

Catscript Compiler

Silas Almgren

CSCI468

Spring 2023

Montana State University

Program Source

The source code for this project is available here: <https://github.com/Si-Alm/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>.

The specifications used to design the program are available here: <https://github.com/Si-Alm/csci-468-spring2023-private#catscript-grammar>.

Teamwork

There are two components within this capstone document that involved teamwork. The section for technical writing, which documents the scripting language that was implemented for this course, was written by my partner. On the other hand, my partner's documentation for the language was authored by me. Additionally, I contributed three unit tests to my partner's testing suite for the language and vice-versa. The tests contributed by my partner are found in the file **CapstoneTests.java** in the project source.

Design Pattern

One design pattern that was used while implementing this software was memoization. This pattern is utilized to optimize expensive function calls by storing the results of their invocation. These results are stored in a cache-like object where the results of the function call can be looked up by their parameter. This becomes particularly effective when a function is called often with a relatively small amount of values that will be commonly returned. It turns out that the Catscript type system had a function that matched this use case pretty closely.

In this project, memoization was used to optimize the function for looking up the types of list variables. Prior to adding memoization, this would be passed a list type - from which the function would return a new instance of the type of the elements contained in the list. After memoizing this function call, the function will store individual instances of each type returned by the function in a HashMap - which acts as the caching mechanism. In this map, the list types are used as keys and the inner types returned by the function are the stored values. The implementation of this pattern can be found in **src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java** in the method **getListType** on line **38**.

While this function is a good place to highlight a simple memoization example, instantiating the type objects that are returned by this function isn't computationally expensive enough to actually warrant using this pattern here - it's just a touch overkill. However, this function is called often and lends itself to becoming a simple example of memoization so it was a good opportunity to learn and implement a useful design pattern.

Technical Writing

Introduction

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

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

Types

CatScript is a statically typed language with a small type system as follows:

- int – a 32-bit integer
- string – a java-style string
- bool – a boolean value
- list<x> – a list of values with the type ‘x’, if ‘x’ is not specified then the component type is object
- object – any type of value

Type compatibility

There are a few rules that CatScript follows with respect to assignability:

- Anything is assignable to object
- null is assignable to any type
- list is assignable to list<a> if the component type b is assignable to type a

Control Flow

Control flow statements are used to control the flow of execution in a program via decision-making. CatScript contains a few different control flow statements described in the section below.

For loops

In CatScript For loops allow the user to execute a block of statements until the end of a list of objects is reached. The syntax of a for loop in CatScript is ‘for’, ‘(’, IDENTIFIER, ‘in’, expression ‘)’, ‘’, statement ‘,’ ‘;’. For example, here is a valid For loop in CatScript:

```
for (x in [1, 2, 3, 4, 5]) {  
    print(x)  
}
```

If and If Else Statements

In CatScript If statements allow the user to execute a block of statements if a condition is met. The user can also add an Else statement onto the If statement that allows them to execute a separate block of statements if the condition is not met. The syntax of an if statement in CatScript is 'if', '(', expression, ')', ', statement', ' ['else', (if-statement — ', statement', ' For example, here is a valid If Else statement in CatScript:

```
if ( 3 < 4 ) {  
    print("3 is less than four")  
} else {  
    print("3 is not less than four")  
}
```

Functions

Function Declarations

Function Declarations are what allow the user to define a function in CatScript. The syntax of a function declaration statement in 'function', IDENTIFIER, '(', parameter-list, ')' + [':' + type-expression], ', function-body-statement', ';', where parameter-list is defined using the syntax [parameter, ', parameter']; For example, here is a valid function declaration in CatScript: For example, here is a valid If Else statement in CatScript:

```
function addNumbers( a: int, b: int): int {  
    var result: int = a + b  
    return result  
}
```

Parameters

Parameters are the inputs for functions and are defined in function declaration statements. They follow the syntax IDENTIFIER [', type-expression]; to assign an explicit type. For example, here are some valid parameters in a function definition.

```
function cat(M: int, e, o, w: int){ return M + e == o + w }
```

Function Calls

Function call statements are what allow the user to invoke a function. These are also known as function call expressions because they evaluate to a value. The syntax for a function call statement is IDENTIFIER, '(', argument-list, ')' where argument-list is defined using the syntax [expression, ', expression'] For example, here is a valid function call in CatScript using the function defined above:

```
var five = addNumbers(2, 3)
```

Return statements

Return Statements are used within function declarations to end the execution of the function and to return to the previous execution environment right where the function was called. These can also pass back a return value from the function to its calling environment. In CatScript return statements can only appear within the body of function definitions. The syntax for a return statement is 'return' [, expression]; For example, here is a valid return statement within a function definition in CatScript:

```
function equalsTen( ten: int) {  
    return ten == 10  
}
```

