Joseph Knappenberger Senior Portfolio - Spring 2024

Joseph Knappenberger

Partner: Victoria White

Section 1: Program

Please see the attached source.zip for my code

Section 2: Teamwork

There was approximately a 95%/5% spilt between myself and partner 2 on my project. I wrote the tokenizer, parser, evaluation, and bytecode generation. Partner 2 wrote documentation and several tests. Working together, we were able to further our understanding of the project and collaboratively learn more than we would have on our own.

Section 3: Design Pattern

I used memoization for the getListType method in CatscriptType. I did this because the method can generate multiple identical listTypes for each type. This duplication is unnecessary, and by using a hashmap this redundancy can be removed.

Section 4: Technical Writing

Catscript Documentation

Provided by Victoria White

Introduction

Catscript is a simple scripting language. Here is what it looks like:

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

Table of Contents

- For Loop Statement
- If Statement
- Print Statement
- Variable Statement
- Assignment Statement
- Function Call Statement
  Function Declaration
- Function Body
- Parameter List
- Return Statement
  Expressions
- Equality Expression
- Comparison Expression
- Additive Expression
- Factor Expression
- Unary Expression
- Primary Expression
- List Literal Expression
- Function Call Expression
- Argument List Expression
- Type Expression

Features

Catscript replicates the fundamentals of most programming languages: statements and expressions. Statements control the flow of the code including the actions the code executes. Expressions evaluate to a single value. Function declaration statements are separate from Statements as they are made up of various other components that are not as streamlined as Statements.

Statements

For Loop Statement

A for loop is declared by using the word “for” and following it with a set of parentheses that contain an identifier (the word “in”) as well as an expression. The expression is anything that reveals a list of values, such as a list literal or a function call that returns a list of some sort. The for loop is executed as a sequence of statements contained within curly braces for each of the items in the list that is created by the expression. It is unique in a sense that an identifier does not need to be declared before. A sample of how to use a for loop is shown below.

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

```
        print(x)
}

//prints 1 2 3
```

If statement

An if statement is declared using the keyword “if” followed by a set of parentheses containing a boolean expression. If the expression ends up being true then the statements contained within the braces that follow the if statement will be executed. Likewise, an else clause could be included if desired that contain statements to be executed if the statement results in a false value.

```
var a = 3
if(a==5){
        print("a is 5")
}else{
        print("a is not 5)
}

// This prints "a is not 5"
```

Print Statement

A print statement is used to write a value to the output. Declared using the word “print” and followed by parentheses containing an expression to be printed. The expression can be a variable, a string, or a value, anything within quotations, or the result of an expression.

```
var x = 3
print(x) //prints 3
```

Variable Statement

A variable statement is used when declaring and assigning variables with an initial value. We declare using the keyword “var” followed by an identifier, which is an optional type annotation, an equal sign, and an expression. The expression can either be the result of an expression, a function call that returns a value, or a literal value.

```
function f(x):int{
        return x*x
}

var a : int = f(3)
var b = true
var c = not(b)

print(a) //9
print(b) //true
print(c) //false
```

Assignment Statement

An assignment statement is used when the value that was previously declared needs to be changed. The syntax consists of an identifier and an equal sign as well as an expression. An example of this is below:

```
var a = 3
print(a) //3
a=6
print(a) //6
```

Function Call Statement

A function call statement is used when invoking a function to execute its code. It consists of a function name, and a set of parentheses containing any arguments passed to the function.

```
var arg1 = 3
var arg2 = true
var result = functionName(arg1,arg2,3)
```

Function Declaration

Function declaration is used when defining a reusable piece of code that can be called multiple times from various areas of the program. The keyword function is first used and then followed with the name of the function as well as a set of parentheses containing any parameters if the function is defined with them. The body is then wrapped in curly braces. This brings us to the next topic: the three kinds of statements used by function declaration statements within Catscript.

```
function functionName(arg1:int,arg2:boolean,arg3):int{
        if(arg2){
                return arg1+arg3
        }else{
                return arg1-arg3
        }
}
```

Function Body

The function body is where the actual logic of the function is defined. It contains only valid Catscript statements such as control structures, function calls, and variable declarations. The function body is defined by the curly braces mentioned above.

```
{
        if(arg2){
                return arg1+arg3
        }else{
                return arg1-arg3
        }
}
```

Parameter List

The parameter list is a comma-separated list that defines the parameters that the function accepts. Each parameter consists of a name and an option type annotation. If there is no type annotation provided then Catscript will try to guess the type from the value that is passed when the function is called.

```
(arg1:int,arg2:boolean,arg3)
```

Return Statement

A function can return a value using return. If the function ends up returning a value, then the return type has to be specific to that of the function declaration. The statement is used to exit a function early at times and also to return a value.

```
return arg1+arg3

- or -

return arg1-arg3
```

Expressions

Equality Expression

Equality expressions are used to compare the two values and determine whether or not they are equivalent. Within Catscript, the double == is used to identify whether two values are equal and the classic != is used to check if the two values are not equal. Equality expressions evaluate to boolean values and are frequently used as the expression in if statements.

```
var a = 10
print(a==7) //false
print(a!=7) //true
```

Comparison Expression

Comparison expressions are used to compare two values and determine whether one is greater, less, or equal to each other. They evaluate to a boolean value and are used in conjunction with if statements (not all the time but more than often). In Catscript, there are various comparison operators as outlined below:

- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to

```
print(3>1) //true
print(3<1) //false
print(2>=2) //true
print(3<=2) //false
```

