Capstone Documentation

Josh Aney, Ryan Day

~ CSCI 468: Compilers ~

Section 1: Program

Please see the zipped source code & specification in source.zip

Section 2: Teamwork

For this project, it was split up into two main parts. The implementer (Team Member 1) and the tester /documentor (Team Member 2).

Team member 1 was responsible for implementing the recursive decent parser for the Catscript language. They implemented, tokenization, parsing, semantic analysis, evaluation, and bytecode generation. All of these elements were created through test-driven development.

Team member 2 was responsible for testing and documenting the compiler for Catscript. They created the test for each element of the compiler, and they also wrote documentation for each part of the compiler. The tests that Team Member 2 created are in `csci468.partner.PartnerTests`. The documentation that Team Member 2 contributed is in the documentation section of this document.

For Team member 1, the percentage of time spent on this project for implementation was around 80%, whereas for Team member 2, the documentation took around 20% of the time.

Section 3: Design Pattern

In this project, the Memoization Pattern in `csci468.parser.CatscriptType.getListType()`. This is done so that the listType field caches the result, so new ListType(this) is only called once for each CatscriptType instance.

This is important because in validation, types are compared using ==. This checks whether the two references point to the same object in memory instead of just checking if the objects are logically equal. If `csci468.parser.CatscriptType.getListType()` created a new ListType every time it was called, then the Catscript.INT.getListType() would produce different objects each time. This would cause == to return false.

Since the memoization pattern was used in this project, the `csci468.parser.CatscriptType.getListType()` method only created one object and then stores it. This guarantees that comparisons with == work properly.

On top of all this, momoization reduces unnecessary object creation. Even though object creation is cheap, in a recursive decent parser it can happen a lot if memoization is not done.

Section 4: Technical Writing/Catscript Guide

CatScript Language Documentation

CatScript is a small statically typed programming language that supports basic program language function such as variables, arithmetic, conditionals, loops, functions, and lists. It has a simple syntax which makes it easy to learn as well as easy to learn the core concepts of compilers. It still provides many of the core features of a modern programming language. It is also a turing complete language.

Catscript Types

CatScript includes the following variable types:

- `int` – Integer values (1, 2, 3)
- `string` – Text values ("Hello Wolrd")
- `bool` – Boolean values (`true` / `false`)
- `list` – Lists can be of any type
- `object` – Generic type representing any value
- `null` – Represents no value

Variables

Variables are declared with the `var` keyword. Type specifications are optional but may be included. If a type specification is not included, the var will still be assigned to the type of the expression.

```
var age = 21
var name : string = "Alice"
var scores : list<int> = [90, 85, 100]
```

Expressions

CatScript supports the standard set of expressions. There are not a ton of complex expressions, but the core ones are included.

Arithmetic

```
var sum = 3 + 4        // 7
var concat = "hello" + 3 // "hello3"
var concat2 = "hello " + "world" // "hello world"
var diff = 10 - 3      // 7
var product = 6 * 7    // 42
var quotient = 20 / 4  // 5
var remainder = 10 % 3 // 1
```

All Arithmetic works as expected in catscript. For a sum, if either side of the expression is a string, the other side will also be treated as a string. All the other arithmetic expressions require integers on both sides, otherwise it will cause an incompatible types error.

Unary Operators

```
var neg = -5
var opposite = not true   // false
```

Unary operators work as expected and are similar to other programming languages.

Comparison

```
1 < 2      // true
5 >= 5     // true
3 > 10     // false
4 <= 3     // false
```

Comparison expressions only work with integer types. Any other types will cause an incompatible types error.

Equality

```
1 == 1     // true
"cat" != "dog"  // true
```

Equality expressions work with any types. Equality expressions always check for logical equality and not just reference equality.

Index Expressions

Access elements in a list using bracket notation:

```
var first = myList[0]
var firstFirst = myList[0][0]
```

List indexing works similar to JavaScript. Catscript is also zero based.

Grouped Expressions

Parentheses can be used to control evaluation order:

```
var result = (2 + 3) * 4   // 20
```

Print Statements

Use `print()` to output values to the console

```
print("Hello, world!")
print(42)
print(true)
```

Control Flow

If / Else Statements

Conditional branching is handled with `if`, `else if`, and `else`.

```
if (x > 0) {
    print("Positive")
```

```
}
else if (x < 0) {
    print("Negative")
}
else {
    print("Zero")
}
```

While Statements

Repeat a block while a condition remains true.

```
var i = 0

while (i < 5) {
    print(i)
    i = i + 1
}
```

For Statements

Iterate over elements in a list using `for ... in`. Catscript does not have an iterative for loop like java it only allows for a for each loop.

```
var names = ["Alice", "Bob", "Charlie"]

for (name in names) {
    print(name)
}
```

Functions

Functions are declared using the `function` keyword. Functions can have specified parameter types, but they are optional. This is also similar to return types where they can have a specified return type, but it is not required.

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

function add(a:int, b:int): int {
    return a + b
}

greet("Alice")
var total = add(3, 4)
```

Return Statements

Use `return` to exit a function and optionally return a value.

```
function isEven(n:int): bool {
    return n % 2 == 0
}
```

List Literals

Lists can be any type in Catscript.

