Compilers Portfolio

CSCI 468

Spring 2024

By Jacob Tanner & Mike Kadoshnikov

Section 1: Program
The code for this project has been included as a “source.zip” file.

Section 2: Teamwork
Team member 1 (90%):

Team member 1 was responsible for writing code for a compiler for the CatScript language. A grammar was predefined and used by team member 1 for the development of the compiler. Initially, they created a tokenizer to break up the language into chunks of important information. Subsequently, they implemented a parser to generate a parse tree for later evaluation. Then, they wrote code that evaluated each CatScript expression and statement. Finally, they constructed the compiler to convert the language into Java bytecode.

Team member 2 (10%):

Team member 2 was responsible for writing a guide for the CatScript language. This guide includes all expressions and statements available in CatScript and examples of how to use them in the language. They also wrote 3 additional tests located in src/test/java/edu/montana/csci/csci468/demo/PartnerTest.java in the repository.

Section 3: Design Pattern
The memoization pattern was used for this project. Memoization is used to speed up a program by eliminating the repetitive computation of results. Memoization is a form of caching that stores previously calculated information for later use. This allows for faster computation as results can be retrieved from cache after being computed only once.

```
1 usage
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
5 usages   Carson Gross +1
public static CatscriptType getListType(CatscriptType type) {
    return cache.computeIfAbsent(type, ListType::new);
}
```

In this project, we employed memoization for the computation of ListType. To avoid redundant computations of the same ListType object, we instantiate only one and store it in the cache. Consequently, when creating a list with a specific type, the program checks the cache and computes the ListType only if it is absent. To implement this, we utilized a HashMap of CatscriptType to ListType. Additionally, we use the computeIfAbsent method to create a new ListType object if one of the desired CatscriptType does not exist in the HashMap.

Section 4: Technical Writing

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. Although this language is a decent representation of a recursive descent parser project, this language happens to lack the property of being turing complete. Therefore not a lot can be done in terms of array manipulation and data structures severely hindering the practical usability of this language with real world applications.

Here is an example of some simple catscript:

```
var x = "foo"
print(x)
```
Described below are the statements and expression that Catscript supports:

Statements

For Statement

The for statement currently only works with lists in the following format:

```
for (x in [1,2,3]) {
    <for body statements>
}
```
This iterates through the list.

If Statement

The if statement uses a conditional expression, equality or boolean expression in order to branch and execute, here is the following format:

```
if (<expression>) {
    <if body statements>
}
```

Else Statement

The else can be used with the if statement to execute the false branch:

```
if (<expression>) {
    <if body statements>
} else {
    <else body statements>
}
```

Print Statement

The print statement can be used with the following format to print items to the screen:

```
print(true)
print("Hello world")
```

Variable Statement

The variable statement initializes a variable and sets a value to it.

```
var x = 10
var y:bool = true
var list1:list<int> = [1,2,3,4,5]
var list2:list = ["Hello", "No", "Yes"]
```

Assignment Statement

The assignment statement is used to change the value of an existing variable statement.

```
var x = 10
var y:bool = true

x = 1
y = false
```

Function Definition Statement

This is used to define a function definition, function also support return types

```
//Function without return type (i.e. void)
function foo() {
    print("This is a function")
```

```
}

//Function with a return type
function foo1(): string {
    return "This is a function"
}
```

Function Call Statement

This is used to call a function previously defined

```
//Function without return type (i.e. void)
function foo() {
    print("This is a function")
}

//Function with a return type
function foo1(): string {
    return "This is a function"
}

// Here are the function call statements
foo()
print(foo1())
```

Return Statement

Return statements are used to return values in functions

```
//Function without return type (i.e. void)
function foo() {
    print("This is a function")
}

//Function with a return type string
function foo1(): string {
    return "This is a function"
}
```

Expressions

Equality Expression

This expression evaluates equality operators

```
10 != 5     // should evaluate to false
10 == 10    // should evaluate to true
```

Comparison Expression

This expression evaluates comparison operators

```
10 > 5      // should evaluate to true
10 <= 10    // should evalutate to true
5 >= 5      // should evaluate to true
5 > 8       // should evaluate to false
```

Additive Expression

This should handle addition with integers and concatenating other types if not addition, this also handles subtractions for integers (nothing special for strings)

```
"5" + null      // evaluates to "5null"
1 + 2           // evaluates to 3
"Hi" + " John"  // evaluates to "Hi John"
```

Factor Expression

This should handle multiplication and division

```
5 * 5       // evaluates to 25
5 / 5       // evaluates to 1
```

Unary Expression

This handles not and negative

```
-5          // evaluates the value 5 to be negative
not true    // evaluates to false
```

Primary Expression

These are the primary expressions allowed in the language

```
x
"Hello"
5
true
false
null
[1,2,3]
foo()
(1*3)
```