Additive Expressions

When evaluating arithmetic operations on numeric values, additive expressions are used. In Catscript, the addition operator is (+) which is used to add values whereas the subtraction operator (-) is used to subtract one value from another.

```
var a = 3
var b = 5
print(a+b) //8
print(b-a) //2
```

Factor Expression

A factor expression in Catscript is an expression that involves multiplication or division of two or more operands. The order of operations applies to the factor expression as is any other mathematical expression in the universe. Multiplication/division are performed before addition/subtraction, which makes it important to use parentheses when needed in order to ensure the correct output matches the desired mathematical expression that is attempting to take place.

```
var a = 3
var b = 5
print(a*b) //15
print((2*b)/a) //3 (Division in catscript is integer division)
```

Unary Expression

A unary expression within this language is an expression that only operates within a single operand. The unary operator can either be not or -. The not operator is specific to logical negation on a boolean value. When this is applied to said values, it returns the opposite of the boolean value. The - does this same approach but to numeric values.

```
var a = 3
var b = true
print(-a) //-3
print(not(b)) //false
```

Primary Expression

A primary expression in Catscript is the simplest form of expression and can take on several forms including being an identifier, a literal, a function call, or even a parenthesized expression.

```
//Example of primary expression include
3
```

```
-12
true
null
(4)
"Hello"
func1(3)
a
b
```

List Literal Expression

In Catscript, a list is a collection of ordered values and a list literal is a way to define a list to be a specific set of values. List literals enclosed with square brackets and each element is separated by a comma.

```
var a = [true,false,true]
```

Function Call Expression

A function call expression is used when a function is being called and we are passing arguments to it. It is composed of a function name, followed by parentheses and contains the arguments that are passed to the function. The arguments can be expressions that end up evaluating the expected data types that are specific in the functions parameter list. Because the function call expression is an expression, it evaluates to a value.

```
var a = 7
func1(3,2,a)
```

Argument List Expression

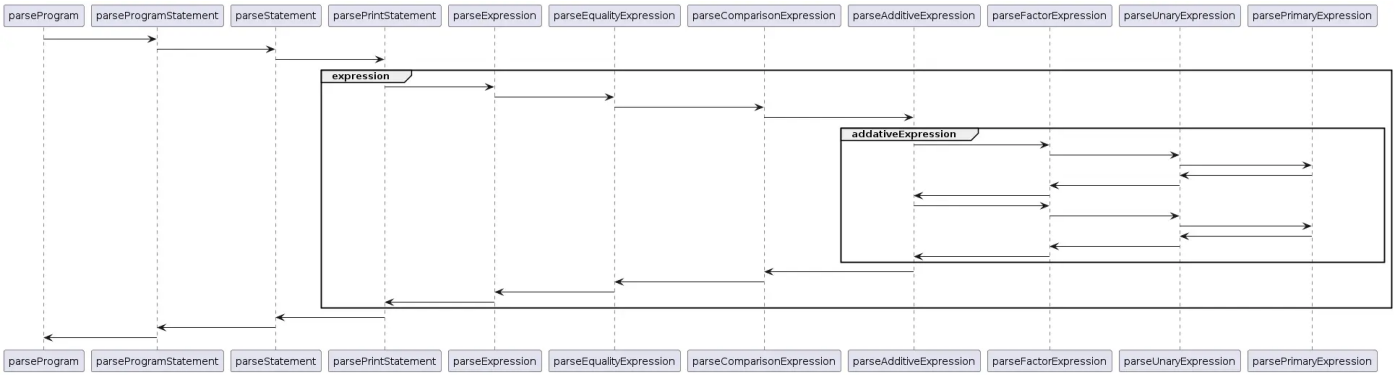
An argument list expression within Catscript is used when passing one or more arguments to a function. There are 0 or more expressions separated by commas and enclosed within parentheses. In the above example, `(3,2,a)` is an argument list expression

Type Expressions

Variables have types that determine the kind of value that they can hold. While CatScript is statistically typed this results in the variable type being known at compile time and not being able to be changed during the program execution. Type expressions have been used various times within the screenshots of my test. They are used in function declarations, variable statements, and parameters lists. The CatScript type system is small and consists of these types:

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

Section 5- UML



The sequence diagram above depicts the parsing of the statement `print(1+1)`. At the top level, you can see the flow of parsing the statement, and from there the expression parsing and additiveExpression parsing sections are further separated. In this diagram, you can see how nested methods allow for the recursive descent parsing that catscript relies on.

Section 6 - Design Trade-Offs

One of the major tradeoffs made in the design of Catscript was the choice not to use a parser generator. Parser generators allow for easier generation of a parser and are often used in compilers classes, but they do not easily allow for some of the more complicated behavior that we implemented in the language, and by building a recursive descent parser ourself, we learned a lot more about how compilers actually work. Given that catscript is a toy language designed for teaching about compilers, the choice to implement a parser from scratch seems to make the most sense to me, and I believe that is why Dr. Gross chose this route.

Section 7: Software Development Lifecycle Model

For this project, we used test driven development. In this process, we had a pre-defined set of tests that we needed to build code to pass. This particular approach worked particularly well for this class, as the tests provided clear, modular, and achievable miniature goals that we could develop towards. Given the complexity of the task, I believe that this was the right approach and that we benefited significantly from having clear goals from the outset.

In the real world, test-first development may not always be the right approach, as requirements and architecture change through the development process. Test driven development is useful though when tests are built to verify and find problems in code once the basics of a project have already been created. Testing is helpful in managing large projects where multiple people need to build code that works reliably together, and it helps avoid errors caused by unintended and unknown side effects that come from developers accidentally changing the behavior of code that other code depends on.