Gage Nesbit

05/02/2024
CSCI 468
Capstone

Section 1: Program

There has been a .zip file containing all of the source code inserted into this directory.

Section 2: Teamwork

Our team was a group of three people. One of the main ways that our team collaborated was that we all exchanged Catscript documentation with one another. Team Member 1 created the documentation for Team Member 2, Team Member 2 made the documentation for Team Number 3, and Team Member 3 made the documentation for Team Number 1. During this exchange process, there was a series of iterations to ensure that everyone was producing and receiving high-quality documentation that accurately describes how Catscript functions.

At the beginning of our involvement in the Catscript development, our instructor Carson Gross included many tests that would ensure the implementation of each aspect of the language were correct. Some additional tests were needed to confirm certain specific functions of the language were working correctly. Tests were created by each member of the group and exchanged similarly to how our documentation was exchanged. These tests were specifically designed to be more rigorous than some of the basic tests and were intended to find any possible errors within another member's code base.

Another key aspect of our collaboration which happened throughout much of the development this semester was the discussion of high-level ideas and concepts. It was not uncommon for one group member to clear up any confusion for another group member thus instilling a higher level of understanding with all members of the group. This culminated in all three of us producing high-quality code that performs exactly how the Catscript design intended.

Section 3: Design pattern

```
static HashMap<CatscriptType,ListType> cache=new HashMap<>();
5 usages   Gage Nesbit +1
public static CatscriptType getListType(CatscriptType type) {
    if(cache.containsKey(type)){
        return cache.get(type);
    }
    ListType listType=new ListType(type);
    cache.put(type,listType);
    return cache.get(type);
}
```

The design pattern that was used in my capstone project was the memoization pattern. This can be found within the CatscriptType.java file in the getListType method. The memoization pattern essentially means to use a cache such that the same computations are not done multiple times. This is for the efficiency of CPU and memory. The getListType method previously had taken in a CatscriptType and created a new instance of a ListType object for every call. After the implementation of memoization, the method will check if the CatscriptType has a ListType object associated with it in the cache and return that if it is found. If one is not found then it creates a new instance of the ListType object, then stores that it in the cache for future use. Then it returns the new ListType object.

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

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting langauge. Here is an example:

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

----[OUTPUT]----
foo
```

Type System

CatScript is statically typed, with a small type system as follows, the Object type is able to hold a value of anyother type.

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of value with the type 'x'
- null - the null type
- object - any type of value

Features

For loop statement

Description:

- The user provides a looping expression which is a variable and a list to be iterated over. The list can be defined in the loop expression or assigned to a variable that has been assigned a list previously.
  Inside the For Loop's body, a user can place statements to be executed by the For Loop.
  The For Loop will iterate through the list, execute all statements and evaluate all expressions contained within the loop body for each item
  within the list in the iteration expression.
- This allows for iteratation over the list, making it possible for a range of numbers to
  provided for statements to be executed fixed number of times.
  A list of items the user wants to use in statements can also be selected for processes.

Example:

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

----[OUTPUT]----
1
2
3
```

If statement

Description:

- The user provides any expression that will evaluate to a boolean value (equality or comparison or boolean expression) for the If Statement to evaluate.
- Inside the If Statement, the user will specify statements to be executed by each condition of the If Statement.
- The evaluation to be done first will be to check if the If Statement's expression will yield a true or false boolean value. The outcome of this value controls which portion of the If Statement will be executed.
  If the expression is evaluted to be true, the If Statement will execute any statement that is held within the true body of the If Statement.
- If the expression of the if statement does not return true, the If Statement's body statements will be skipped over.
  Optional:
- The user may provide an Else Statement or a nested If statement within the body of an If or Else.
- Unlike the If, Else Statements only require statements inside the Else Statement body.
  The statements of the Else Statement will be executed if the If Statement's expression returns the boolean value of false.

Example:

```
var x = 7
if (x >= 10) {
    print("Number is greater than 10")
}
else if (x >= 5) {
    print("Number is between 5 and 10")
}
else {
    print("Number is less than 5")
}

----[OUTPUT]----
Number is between 5 and 10
```

Print statement

Description:

- The user provides an expression of any type or kind to be evaluated inside the Print Statement. An important detail is that a print must be provided with an expression, an empty print statment will cause an error to be reported back to the user. If the expression is a variable, the value held by the variable within the symbol table will be written. If it's another type of expression, then that value will be written.
- When the Print Statement is executed, the value of the evaluated expression will be written for the user to read. The value that is written is added to the output buffer of the Catscript runtime. Every print also appends the newline character to the output buffer after the expressions value.

Example:

```
print("Hello, world!")


----[OUTPUT]----
Hello, world!
```

Variable statement

Description:

- The user provides an identifier to be used as the variable's name and an expression to be used as the variable's value. These two values will be associated with eachother in the current scope of the Catscript symbol table. This creates a variable in the memory of the program and the variable name can not be used again within the same scope.
- **Optional:** The user may provide an explicit type by after the identifier giving a colon followed by one of the Catscript types.
  If an explicit type is given, the value of the expression provided by the user must match the explicit type or be assignable to the explicit type.
  If no explicit type is given, the type will be inferred based on the type of the expression value provided. The type of a Catscript variable is static and unable to be changed.

