal so long as the updated value is in agreement with the type system rules.

```
width = 3
```

Variables can also be set to mathematical expressions or non-void functions.

```
var area = width * height
```

For Loops:

Catscript includes a for statement that is used to iterate over lists. For loops function similarly to for-each loops seen in Python and are implemented by saying for “identifier” in “list”. Since Catscript uses lexical scoping, the identifier exists solely inside its loop and doesn't extend beyond its local environment.

The example below describes how to iterate over a list of integers and print out each value.

```
var lst = [1, 2, 3, 4, 5]
for (n in lst) {
    print(n)
}
```

```
> 1 2 3 4 5
```

If Statements:

If statements are a feature useful for controlling the flow of the program. They work by executing a block of code if a boolean expression is met. The if statement has an optional “else” keyword that can direct the program flow only when the if condition is not met. In this example, the output of the program varies depending on the size of the integer variable age. The output will change based on whether the condition is true or false.

```
var age = 14

if (age < 16) {
    print("Too young to drive!")
}
else {
    print("Happy Driving!")
}
```

```
}
```

```
> Too young to drive!
```

Function Definitions:

Functions are a key feature of Catscript that enable a specific block of statements to be used repeatedly. Information can be passed to the function through one or more parameters, which then act as variables inside the function body. A return type can be declared after the parenthesis containing the parameters, using the same convention as variables, with a (:) and the desired return (<type>).

```
function foo(x : bool) {  
  print(x)  
}  
foo(true)
```

```
> true
```

Return Statement:

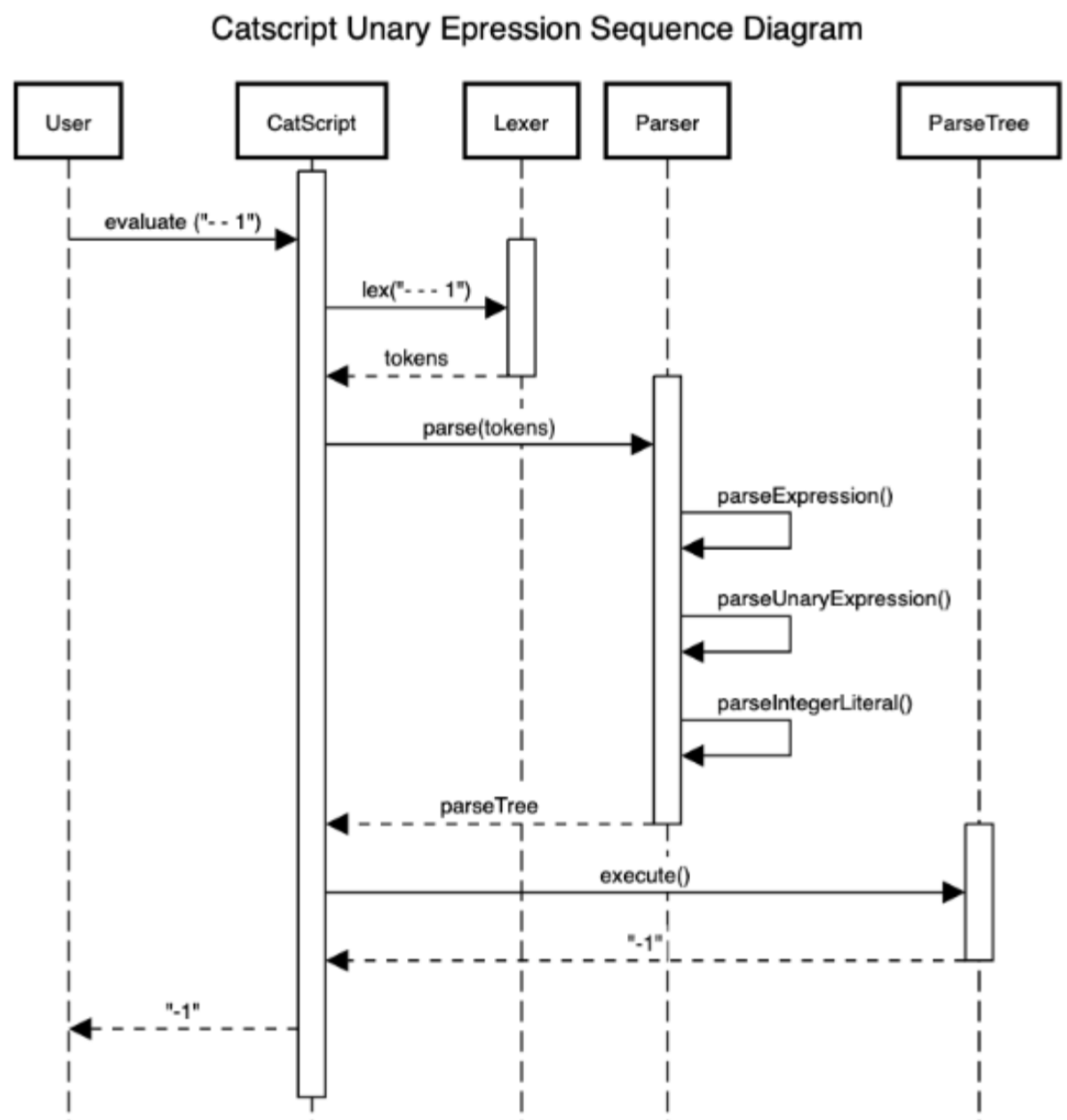
Return statements end execution of a function and optionally returns an expression. Once returned, the control is restored back to where the function was called. Here's an example that compares two integer parameters and returns the largest value:

```
function max(a : int, b : int) : int {  
  if (a < b) {  
    return b  
  }  
  return a  
}  
print(foo(1, 2))
```

```
> 2
```