Type Expression

Catscript supports these types

```
int
string
bool
object
list
list<type_expression>
```

Features (Usages)

Variables

```
var variable1 = 10
print(variable1)
```
Output : `10`

Function Defintion and Function Calls

```
function foo(x) {
    print(x)
}
```

```
foo("Test String")
foo(10)
```
Output : `Test String 10`

Not only do function have the ability to perform computation we can also return values with a function:

In order to specify the return type we need to declare it as such: `function foo(input: int): int {}`

Here is an example of use:

```
function foo(input: int): int {
    return 2*input
}


print(foo(5))
```
Output : `10`

This example returns the doubled value of the integer provided not only can we return integers, other types we can return are strings, lists, booleans, null, and objects

Here are some examples:

Object return:

```
function foo(input: int): object {
    return 2*input
}


print(foo(5))
```
Output : `10`

This function still returns 10 because everything should be extendable from object

Lists return:

```
function foo(listofints : list<int>): list<int> {
    for(x in listofints) {
        print(x)
    }
    return listofints
}
```

```
print(foo([1,2,3]))
```
Output: `1 2 3 [1, 2, 3]`

In this example we specify the input list with `list<int>` and to specify the return value after the function as such `function foo(...): list<int>` lists can be made from all types in Catscript such as object, string, int, and boolean lists.

For Loops

```
for(x in [1,2,3,4,5]) {
    print(x)
}
```
Output : `1 2 3 4 5`

Although CatScript doesn't support a generic for statement that you see in Python, Java, C++, etc. you are able to perform the same function with a recursive function:

```
function foo(initialVal : int, limitVal:int) {
    if (initialVal < limitVal) {
        print(x)
        foo(initialVal+1, limitVal)
    }
}

foo(0, 5)
```
Output : `0 1 2 3 4`

If Statements

```
var variable1 = 10

if (variable1 > 5) {
    print("The variable is greater than 5!)
}
```
Output : `The variable is greater than 5!`

If-Else Statements

```
var variable1 = 2
if (variable1 > 5) {
    print("The variable is greater than 5!")
```

```
} else {
    print("The variable is less than 5!")
}
```
Output : `The variable is less than 5!`

Recursive Functions

```
function foo(x : int) {
    print(x)
    if (x > 0) {
        foo(x-1)
    }
}

foo(9)
```
Output : `9 8 7 6 5 4 3 2 1`

