Kyler Gappa

Carson Gross

Compilers

6 May 2022

Compilers

Program

https://github.com/kylergappa1/csci-468-spring2022-private/blob/master/capstone/portfolio/src.zip

Teamwork

We used a test-driven development process, so the teamwork mostly was comprised of writing tests and documentation for each other. This provided a unique challenge because even though we where both working on creating a similar compiler, the methods that we used varied a bit so the test that where given may have passed for one person but not the other. It was also valuable to get the views of a separate person on what should be valid code in the compiler. River Kelly provided very concise and clear documentation for my compiler so that it is very easy to show to other people.

Design Pattern

In this compiler, we memoized our CatScript Type list so that there was more effective caching. This is a relatively simple pattern that checks to see if there is already an existing List and if there is it adds the type to the list. If there is no existing list, then a list is created and is updated from then on. This technique is often called the flywheel pattern and is useful for reducing the amount of objects that are created in our code.

Technical Writing

Catscript Documentation

Catscript is a simple scripting language that is statically typed and compiled to JVM bytecode.

Features

Here are some features of the Catscript language.

Comments

Comments are used to better help document your code. To write a command in Catscript, the use of two forward slash characters (//) is used.

```
// this is a comment

var a = 1
```

Variable Statement

To declare a variable in Catscript, the keyword var is used.

```
var x = 1
```

Types

Catscript supports explicit typing. To declare a type the use of a colon character (:) followed by the desired type.

```
var x : int = 1
```

Note: If a type is not explicitly declared, then the type is inferred.

Here is a list of the supported types:

- int
- string
- bool
- list
- null
- object

Print Statement

Catscript has a built-in statement to send output to the console. The built-in statement has a reserved keyword print

```
print(10) // will output: 10
```

Math Operation

Catscript supports basic math operations.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division

Example

```
print(2 + 1) // Will print out: 3

print(2 - 1) // Will print out: 1

print(2 * 1) // Will print out: 2

print(2 / 1) // Will print out: 2
```

Comparison

Catscript supports the following comparison operations.

Operator	Name
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Example

```
1 > 0   // true

1 >= 0  // true

1 < 0   // false

1 <= 0  // false
```

Equality

Catscript can check for the equality of two values that are of the same type. These operands include the "equal equal" (==) and "bang equal" (!=).

```
1 == 0 // false

1 != 0 // true
```

Unary Expression

There are two ways to negate a value depending on the type. Variables use the minus symbol (-) and booleans use the not keyword.

Variables

Using the minus symbol character (-) variables will be negated.

```
var x = 5

var y = 10
```

```
print(x + y)  // Will print out: 15

print(x + -y) // Will print out: -5
```

Booleans

Using the keyword not, booleans will be negated.

```
var x = true

print(x)      // Will print out: true

print(not x)  // Will print out: false
```

For Statement

Catscript supports the operation of for-loops. Using the keyword in, a for loop will iterate of each of the items in a list.

```
var list = [1, 2, 3]

for (x in list) {

   print(x)

}
```

If-Else Statement

If statements are similar to those seen in most other common programming langauges. Using the keyword if followed by and expression and body statements. Catscript also support if-else and else operations.

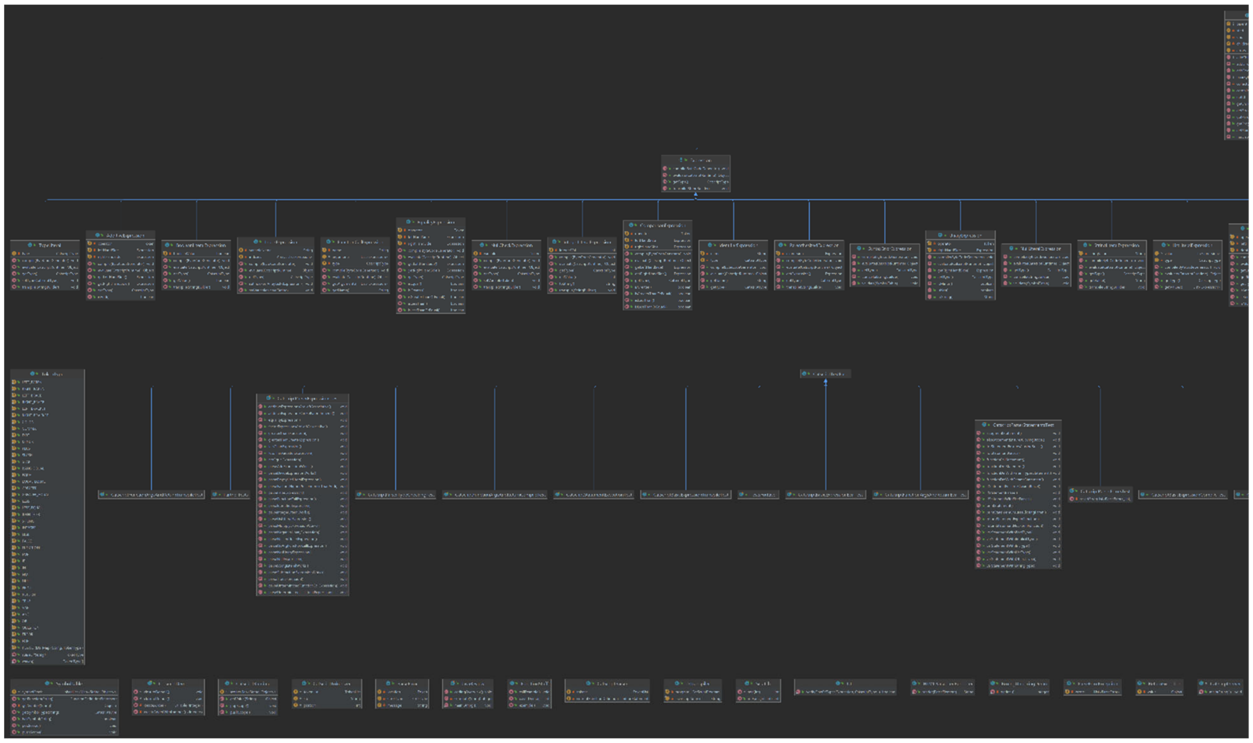
```
if ( expression ) {
   // statements
} else if ( expression ) {
   // statements
} else {
   // statements
}
```