SECTION 5: Sequence Diagram

The user interacts with the cat script subsystem when he or she submits or calls evaluate on a string (in this case, "- - -1"). The cat script subsystem submits that to the lexer which produces tokens which are then submitted to the parser. This internally expresses exactly what happens in that internal recursive descent algorithm. In this case, we call parse expression and then followed by the parse unary expression, and then parse integers down in depth. When that's done it gets back a parse tree on which execute is called. This then returns the result which is returned to the user.



SECTION 6: Design Trade-Offs

We did not use a visitor pattern for evaluation. We put the evaluation directly on the parse tree. This method can be time consuming as it requires the parser to traverse the entire tree and make decisions based on the data, as well as the bytecode generator to generate code to perform the evaluation. Additionally, since the evaluation is embedded in the parse tree, it can take up a large amount of space, depending on the size of the tree.

Rather than using a parser generator to create our catscript parse, we developed a handwritten recursive parser. Typical methods for creating a parser in compilers courses involve the use of parser generators. All they require is an input grammar, which would have been the focus of most of the course. Specifications are easier to write if the input format is close to normal. This results in a product that can typically be maintained and understood easily. These parsers are also faster than handwritten ones, but not handwritten recursive descent parsers. It should be noted, however, that parser generators can reject grammars from time to time, so it is important to research the specific generator before using it.

There are many advantages to using a handwritten recursive descent parser. As a result, the programmer gains a deeper understanding of all components, the parser runs faster because it starts at the beginning of the program and does not backtrack, and the output creates a parse tree, an extremely useful and fast data structure. In this case, we decided that using a recursive descent style algorithm would be more intuitive than using LAX to generate one, since it would be able to teach us the grammar more in depth. There was a great deal of effort involved in implementing a tokenizer, parser, evaluator, and bytecode generator. As a result of the large number of function calls and recursion involved in this method, it is also less space efficient than other methods.

SECTION 7: SDLC model

We used Test Driven Development (TDD) for this project. TDD is a software development process that starts with writing tests for the desired functionality and then writing code to pass those tests. This allows better quality control and fewer code errors. We test all tests before writing new code, and if a test fails, we write new code to fulfil it.

It was daunting at the beginning of the project to pass so many tests. However, after seeing how the tests were broken down into checkpoints of four, having around 30-70 tests each made our lives easier. For instance, once we passed the tests for the first checkpoint, we felt a sense of accomplishment and gained confidence to continue with the rest of the project. As a result, we were able to isolate a specific piece of code that needed to be implemented properly before moving on to the rest of the project. This made it easier to identify errors or bugs in our code and fix them quickly.