Example:

```
var int count = 25
```

Assignment statement

Description:

- These statements are used to reassign an expressions value to an already initialized variable.
- The statement requires the identifier followed by the assignment operator the "=". This is followed by an expression which serves as a new value to be associated with the identfier in the symbol table which replaces the variable's old value.
- The new value must be of the same type or be assignable to the type of the variable. This is true even if an explicit type was not given to the variable when it was first initialized in the Variable Statement.

Example:

```
int total = 50
total = 68
```

Function definition statement

Description:

- The user provides an identifier which serves as the name for the function in the symbol table followed by parentheses. Interestingly, in Catscript functions are stored in the same namespace as variables.
  Within the bounds of the parentheses, the user may provide zero or more identifiers to serve as parameters to the function.
  These parameters are used as the variable names of the parameters, the value of which will be determined by the Function Call Statement which invokes the funtion after it has been defined.
- **Optional:** The user may give the function an explicit type very similarily to how explicit types are given to variables.
  The explicit type can be given after the parentheses and determines what the type of the return expression will be. Returning a value of a different type or a type that is not assignable to the type of the function will result in an error.
  Each parameter may have an explicit type as well.
  If the user does not provide a specific type, the type will be inferred exactly like a variables type would be.

Example:

```
function sayHello (name : string) {
    print("Hello, " + name + "!")
}
```

Return statement

Description:

- The user can use a return statement to have a function return a value for use outside the bounds of the function. A function is able to contain more than one return statement as a way to break out of function execution. Once the a return statement is executed, the execution of the function is halted and no more statments are executed.
- The Catscript type of the value being returned must match the explicit type of the function if one is given. Any function that is not of type void must contain at least one return statement that is the last statement within body execution. Functions of void type cannot return values. Functions of any other type than void must have a return statment.

Example:

```
function addTwoNumbers (number1 : int, number2 : int) : int {
    sum = number1 + number2
    return sum
}
```

Expressions

Equality expression

Description:

- This exression checks the equality of the values of expressions using the equals (“==”) operator or the not equals (“!=”) operator.
- This expression returns a boolean value of true or false.
- May be used on values of any type.

Example:

```
print((12*4) == (24+24))
print("foo" == "bar")
print(false != true)
print(5 != 5)

----[OUTPUT]----
true
false
true
false
```

Comparison expression

Description:

- Compares two values using the less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=).
- Returns the boolean value of true or false.
- May only be used on expressions of type int.

Example:

```
print(100 < 100)
print(100 <= 100)
print(5 > 3)
print(2 > 3)

----[OUTPUT]----
false
true
true
false
```

Additive expression

Description:

- When using the plus (+) operator, if both values are integers, the operator will perform the addition operation on two expressions of type int.
- If both expressions are of type string, the operator will be used to concatenate the strings together.
- The minus (-) operator only handles subtraction of two integers and will throw an error if used with two strings.
- Mismatching a string and int type with the plus operator will result in error.

Example:

```
print(5 + 3)
print("foo" + "bar")
print(5 - 3)

----[OUTPUT]----
8
foobar
2
```

Factor expression

Description:

- The star (*) operator handles multiplication of two integers.
- The slash (/) operator handles division of two integers.

Example:

```
print(25 * 5)
print(25 / 5)

----[OUTPUT]----
125
5
```

Unary expression

Description:

- Unary expressions handles negative integer values by using the minus (-) operator
  and handles logical negating of boolean values by using the “not” keyword. The resulting value after evaluation will be of either integer or boolean.

Example:

```
print(-5 + 15)
print(not true)

----[OUTPUT]----
10
false
```

Function call expression

Description:

- Function Call Statements are used to invoke functions that have already been defined with a Function Declaration Statement.
- The user must invoke the function with the same number of arguments as parameters that were defined in the function definition statment.
  *If the function is not of type void, the value that is returned can be saved to a variable when the Function Call Statement is used.

Example:

