

CSCI 486

Spring 2023

Emily Foss

Sam Kynett

Capstone Project

Section 1: Program

For this capstone project we created a small compiler for the mock programming language Catscript. The details of the language and how its functions are laid out in Section 4. We were given a shell of java code and a suite of tests, and over the course of the semester filled out the java code until we had a working compiler that could read and implement Catscript code.

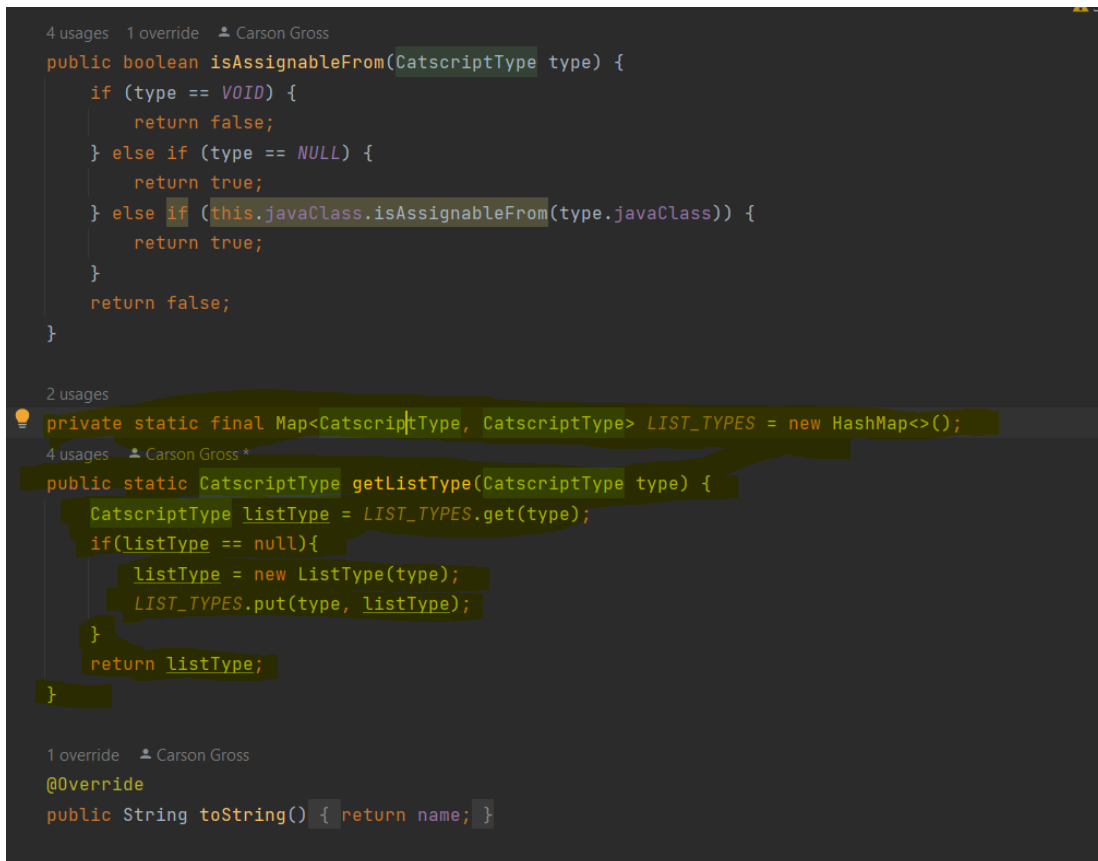
For the code of this finished project, see the source.zip file located in this directory.

Section 2: Teamwork

For this project in which we created a simple compiler, Sam Kynett (team member 1) was the primary engineer, and Emily Foss (team member 2) was the documentation and testing engineer. As the primary engineer, team member 1 wrote and implemented code in order to get the compiler to function in accordance with the rules of Catscript (the mock language that the compiler was written for). As the documentation and testing engineer, team member 2 wrote out the documentation for Catscript, and created tests to ensure that the compiler functioned in accordance with the rules of Catscript. Team member 1's contributions are in the zip file given in Section 1. Team member 2's contributions are the documentation listed out in Section 4, and the three which can be found in `edu/montana/csci/csci468/eval/FossTests.java`

Section 3: Design Pattern

For this project a design pattern that was implemented is memoization. Memoization involves storing the results of a function so that if the function is called again, the compiler can save resources by grabbing the result from memory instead of recreating it. The image below shows the section of our code that utilizes memoization (highlighted in yellow).



```
4 usages 1 override Carson Gross
public boolean isAssignableFrom(CatscriptType type) {
    if (type == VOID) {
        return false;
    } else if (type == NULL) {
        return true;
    } else if (this.javaClass.isAssignableFrom(type.javaClass)) {
        return true;
    }
    return false;
}

2 usages
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();

4 usages Carson Gross *
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;
}

1 override Carson Gross
@Override
public String toString() { return name; }
```