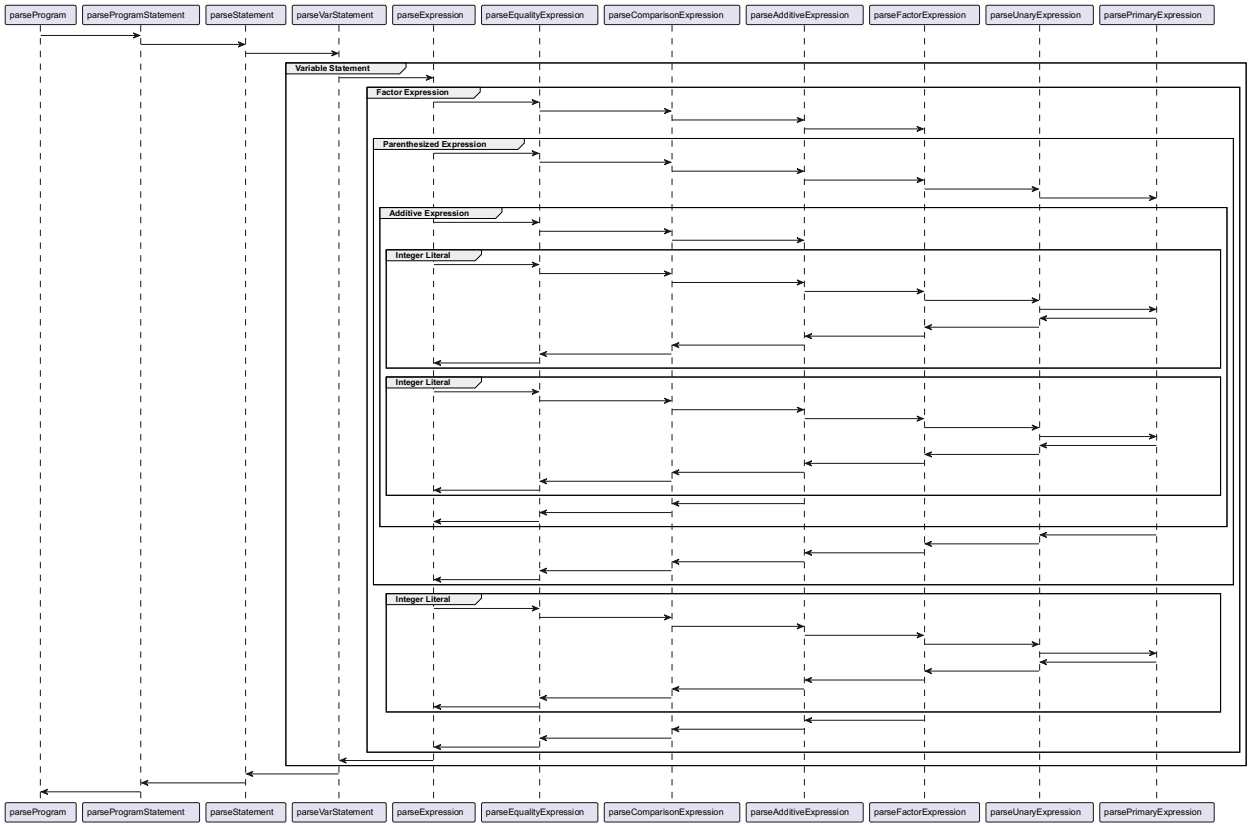
Lists

The list system is read only as it simplifies the generics, so we are unable to add to a list but we can do things like this to lists:

```
function foo(tempList : list<object>): list<object> {
    return [tempList, tempList]
}

print(foo([1,2]))
```
Output : `[[1, 2], [1, 2]]`

Section 5: UML



The UML diagram above shows the parsing of “var x = (2+2)*5”. It demonstrates the recursive nature of the CatScript parser, stemming from the natural recursive structure of grammars. The generated parse tree of “var x = (2+2)*5” is as follows:

- Variable Statement
    - Factor Expression
        - Parenthesized Expression
            - Additive Expression
                - 2 (Integer Literal)
                - 2 (Integer Literal)
        - 5 (Integer Literal)

When parsing a variable statement the tokens “var”, IDENTIFIER, and “=” are required. In this instance, “var”, “x”, and “=” are provided. Subsequently, the right side of the “=” is evaluated with parseExpression() to find the value “x” should be initialized with.

As we descend though the expression evaluation methods, we first encounter is a factor expression. For a factor expression, both right and left side expressions are evaluated with parseExpression(). On the left side, we find a parenthesized expression, and its contents are again evaluated using the same parseExpression() method. The parentheses contain an additive expression which has a right and left side that both must be evaluated again using the same method. Both sides are parsed as integer literals with the value 2. The additive

expression evaluates to 4, which means the parenthesized expression evaluates to 4, and then the left side of the factor expression evaluates to 4. Now, the right side must be evaluated using the parseExpression() method. The right side is just an integer literal with the value 5. With both sides evaluated, the factor expression can be evaluated to 20. Therefore, the variable "x" is initialized with the value 20.

Section 6: Design trade-offs

For this project we decided to use recursive descent for our parser. This is because it teaches the recursive nature of grammars in a very clear way. This avoids much complexity in our parser. It also widely used in industry because of the simplicity and the control you get with a handwritten recursive descent parser. The alternative is a parser generator which is more complex and confusing than recursive descent.

Parser generators are programs that take a language specification and generate a parser for that specification. The input is typically a lexical grammar defined in a regular expression or a language grammar defined in EBNF. The main benefit of parser generators is that they generally require you to write less code than doing it all by hand. They are praised in academia because they are complex, and many papers can be written about them. However, they often have obscure syntax and are not widely used in industry.

Recursive descent parsers, on the other hand, are top-down parsers build from a set of recursive methods where each method represents a nonterminal of the grammar. They are handwritten and not generated using grammar defined in regular expressions or EBNF. This means that you must very carefully think about the details of the grammar. They are a lot simpler and allow you to see the recursive nature of grammars. It also allows for finer control and better error messages.

Section 7: Software Development Life Cycle Model

For this project, we utilized Test Driven Development (TDD), a software development approach where tests are written before code, and then code is implemented to pass these tests. TDD ensures comprehensive test coverage, as all new code must be covered by at least one test. This leads to a robust software and ensures that changes to code do not inadvertently break existing functionality.

Tests were organized into several groups to ensure the project remained on track. These groups included tests for the tokenizer, parser, expression and statement evaluation, and bytecode generation. Each set of tests represented an important checkpoint in our development process. The tokenizer was the first checkpoint, responsible for tokenizing the input. The next checkpoint involved the parser, which generated a parse tree based on the grammar and tokens from the tokenizer. The third checkpoint was expression and statement evaluation. The last checkpoint was bytecode generation, which compiles CatScript into Java bytecode.

By employing TDD, we ensured that each part of the project was complete before proceeding. This iterative approach is particularly crucial for projects that evolve incrementally. The tests served to validate code quality and provide assurance that any modifications would not cause issues with existing functionality.