Montana State University Computer Science Department

Senior Team Portfolio

CSCI 468- Compilers, Spring 2024

Garret Streit, Tadeo Aviles Zuniga

Section 1: Program.

The src.zip file, located in the capstone folder under the directory 'capstone/portfolio/src.zip', contains the source code for the project. The code is structured into parsing statements and expressions within a class-based architecture. This organization fosters modularity, readability, and maintainability, enabling clear boundaries and responsibilities for each component of the parser.

Section 2: Teamwork.

For our capstone project, Garret Streit and I focused on developing a parser for Catscript, a language we dedicated ourselves to refining throughout the semester. To ensure the reliability and accuracy of our parser, we implemented a suite of unit tests that served dual purposes—verifying the functionality of our code and ensuring that the interpreter parsed and executed Catscript code accurately. These tests, particularly those crafted by Garret, are located in the src/test/java/edu/montana/csci/csci468/tests/PartnerTest.java directory.

Complexity and Object Identification Test: One of Garret's notable contributions includes the complexityOfObjectListWorks test, which checks the parser's ability to recognize and handle objects within a list. This test is critical for validating the interpreter's type recognition capabilities, ensuring that objects embedded in lists are identified correctly, which is vital for any language that supports complex data structures.

Variable Arguments Handling: Another significant test, argListParameterWorks, ensures that our parser correctly handles functions with variable argument lists, an essential feature for flexible function calls in any modern programming language.

Extended Equality Check: The equalityExpressionExtended test validates the interpreter's ability to assess and confirm equality between variables, a fundamental aspect of logical operations in programming.

These tests collectively helped us pinpoint specific aspects of Catscript that needed refinement and provided a clear path to enhancing its stability and functionality. Each test was crafted to challenge and verify different segments of our parser's capabilities, from basic syntax handling to more complex functional executions such as nested loops, conditional branches, and parameter unpacking.

Documentation and Technical Writing:

In addition to our testing efforts, we also dedicated a significant portion of our project to technical documentation, detailed in section 4 of our portfolio. This documentation not only serves as a comprehensive guide to using and understanding Catscript but also complements our coding efforts by explaining the language's features, syntax, and operational semantics in a structured format. This makes the language accessible to new users and provides a reference that aids in future developments or debugging efforts.

Our collaboration was pivotal in the project's success. By dividing the workload—where I focused on some tests and Garret on others—and then reviewing each other's code, we ensured a well-rounded development process. This cooperative approach extended to our documentation efforts, where we each contributed to different sections, ensuring that the information was not only accurate but also clear and user-friendly.

Section 3: Design pattern.

In my capstone project, I implemented the **Memoization design** pattern to optimize the **getListType** method within the CatscriptType class, found in the directory 'src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java'. This method is critical because it frequently generates ListType objects based on given CatscriptType inputs, which can be computationally expensive.

To enhance efficiency, I introduced a static HashMap called listTypeCache. This cache stores ListType instances where each CatscriptType input serves as a key. When the getListType method is called, it first checks this cache: if an entry exists for the given type, the cached ListType is returned immediately, bypassing the need for new object creation. If no such entry exists, the method constructs a new ListType, stores it in the cache, and then returns this new instance.

This memoization strategy significantly reduces the computational overhead and improves the overall performance of the system. It's particularly beneficial in a parser environment like Catscript, where the same types may be requested multiple times during the parsing of a script. By preventing repeated creation of identical ListType objects, the application runs faster and uses memory more effectively, which is essential in scenarios involving large scripts or multiple simultaneous parsing tasks.

Section 4: Technical writing.

2# Catscript Guide

Introduction

Catscript is a simple scripting langauge similar to Java. Some notable properties of Catscript is that it does not support variable shadowing, has immutable list types, and only supports whole numbers such as integers.

Features

Variable Statement

Used to initalize a variable in catscript. If type is not specified it will be inferred from the right hand side of the equality.

```
var x = "bird"
```

Here a variable named x is initalized to the string of "bird". Since its type is not specified it is inferred as a string.

```
var z:int = 10
```

Here a variable named z is initalized to the integer value of 10. Since the type was specified it must be initlaized to an integer data type. If for example it was assigned to a string this would throw an error.