This is our getListType method. It is located at
src\main\java\edu\montana\csci\csci468\parser\CatscriptType.java

The body of the method could just be the line of code

```
return new ListType(type);
```

but in order to save on potentially expensive construction, what we instead do is declare a hashmap that contains types and their respective list types. When a list type is needed, it first checks our hashmap to see if the list type is already there. If it is, it simply grabs it and returns it. If it isn't, then it constructs the new list type, adds it to the hashmap for future retrieval, and then returns the newly created list type. This optimizes our compiler by not requiring the list types to be built every time we need one.

Section 4: Technical Document

Introduction

Catscript is a mock programming language developed for the senior capstone course at Montana State University. It is statically typed, features lexical scoping, supports recursion, variable and return types can be set both explicitly and implicitly, and the plus operator is overloaded for string concatenation to be consistent with most of the popular modern programming languages. An overview and description of Catscript features is to follow.

Type Literals

Catscript supports six data types: int (32 bit), string (Java style string enclosed in double quotes), bool (boolean, true or false), list (list of type x), object (can be any type) and null. These are the keywords used to refer to said data types. As aforementioned, Catscript is statically typed, meaning once a variable is declared with a given type, it cannot be reassigned to a different type at any point throughout the execution of the program.

For Statement

For statements iterate over elements in a given expression and take the general form of the keyword “for” followed by parenthesis, an identifier, the keyword “in” and an expression.

An example for statement:

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

Produces the following output:

```
1 2 3
```

If Statement

If statements allow for control flow in a program to conditionally execute a statement. They take the general form of the “if” keyword, followed by a test expression in parenthesis, followed by a statement wrapped in curly braces and an optional “else” keyword.

An example if statement:

```
var num = 50
if (num < 100) {
    print (“The number is less than 100”)
else{
    print(“The number is greater than 100”)
}
```

Produces the following output:

The number is less than 100

Print Statement

The print statement takes data and sends it to be displayed in the console. The general form is the “print” keyword followed by an expression in parentheses.

An example print statement:

```
print("Example print statement")
```

Produces the following output:

Example print statement

Variable Statement

A variable statement defines a variable for use in a program. In Catscript, the general form is the “var” keyword, followed by the variable name, optionally a colon and the explicit data type (if this is not included Catscript can infer the type implicitly), followed by an “=” and finally an expression. An example variable statement with explicit type:

```
var digit : int = 3342
```

An example variable statement with implicit type:

```
var digit = 3342
```

Assignment Statements

Assignment statements are used to set or update the value of a variable. These statements have the form of the variable name followed by a “=” followed by an expression. The value of the expression is copied into the variable. Since Catscript is statically typed, the data type of the expression must match the data type of the variable if it has been defined previously, else this operation will result in an error.

An example assignment statement:

```
digit = 3343
```

This would cause an “Incompatible Types” error:

```
digit = “This is a number”
```

Functions

Functions are blocks of code that can be used repeatedly and can break code into smaller pieces that execute a particular task. The general form for declaring a function in Catscript begins with the keyword “function” followed by the function name, followed by the list of parameters in parenthesis. There can be 0 or more parameters passed into a function. Next there is an optional function return type preceded by a colon. If this is omitted, the return type defaults to void. Finally the function body is enclosed in curly braces.

An example function definition:

```
function areYouOld ( age : int ) {  
    if (age < 50) {  
        print (“You are young”)    }  
}
```

```
        else{  
            print("You are old")  
        }  
    }
```

```
areYouOld(56)
```

The final line in the above snippet is the function being called with the parameter 56 being passed in. The following output would be produced:

```
You are old
```

Return Statements

Return statements end the execution of a function and return control to the calling function at the point immediately following the call. In Catscript, these are specified by the “return” keyword followed by an expression.

An example return statement:

```
function returnsTrue ( ) : bool {  
    return true  
}
```

Expressions

An expression is something that can be evaluated to a value and they are typically used to assign values to variables and assist with the control flow within a program.

Equality Expression

Equality expressions are used to assert that two quantities equal one another or do not equal one another. These expressions evaluate to boolean literals. In Catscript equality expressions are denoted by either a double equals sign or a bang equals sign between two quantities to assert that the entities are equal and not equal respectively.