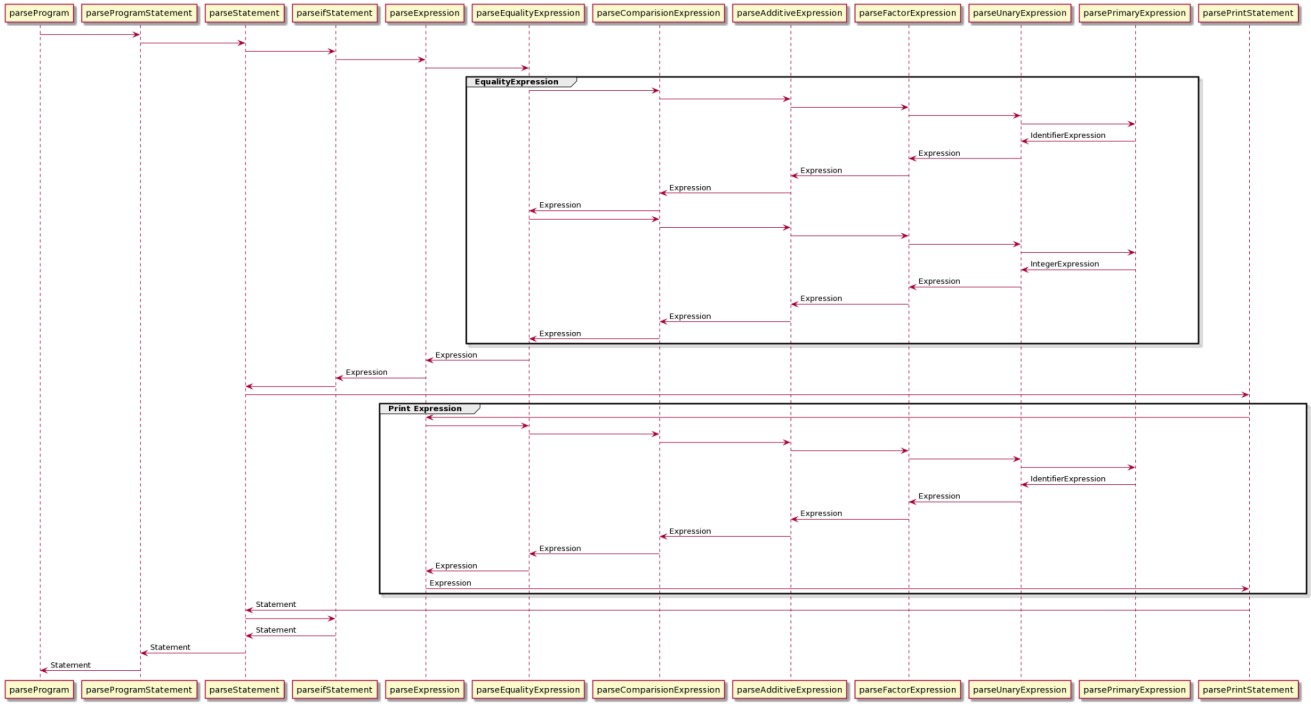
```
int x = 5
int y = 8
sum = addTwoNumbers(x, y)
print(sum)

----[OUTPUT]----
13
```

Section 5: UML.

What is being parsed in this sequence diagram:

```
if(x == 10){
        print(x)
}
```



The above UML sequence diagram is a visual representation of a recursive descent parser. The recursive descent algorithm can be seen by a series of function calls that go deeper and deeper into the grammar of the parser and then return back up the call stack with the program being fully parsed. A series of calls first start the parsing process where the if statement begins to be parsed. During this process, the if keyword and the opening parenthesis are consumed.

The next section takes care of the parsing of all of the if statements children within the parse tree. The first of those is the equality expression that too has children expressions of the x identifier expression and the 10 integer expression. Once the children are parsed the algorithm climbs back up to the equality expression and completes parsing of it. Then more tokens are consumed and it's time to parse statements within the body of the if statement. In this case, there is only one statement. To parse the print statement parseStatement is called and from there, the print statement parsing begins. Its child the x identifier expression is parsed as a primary expression. This is where the final unraveling of the call stack happens. There is no more descending to be done there will be a fully realized parse tree left from the process.

Section 6: Design trade-offs

When designing a programming language many people will choose to use tools that generate lexers and parsers. In the development of Catscript, we chose to create our lexer and parser by hand. This was done for a few reasons. One such reason was that the code generated by these tools is way longer than it needs to be because it's not written in a way that a human would write it. Another reason is that this generated code is not human-readable, variable names and functions and many other things are given complex and vague names not giving a programmer any real sense of how it works. Additionally, for many of the reasons above this makes generated code really difficult to modify yourself and it would take more time to learn how the generated code is working than to write it yourself. You also sacrifice control over how things are lexed and parsed.

The design of Catscript was very intentionally meant to be simple. Behavior was designed to be local to one space. This means eval, and compilation all happen within one file for each type of statement or expression. Some may argue that this could be a drawback as you aren't treating these as separate concerns. I would disagree in that it allows for much more readable code where all behavior for a specific class is within that class.

There were some other design trade-offs made. The design of Catscript's syntax was intentionally made to be similar to Java and JavaScript. This was done to open the door to bytecode compilation and translation. This is because the more similar a language is to another the easier it is to implement translation and compilation.

Section 7: Software development life cycle model

In the Catscript development, we used test first, test-driven development. This allowed us to have our end goal in mind the entire time. With tests being given at the start we just had to focus on the implementation. It also assisted with clarifying exactly what we were supposed to accomplish. One area I can see how this cycle could have harmed our design is that it would make pivoting or modifying the design of the language more difficult as tests would have to be rewritten. However, considering that

we as students were not in charge of the overall architecture of the project that was not an issue. The tests were also structured in such a way that as you implement each one you're setting yourself up well to complete later tests that you otherwise would have been unable to.