Variable Statements

In CatScript, variable statements declare variables that can be used elsewhere in the code and assign them specific values. Types can be explicitly or implicitly declared. The syntax for a variable statement in CatScript is 'var', IDENTIFIER, [':', type-expression,] '=', expression For example, here is a valid variable Statement in CatScript:

```
var helloThere: string = "General Kenobi"
```

Assignment Statements

In CatScript assignment statements are used to give variables new values. The new value must have the same type as the variable otherwise a type exception will be thrown. The syntax for an assignment statement in CatScript is IDENTIFIER, '=', expression For example, here is a valid assignment statement in CatScript:

```
helloThere = "Obi Wan"
```

Print Statements

In CatScript print statements are used to write output to the device's screen. The syntax for a print statment is 'print', '(', expression, ')' For example, here is a valid print statement in CatScript:

```
print("hello world")
```

Primary expression

Primary expressions are the low-level building-block expressions. They help build more complex expressions in CatScript. The syntax for a primary expression is defined as IDENTIFIER — STRING — INTEGER — "true" — "false" — "null" — list-literal — function-call — "(", expression, ")"

Type expression

Type expression are what allows the user to define types of variables, parameters, functions, and the component types of lists in CatScript. The syntax for a type expression in CatScript is 'int' — 'string' — 'bool' — 'object' — 'list' [, ']' , type-expression, ':'

Arithmetic Operators

+ The plus operator is overloaded allowing the user to either add two numbers together or concatenate two strings together. For example, 3+4 and "Hello"+"there" are both valid uses of the plus operator in CatScript.

- The minus operator is overloaded allowing the user to either subtract a number from another number or to flip the sign of a number. For example, 3-4 and -3 are both valid uses of the plus operator in CatScript.

* The multiplication operator allows the user to multiply two numbers together. For example, 3*4 is a valid use of the multiplication operator in CatScript.

/ The division operator allows the user to divide one number by the other number. For example, 3/4 is a valid use of the division operator in CatScript.

Logic operator

not the not operator reverses a Boolean expression. For example, not (3 == 4) is a valid use of the not operator in CatScript.

Comparison operators

== The equals operator allows the user to compare two expressions to see if they are equal. It returns true if the expressions are equal and false if they are not. For example, 3 == 4 is a valid use of the equals operator in CatScript.

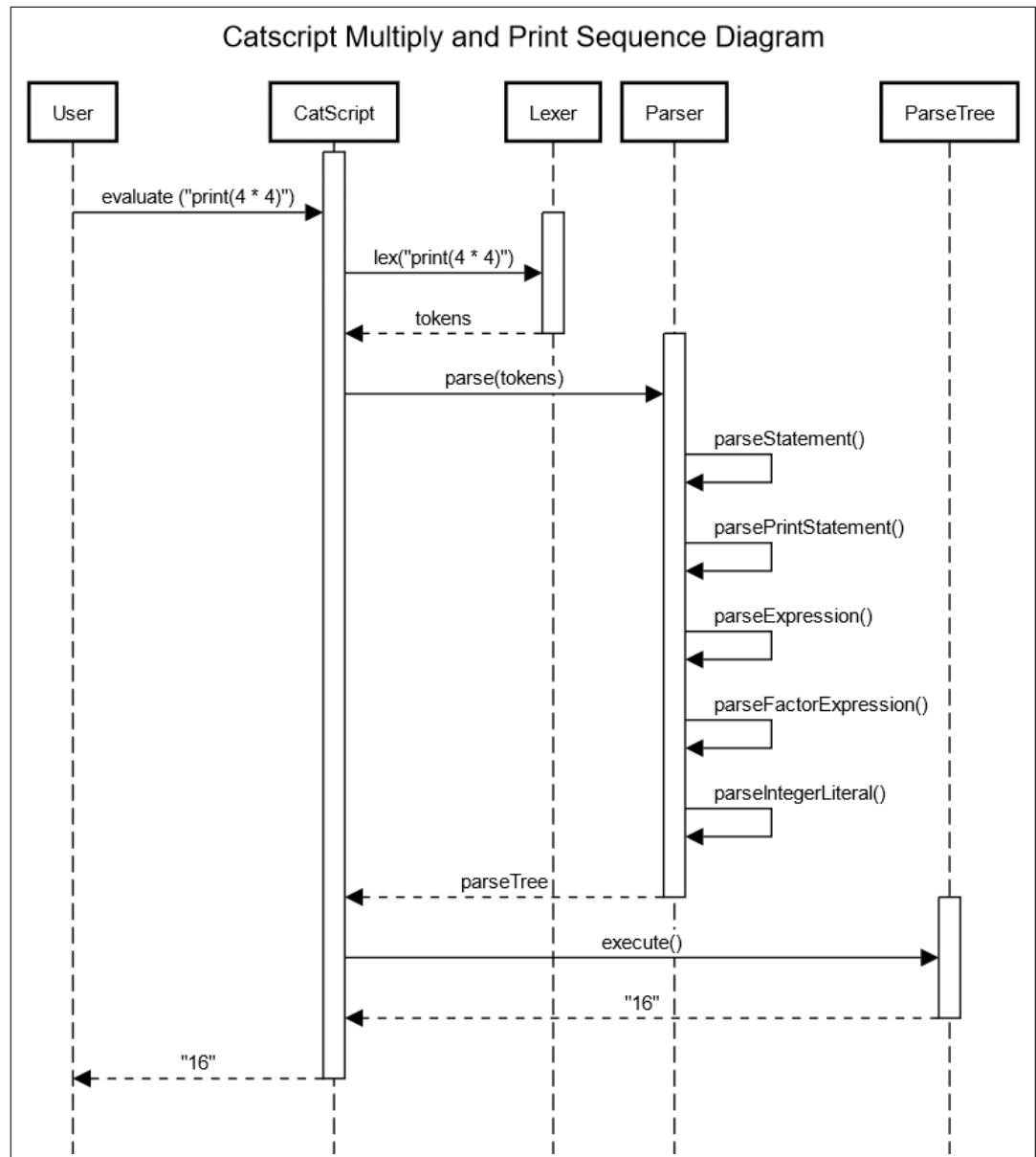
!= The not-equals operator allows the user to compare two expressions to see if they are not equal. It returns true if the expressions are not equal and false if they are. For example, 3 != 4 is a valid use of the not-equals operator in CatScript.