Print Statement

Used to output content to the terminal in the form of text.

```
var skyscraper:string = "Burj Kalifa"
print(skyscraper)
```

Prints: "Burj Kalifa"

For Loop

Takes a dummy variable to iterate over each element of an object such as a list. Within the body of the for loop any number of statements can be used such as a if statement, print statement, assignment statement, etc. For loops also have their own scope as well.

```
var animals = ["Penguin", "Giraffe", "Tortoise"]
for(x in animals){
  print(x)
}
```

Prints: "Penguin" "Giraffe" "Tortoise"

If Else Statement

If the expression on the if statement is true execute the statements within the if statement. If the expression is false execute the statements within the else statement if it exists. Any number of statements can be included within the if and else.

```
var weather = "bad"

if(weather == "bad"){
    print("Rain Rain Go Away")
}
else{
  print("Not to bad after all")
}
```

Prints: "Rain Rain Go Away"

```
var weather = "bad"

if(weather == "good"){
    print("How Splendid")
}
```

Result: null

if else statments can be nested within each other

```
var weather = "okay"

if(weather == "bad"){
    print("Rain Rain Go Away")
}
else{

  if(weather == "good"){
    print("Amazing")
  }
  else{
    print("Aint bad nor good")
  }
}
```

Prints: "Aint bad nor good"

Function Declaration

Used to create a new function. Functions can optionally take a list of parameters and can contain any number of statements excluding function definitions. the return type of a function can be optionally specified as well.

```
function calculator(num1:int,num2:int){
 var sum = num1 + num2
 return sum
}
```

This function takes two integer arguments and returns their sum.

```
function zoo(){
  return "Giraffe"
}
```

This function takes no arguments and returns "Giraffe"

```
function calculator(num1:int,num2:int):int{
 var sum = num1 + num2
 return sum
}
```

This function takes two integers and is constrained to only returning an integer.

Function Call

Used to execute a function based on its definition.

```
calculator(5,10)
```

Returns 15

```
zoo()
```

Returns "Giraffe"

Assignment Statement

Used to update the value of a variable after its initalization.

```
var count = 5
count = 10
```

Variable count is initlaized to an integer of 5. Later it is assigned to the integer value of 10.

Bad Example:

```
var count:int = 5
count = "Bannana"
```

Here count is initalized to an integer of 5. Later someone has attempted to assign it to the string bananna. Since count is of type int this is illegal.

Objects can Be Assigned to Anything

```
var count:object = 5
count = "Bannana"
```

Here count is initalized to the integer 5. Later it is assigned to the string bananna legally since a variable of type object can be assigned to any other data type.

Return Statement

Used to return a value from a function and kill further execution. Can only exist at the top level of a function.

Valid Example

```
function moo(isCow:bool){

    var sound = "muo"
    if(isCow){
      sound = "moo"
    }
    return sound
    print(sound)
}
```

The final print statement is never reached since the return statement kills the programs execution.

Invalid Example

```
function moo(isCow:bool){

    var sound = "muo"
    if(isCow){
      sound = "moo"
      return sound
    }
    return sound
    print(sound)
}
```

The return within the if statement is illegal since it occurs within another statement and is no longer at the top level of the function body.

Equality Expression

Used to compare whether two or more objects are equal to one another. evaluates to a boolean (true/false).

```
var animal1 = "Turkey"
var animal2 = "Ostrich"

print(animal1 == animal2)
```

Since a "Turkey" is not an "Ostrich" the print statement will print "false".

```
var animal1 = "Turkey"
var animal2 = "Turkey"
var animal3 = "Turkey"

print(animal1 == animal2 == animal3)
```

Prints true since all three animals are the same.

Comparison Expression

Used to check weather two or more objects are greater than (>), greater than or equal to (>=), less than (<), or less than or equal (<=) to on another.

```
var accountX = 1000
var charge = 800

if(accountX > charge){
 print("Sufficent Funds with Surplus")
}

if(accountX >= charge){
 print("Sufficent Funds")
}

if(accountX < charge){
 print("Insufficeient Funds")
}
```

Prints: "Sufficent Funds with Surplus" "Sufficent Funds"

```
 var size1 = 1
 var size2 = 5
 var size3 = 10

 print(sizes1 < size2 < sizes3)
```

Prints: "true"

Additive Expression

Used to add or subtract 2 or more integers and strings together. If an integer is added to a string both will be concatenated together as one string. Attempting to add objects other than an integer or a string will result in an error.

```
print(3+4)
print(7-3)
print(10+20+30)
```

Prints: "7", "4", and "60" respectively

```
print("cre" + 8 + "tive")
```

Prints: "cre8tive"

Bad Example

```
var x = [1,2,3]
print("names" + x)
```

produces an error since a list cannot be added to another data type.

Factor Expression

Used to multiply or divide objects of the type int. Can include two or more numbers in a sequence.

A simple multiplication and division

```
print(5*9)
print(10/2)
```

Prints: "45" and "5" respectively

A complex multiplication and division

```
print(10*10*10)
print(1000/10/10)
```

Prints: "1000" and "10" respectively

Not Operator

used to change a boolean value or an expression that evaluates to a boolean to its opposite.

```
print(true)
print(not true)
```

The first print statement prints "true" and the second one "false".

Data Types

catscript supports strings, booleans, integers, objects and lists.