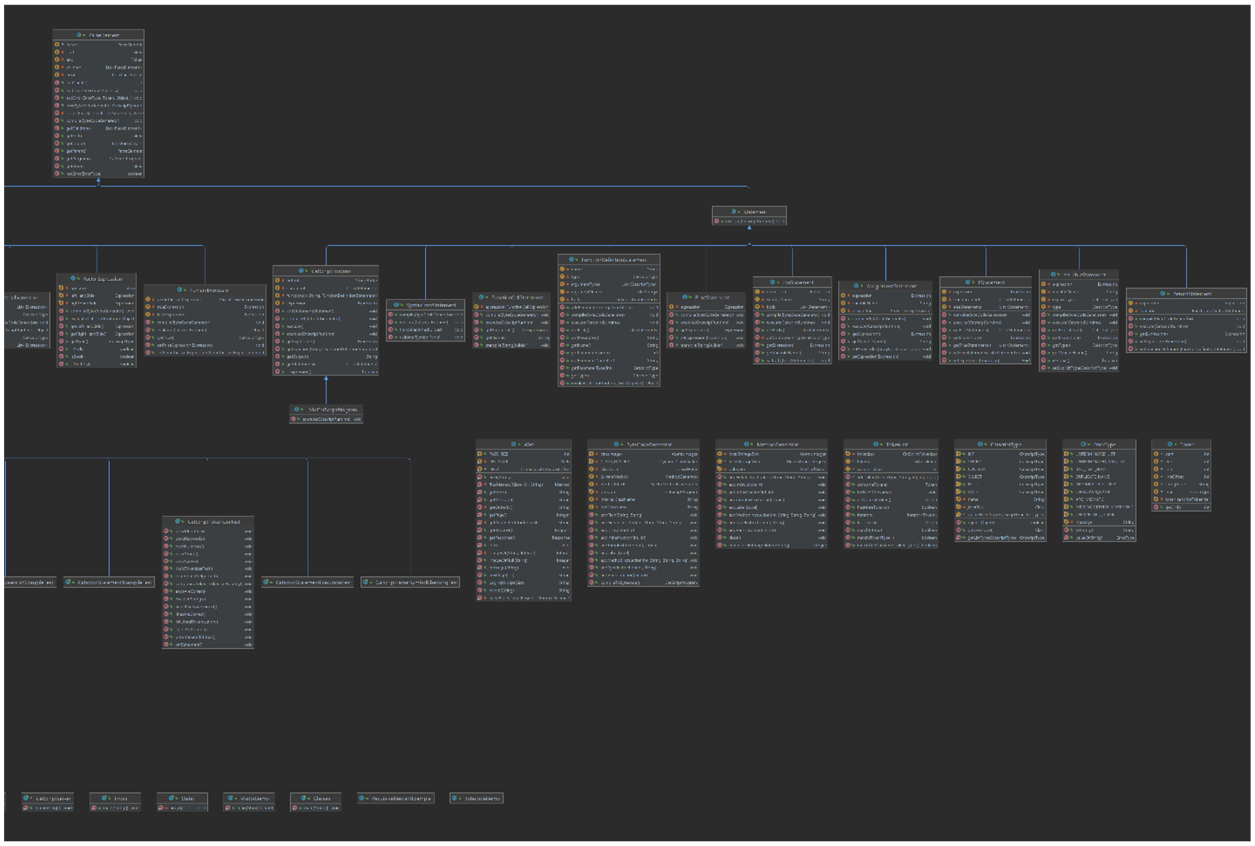
Functions

Functions in Catscript are defined by using the keyword function before the idenitier name.

```
function foo() {
   print("Hello World")
}
foo()
```

UML





I chose not to include all the dependencies as it would create too cluttered of a diagram. Many of these dependencies are also self-explanatory as most of the classes that help tokenize parts of the code must check for expressions and statements.

Design Trade-Offs

For this compiler, we used a recursive decent design. The original choice for this was made by the project requirements. However, this was very useful in seeing as this is a very common development for compilers in production. We could have used a parser generator but there are a few drawbacks to this. This does not give us experience with the tokenizer and the lexical ideas. Using recursive decent also allowed for more direct access to the codebase to help us understand each process better.

Software Development Life Cycle

We used test-driven development for this project. I really enjoyed this method of development as it was a very systematic approach to building out the framework that was provided for us. This method also resulted in very objective goals to structure a workflow around. The immediate feedback from the testing is much more effective than relying on a client to constantly check back in for an approval. This method worked particularly well in this case as we were working with all the end constraints known and immutable. If the requirements had constantly updated, a different method could have been more effective.