> and >= the greater-than and greater-than-or-equal-to operators are used to see if one number is greater than or greater than or equal to another number. For example, 3 > 4 and 3>=4 are valid uses for these operators in CatScript.

< and <= the less-than and less-than-or-equal-to operators are used to see if one number is less than or less than or equal to another number. For example, 3 < 4 and 3<=4 are valid uses for these operators in CatScript.

UML

Below is a sequence diagram of the Catscript compiler evaluating a simple statement that multiplies two numbers and prints the result. As seen in the diagram, a sequence diagram shows the interaction of components in a system as a piece of data flows through it. In this sequence diagram, one notable feature is the branch for the parser. The descending parse calls that can be seen on this branch clearly highlight the logic implemented in the Recursive-Descent Parser that is discussed in the next section.



Design Trade-Offs

The most prominent design trade-off that was made in the creation of this compiler was to use a Recursive-Descent Parser rather than a parser generator. When designing a programming language, the first step in creating a compiler, a grammar is written that defines the syntax of the language being created.

The logic defined in this grammar is then used to create a parser, which takes a string contained in the grammar (i.e. a program written in the language being implemented) and produces a parse tree. This parse tree effectively contains the instructions that will be compiled and executed.

In practice, such a parser isn't typically written by hand. Rather, tools called parser generators are used to automatically create a parser by providing a grammar written in the generator's specific syntax. However, for this project, we opted to write our parser by hand. In doing this, I was able to understand the grammar for the language on a *much* deeper level. This process required me to learn exactly what each line in the Catscript grammar meant and then translate that into methods that can parse a corresponding piece of code into a branch in a parse tree. In doing this, it forced me to break down the compiler into small, individual components - which rendered a much better picture of the larger, overall system. This deeper understanding of how a program is parsed and stored also made it much easier to write the code for executing and compiling the programs. So ultimately, it may have not been the most efficient or most aligned with best practices, but writing this parser by hand was a worthwhile process.

Software Development Life Cycle Model

The software development model that was used for this project was Test Driven Development (TDD). This is a technique where unit tests are written before development. These tests cumulatively represent the desired functionality of each component in the software. In turn, as the software is developed, updated, and expanded, the tests can be run to verify the functionality of the system as a whole. This life cycle model was chosen for this project because it is a system that requires each component to function *exactly* as needed for the compiler to work as a whole. Even having one small edge case failing can cause unexpected consequences in a complicated piece of software such as this.

At the beginning of the semester, we were provided with a test suite containing around 150 unit tests to verify our compiler was working. These tests had four separate sections to test each step in the compiler: tokenization, parsing, execution/evaluation, and compilation. The code corresponding to each section of tests was written sequentially. Since each component required the output of each previous component, which is highlighted in the provided sequence diagram, this sequential development made the most sense.