Creating a String

Strings are placed within " " quotes.

```
var z = "hamster"
```

Creating an int

integers must be whole numbers only. decimal values are not supported.

```
var height = 30
```

Creating a Boolean

booleans take on either the value of true or false.

```
var isGood = true
var isSet = false
```

Creating an Object

Essentially all of the other data types could be considered an object. Creating a variable of type object allows you to assign it to any other data type.

```
var x:object = "hamster"
x = 20
```

Since both a string and int are of type object this assignment is legal.

Creating Lists

Lists can be of type object which allows them to contain any sort of data type (list,int,string,boolean). On the otherhand, if a list is specified as a particular type such as int it can only contain integers and/or a list of ints. If a lists data type isn't specified it is assumed to be of type object. Lists are also immutable in Catscript. Lists can also contain the value null no matter its data type.

Object List

```
var bag:object = [1,2,"dog","cat"]
```

List of Lists

```
var bag:int = [1,2,[3,4,5]]
```

List with Null Values

```
var partFullBag:int = [1,2,null]
```

Bad Example

```
var bag:int = ["dog",1,3,4,5]
```

This is illegal since catscript is expecting a list of only integers but it found a string as well.

List Covariance

If the data type of one list is assignable to the data type of another, so are the lists. Since lists are immutable they remain type safe despite this feature.

```
var x: object = []
var z:int = [1,2,3,4]
x = z
```

Intuitively list z which once was of type int could now have anything inserted into it since it was assigned to a variable representing a list of objects. However, since lists are immutable this is not possible.

Varibale Scoping

A Scope is defined as a section of code in which a variable can be used. After the section of code in which the variable has been defined is executed it no longer exists. Variables can also be used within any scopes lower than the one it is defined in. function defintions, for loops, and if statements each have their own scopes and can be nested in a hierarchy.

Catscript Program with one scope

```
var x = 5
print(x)
```

By default everything that isn't defined within an if, for, or function definition is defined in the global scope (highest level).

Catscript Program with two scopes

```
var x = 5

for(i in [2,4,6]){
  print(x*i)
}
```

In this example we have our global scope where x is defined, and can be used within any lower scope. The for loop has its own scope where the variable i is used. After the for loop executes i no longer exists since it was defined within the for loops scope.

Catscript Program with three scopes

```
var x = 5

for(i in [2,4,6]){
 if(i == 2){
  var j = 4
  print(x*i*j)
 }
 print(x*i)
}
print(x)
```

global scope: var x
for loop scope: var i
if statement scope: var j

Since var j is defined within the if statement it only exists while the if statement is being executed. The variable i exists for the duration of the for loop and the variable x exists for the duration of the catscript program.

Variable Naming

If a variable name is already taken in a higher scope it is illegal to use that name again in the same scope or a lower scope in catscript.

```
var snack = "cheese"

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

This is illegal since the global scope has a variable with the name snack and the for loop scope is also defining a variable with the same name.

```
var x = 5
var x = 20
```

This is illegal as well since the name x is already being used in the global scope.

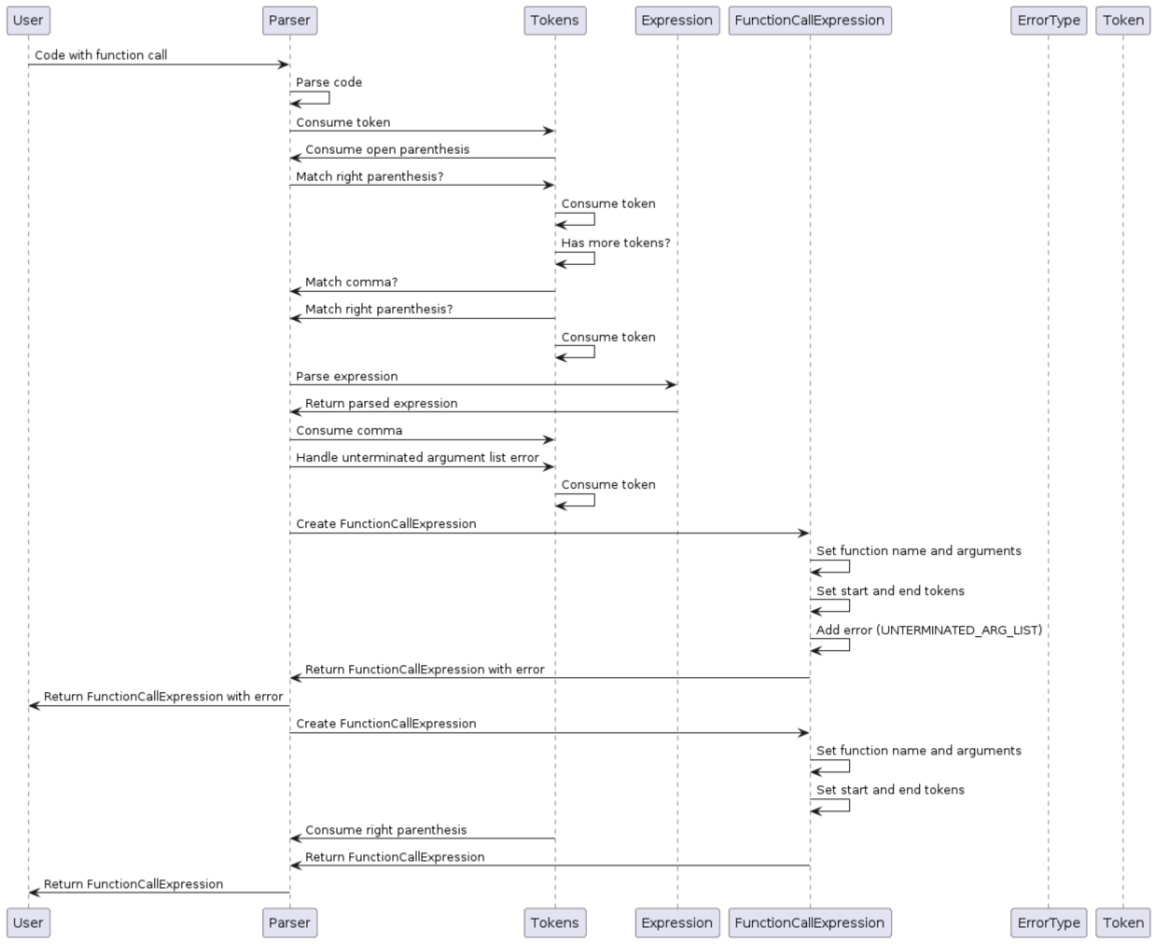
```