```
var numbers = [1, 2, 3, 4]
var words = ["cat", "dog", "bird"]
```

Index can access elements

```
print(numbers[0]) // 1
print(words[1]) // 2
```

Example Program in CatScript

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

var numbers:list<int> = [1, 2, 3, 4, 5]

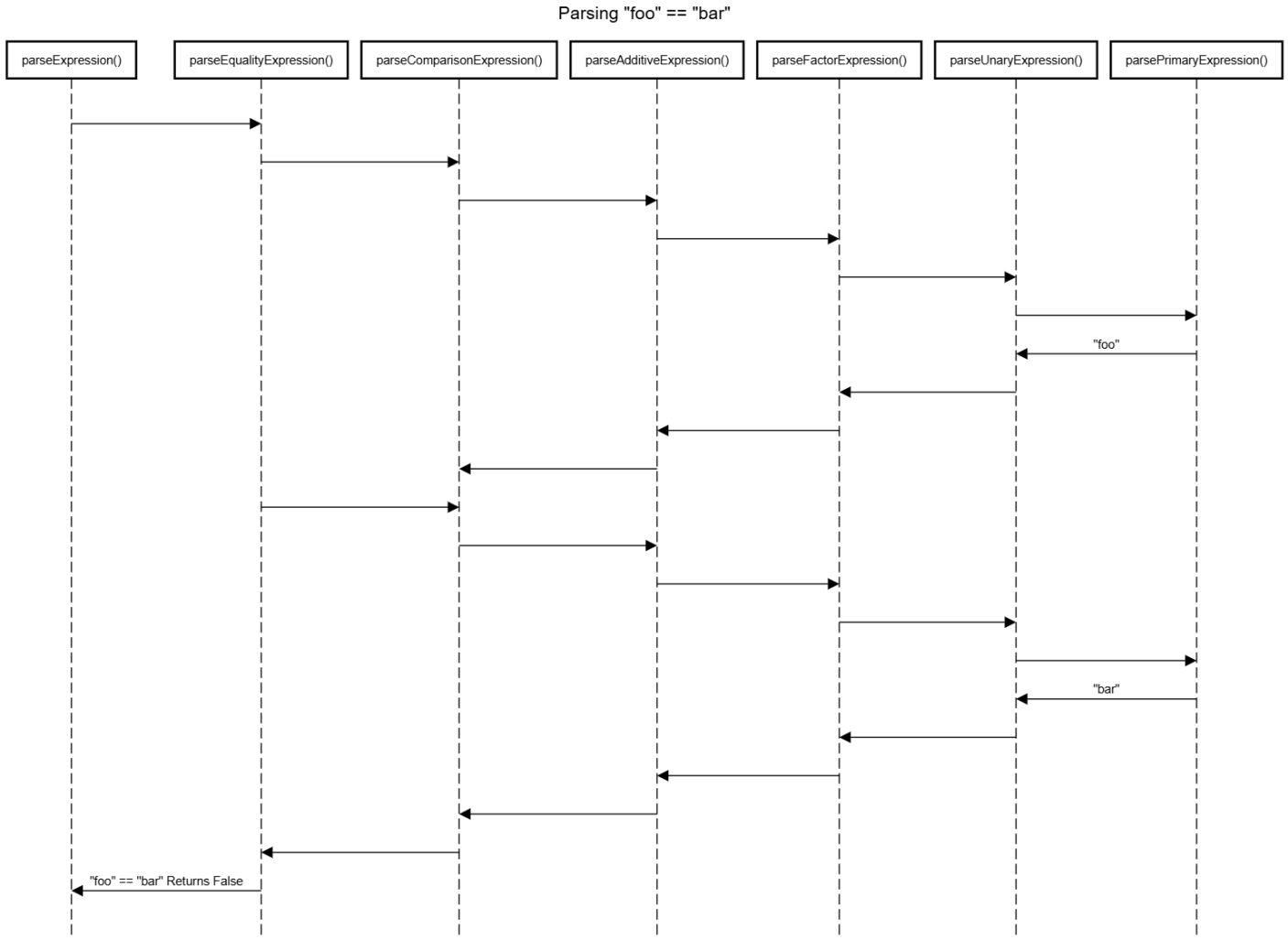
for (num in numbers) {
    var result = square(num)

    if (result % 2 == 0) {
        print("Even square: " + result)
    }
    else {
        print("Odd square: " + result)
    }
}
```

Summary

Catscript was built as a capstone project. It was meant to highlight the core aspects of compilation. Catscript uses lexing, parsing, abstract syntax tree construction, type checking, evaluation, and bytecode generation. It was meant to show how a recursive decent algorithm can parse through code and compile it down to java bytecode. Overall, catscript was created to be a learning tool to learn how compilers work.

Section 5: UML



Section 6: Design Trade-offs

In a hand-written parser like what has been done here for Catscript, there are many tradeoffs when comparing it to a parser generator class. First, you have full control over error messages and recover, and it is also easier to handle ambiguous grammars. On top of this, a hand-written recursive descent parser is much easier to debug through and it is much more readable. The biggest downside of doing something like a hand-written recursive descent parser is the speed of implementation compared to something like a parser generator. Parser generators can also handle more complex grammars automatically and grammar changes are easy to make.

Overall, the biggest tradeoff between hand-written parsers and parser generators is control vs. convenience. Parser generators are quick to get running, but a hand-written parser gives way more flexibility and ease of debugging.

Section 7: Software Development Life Cycle Model

We used Test Driven Development as our life cycle model.

Team Member 2 was asked to write three unit tests to test if the code for the Catscript compiler was working. The tests were asked to be end-to-end to test the full functionality of the compiler. Something that tested tokenization, parsing, semantic analysis, evaluation, and bytecode generation.