CSCI 468 - Compilers Portfolio

Ben Heinze, Braxton McCormack

Section 1: Program

The source code has been zipped into the file “source.zip” and exists in the capstone folder.

Section 2: Teamwork

Describe how your team worked on this capstone project. List each team member’s primary contributions and estimate the percentage of time that was spent by each team member on the project. Identify team members generically as team member 1, team member 2, etc.

For Montana State University’s Computer Science Compilers Capstone, I (Ben Heinze) worked on this project with Braxton McCormack. I was responsible for the implementation of the Catscript Language and Braxton was responsible for the documentation along with tests to verify my compiler is running successfully.

Here are the tests given for me to test my code:

```
//Tests if nested function calls work properly.

    @Test
    void nestedFunctionCallWorksProperly() {
        assertEquals("6\n", executeProgram(
                "function double(n : int) : int { return n * 2 }\n" +
                "function main() : int { return double(3) }\n" +
                "print(main())"));
    }

//Tests a complex Conditional Statement
    @Test
    void complexConditionalTest() {
        assertEquals("greater\n", executeProgram(
                "var x = 10\n" +
                "if(x > 5) {\n" +
                "  if(x > 9) {\n" +
                "    print(\"greater\")\n" +
                "  } else {\n" +
```

```
                "    print(\"smaller\")\n" +
                "  }\n" +
                "} else {\n" +
                "  print(\"else part\")\n" +
                "}\n"));
    }

//Tests complex concatenation
    @Test
    void stringConcatenationWithExpressionsTest() {
        assertEquals("result: 3", evaluateExpression("\"result: \" + (1 +
2)"));
        assertEquals("1 plus 1 equals 2", evaluateExpression("1 + \" plus 1
equals \" + (1 + 1)"));
    }
```

Section 3: Design pattern

The design pattern implemented in this project is memoization. Memoization is a coding technique implemented for optimization/efficiency purposes: instead of having the computer constantly recalculating a result, you can instead store that result in a data structure like a hashmap or array for quick recalls. This reduces the lookup time of the result to O(1).

Memoization has a specific use in terms of the Catscript compiler capstone. Catscript has seven data types: integers, strings, booleans, objects, nulls, voids, and listTypes. Catscript's listTypes work almost the same as Python's lists except the contents of the list have to be the same typing. Instead of generating a new listType everytime a listType is created, I implemented memoization to cache a new listType everytime one is created, then return the listType if the listType has already been created.

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

Catscript Documentation

Braxton McCormack
04/25/24

1 Introduction

At its core, Catscript is a statically typed programming language designed to compile into JVM Bytecode. It utilizes a recursive descent parser and handles type inference, objects, primitives, among other capabilities highlighted in this documentation.

2 Expressions

2.1 Primary Expression

The base expression is the Primary Expression, which may be any of the following:

- Identifier
- String literal
- Integer literal
- Boolean literal
- Null literal
- List literal
- Parenthesized expression
- Function call

In the end, every expression or statement involving another expression simplifies down to a primary expression. ### 2.2 Identifier Expression

The Identifier Expression represents how variables are implemented in Catscript. It contains the variable's name as a string and specifies the variable's type, since Catscript operates as a statically typed language. This variable name is then used to access the associated value in the symbol table. ### 2.3 Unary Expression

The Unary Expression in Catscript implements the 'not' and negative '-' operators. Valid Catscript unary expressions include:

```
not true
-1
```

2.4 Factor Expression

The Factor Expression in Catscript is used to handle multiplication and division operations, denoted by '*' and '/'. Valid Catscript factor expressions include:

```
2 * 5
10 / 2
```

2.5 Additive Expression

The Additive Expression in Catscript is used for addition and subtraction operations, represented by '+' and '-' respectively. The '+' operator is also overloaded to facilitate string concatenation. Valid Catscript additive expressions include:

```
5 + 5
10 - 5
"cat" + 5
"cat" + "script"
```

2.6 Comparison Expression

The Comparison Expression in Catscript is used for relational operations, including strictly greater than '>', strictly less than '<', greater than or equal to '>=', and less than or equal to '<='. Valid Catscript comparison expressions include:

```
2 > 1
1 < 2
5 / 1 <= 5
5 / 1 >= 5
```