An example double equals that would evaluate to true:

$$45 == 45$$

An example bang equals that would evaluate to true:

$$45 != \text{"cat"}$$

Comparison Expression

Comparison operations can perform comparisons between strings or integers based on a condition and returns a boolean literal corresponding to the relation between the two entities.

There are four different comparison operators implemented in Catscript. They are strictly greater than, strictly less than, greater than or equal to, and less than or equal to.

An example strictly greater than expression that would evaluate to true:

$$45 > 40$$

Similarly, the same greater than or equal expression would evaluate to true:

$$45 >= 40$$

This less than or equal expression would evaluate to false:

$$45 \leq 40$$

Additive Expression

Additive expressions handle addition and subtraction arithmetic operations. Intuitively, the operators used are the plus sign and minus sign. The operators are left associative meaning operators with the same precedence are evaluated left to right.

An example addition:

$$45 + 40$$

Printing the result of this addition would output:

$$85$$

An example subtraction:

$$45 - 40$$

Printing the result of this addition would output:

$$5$$

The plus operator is overloaded to perform string concatenation, so the following additive expression:

“The sun “ + “is shining today!”

Would produce the following string:

“The sun is shining today!”

Factor Expression

Factor expressions are used to evaluate multiplication and division expressions of integers. The operators used are the asterisk and the forward slash.

An example multiplication:

$5 * 8$

Printing the result of this multiplication would output:

40

An example division:

$40 / 8$

Printing the result of this addition would output:

5

Expressions can be parenthesised to enforce order of operations and execute more complex mathematical statements such as:

$16 + (8 - 2) * 8 / 2 + 2$

Printing the result of this expression would output:

Unary Expression

Unary expressions consist of one unary operator and one operand. The unary operators available for use in Catscript are the minus sign to make an integer value negative and the logical operator signified with the keyword “not” to flip the value of a boolean literal. A double negative sign or not keywords will cancel each other.

A numerical example:

-42

The above example would evaluate to negative 42.

A logical example:

not true

The above example would evaluate to false.

Another logical example:

not not false

Would also evaluate to false.

List Literal Expression

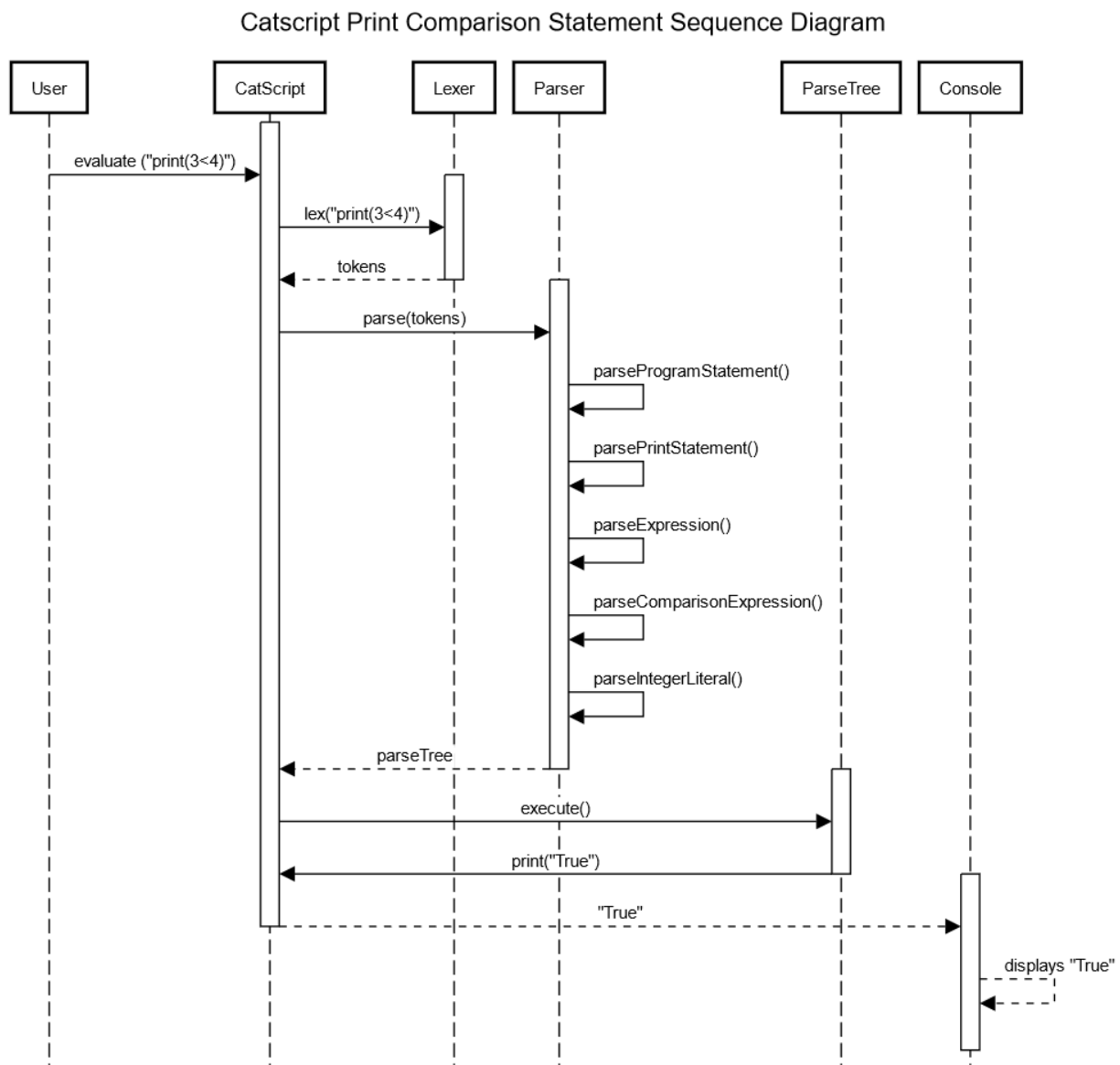
A list literal expression is essentially an array. They are represented by a set of square brackets with expression elements separated by commas. Empty lists are not supported. Lists can contain elements of different types. For example:

[45, true, false, “finger”, 672, “eyeball”]

would be a valid list in Catscript.

Section 5: UML

Below is a sequence diagram of the path for compiling a print statement with a comparison expression



Section 6: Design Trade-Offs

For this capstone project, one of the biggest design trade-offs that was decided on was the decision to write our parser by hand utilizing recursive descent, as opposed to using a parser generator such as Yacc or ANTLR.

In defense of parser generators, they do simplify the process by being able to use an already built library to create a parse for you. All that we would have had to do on our end would be to feed it our lexical grammar, and write a language grammar in some sort of Extended Backus-Naur Form (EBNF).

There were a multitude of reasons as to why we didn't go this route. First and foremost being that it was a better tool for intuitively teaching how grammars function and are structured, and how to implement recursive descent. Writing our own parser also has the added benefit of making the process of debugging code far easier. Generated code is often a mess to have to parse through as there are often variables named in nonsensical ways, and the code can be structured in a way that can be confusing for someone reading through it. Debugging is an easier task when you are intimately familiar with the code that is being debugged, as you're the one who decided what everything is called and how the code was structured. Furthermore, using a library for parse generators means that you are working with an abstracted tool that you have very little control over. If we were to use ANTLR, we wouldn't be able to modify any of the external trees that the ANTLR generator provides. Instead we were able to implement methods such as evaluation onto our parse tree, and this made working with the code more intuitive.

All in all the trade-off that was made to write a parse by hand instead of using a parser generator was the right decision. Using recursive descent made a lot of intuitive sense, we were able to alter any aspect of the code that we wanted, we had an easier time debugging, and we didn't need to sort the messy auto-generated code.

Section 7: Life Cycle Model

For this capstone project, we used Test Driven Development as our software development life cycle model. At the start of the project we were given a suite of tests that determined whether our compiler functioned in accordance to the Catscript language. All of these tests failed when we were given them, and then over the course of the semester we filled in our tokenizer, parser, etc. in order to get more and more tests to pass with the goal being that all of them will pass once the compiler is finished. We were also directed to work on sections of tests in a certain order (first tokenization, then parsing, then evaluation, then bytecode).

This was a successful model to use in our opinion, as the tests told us exactly what features needed to be implemented, and thus focused our efforts. It made it relatively simple to determine how far the project had come along and where further efforts were needed as we could run a block of tests and could tell that the ones that passed required no further work, and the ones that failed would need to be looked into as to why they were failing. Separating the different tests into ordered sections was also useful, as it not only ensured that we weren't overwhelmed by all of the tests at once, but that they were also ordered to better teach the steps of compilation. Tokenization comes well before worrying about bytecode, so not worrying about bytecode until tokenization was complete made intuitive sense and furthered our understanding of the topic.

There were certain aspects of other development life cycles that could have been useful however. For instance Agile's scrum meetings have many uses including ensuring that everyone is on the same page at regular intervals and giving opportunities to take a step back and maintaining consistent work. During this project it felt like progress was being made in large bursts rather than as a continual process, and that aspect could have been improved using a different life cycle model (or at the very least by tweaking aspects of this one). With that minor nitpick aside however, Test Driven Development served us very well as our life cycle model, and gave us clear focus and direction while working on this project.