for(x in [1,2,3]){
  var snack = "swissCheese"
  print(snack)
}

var snack = "cheese"
```

This is legal. Since the for loop executes first the name snack is no longer in use by the time the line var snack = "cheese" is executed.

Let Your Catscript Journey Begin

I hope you found this guide useful as you embark on your journey through the catscript programming language.

Section 5: UML.



This **UML diagram** served as a guiding blueprint throughout the coding process for the **functionCallExpression.**

Initial Parsing Steps: The parsing process for a functionCallExpression starts when the parser identifies code indicative of a function call, typically marked by a function name followed by an open parenthesis. This initial step is crucial as it sets the context for the parser that a function call is being made, prompting it to begin interpreting subsequent tokens specifically within the framework of function call syntax.

Token Consumption and Validation: As the parser progresses, it consumes tokens sequentially—first the function name, then an open parenthesis. These tokens are essential markers that define the start of a function call. The open parenthesis triggers the parser to shift into a mode where it expects to parse a list of arguments, crucial for correctly building the function call structure in memory.

Parsing Arguments and Handling Errors: Inside the argument list, the parser looks for commas to separate individual arguments, each of which is an expression that needs to be parsed recursively. This recursive parsing allows the function to handle complex expressions as arguments. Error handling is integrated into this step; if the parser encounters a syntax error such as a missing comma or parenthesis, it identifies these as unfinished argument lists. Handling such errors robustly is vital for the stability and reliability of the parsing process, ensuring that the code is syntactically correct or appropriately flagged if not.

Completing the Function Call: Upon successfully parsing all arguments and consuming the closing parenthesis, the parser finalizes the FunctionCallExpression. It compiles the function name and its arguments into a structured object that the rest of the compiler or interpreter can process. This final step marks the end of parsing for that particular function call, transitioning the parser back to a general parsing mode ready to continue with the next segment of code.

This detailed breakdown encapsulates the logical sequence of parsing a function call, highlighting the systematic approach taken by compilers to dissect and interpret each component of the function call syntax. The process not only ensures that the function calls are syntactically correct but also prepares the parsed data for further semantic analysis and execution by the programming environment.

Section 6: Design trade-offs.

In our project to build the Catscript compiler, we embraced the '**Recursive Descent Parser**' design. This approach was chosen over parser-generated code due to its clarity and control. By defining our grammar, we gained explicit control over syntax rules and enhanced our ability to debug and innovate in language design. Parser-generated code, in contrast, would have made interpretation and debugging cumbersome, while limiting our control over language functionality.

Section 7: Software development life cycle model.

In our capstone project, we employed **Test Driven Development** (**TDD**) as our primary development model. TDD involves writing tests before implementing code, ensuring that each feature is thoroughly tested and meets its requirements. This approach provided a structured framework for development, allowing us to focus on implementing specific features while ensuring their functionality through testing. However, one potential drawback of TDD is the risk of passing tests with incorrect code, leading to false positives. Despite this challenge, TDD offered direct control over feature implementation and facilitated a clear understanding of desired functionality. As someone new to TDD, I found this approach enjoyable and effective, as it empowered us to systematically develop and validate our codebase.