2.7 Equality Expression

The Equality Expression in Catscript is designed for handling the operations 'equal to' '==' and 'not equal to' '!='. Valid Catscript equality expressions include:

```
foo == bar
x != y
```

3 Statements

3.1 Print Statement

A very basic but crucial feature in Catscript is the Print Statement, which begins with the 'print' keyword followed by an expression. The statement evaluates the expression and then displays the result to the standard output. Inputs to the Print Statement must be expressions, as statements that do not produce a value are not considered valid. The following are examples of valid Catscript Print Statement calls:

```
print("foo")
print(foo)
print(foo + "bar")
```

3.2 Variable Statement

In Catscript, the Variable Statement is utilized to declare and assign variables. It is a requirement that every new variable declaration includes an expression to assign as a value; simply declaring a variable without an assignment is not allowed. Below are examples that demonstrate what is needed for variable declarations and assignments in Catscript: 1. A required "var" keyword 2. A required variable name 3. A required "=" symbol 4. A required expression In Catscript, when a type is specified, the system ensures that this type can be assigned from the type returned by the expression's getType() method. If they do not match, an Incompatible Types error is thrown. Conversely, if no type is specified, the type of the variable is inferred from the type provided by the expression's getType() method. Below are examples of valid Catscript variable statements:

```
var foo : int = 3
var foo : object = 5
var foo = 2
```

3.3 Assignment Statement

In Catscript, the Assignment Statement allows for the modification of an existing variable's value. The established syntax for executing an assignment statement in Catscript is: 1. A required identifier 2. A required "=" symbol 3. A required expression Below are examples of valid Catscript Assignment statments:

```
foo = "bar"
foo = bar
foo = 1 + 3
```

3.4 If Statement

Similar to its counterparts in other programming languages like Java, the Catscript If Statement determines whether to execute the subsequent statements. The format for constructing an if statement in Catscript is: 1. A required "if" keyword 2. A required "(" symbol 3. A required expression 4. A required ")" symbol 5. A required "{" symbol 6. A required series of statements to execute 7. A required "}" symbol 8. An optional "else" keyword 1. An optional if statement or 2. A required "{" symbol 3. A required series of statements to execute 4. A required "}" symbol Below are examples of valid Catscript if statement:

```
if (foo == "foo") {
    print("foo")
} else if (foo == "bar") {
    print("bar")
}
```

3.5 For Statements

The Catscript For Statement syntax employs the 'in' keyword and is designed exclusively for list iteration. Unlike in some languages where you can count up to a specific value with loops like `int i = 0; i < 5; i++` in C-style languages or `i in range(5)` in Python, Catscript does not support these constructs. To create a for statement in Catscript, follow this structure: 1. A mandatory 'for' keyword 2. A mandatory '(' symbol 3. A mandatory variable name 4. A mandatory 'in' keyword 5. A mandatory expression for iterating 6. A mandatory ')' symbol 7. A mandatory '{' symbol 8. A mandatory list of statements to execute 9. A mandatory '}' symbol Below is an example of a valid Catscript for statement:

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

3.6 Function Definition Statement

The Function Definition Statement allows for the creation of functions that can be invoked from different parts of the program. Functions can explicitly specify a return type, or default to returning void if no type is indicated. When a return type is declared, the program ensures that the function indeed returns that specific type. Parameters are declared in a list format, with each parameter name followed by an optional colon ":" and its type. To define a function in Catscript, the following steps are taken:

1. A required "function" keyword
2. A required function name
3. A required "(" symbol
4. A parameter list
5. A required ")" symbol
6. An optional ":" symbol followed by the function's return type
7. A required "{" symbol
8. A series of statements to evaluate
9. A required "}" symbol

The following is a valid Catscript function definition:

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

3.7 Function Call Statement

Following the definition of a function, it can be executed using a Function Call Statement. The program checks for a match in both the number and type of arguments as per the function's definition and confirms the existence of a similarly named function in the symbol table. Here's how to formulate a function call in Catscript:

- A required function name
- A required "(" symbol
- A list of arguments
- A required ")" symbol

Below is an example of a valid Catscript function call:

```
ab(a, b)
```

3.8 Return Statement

The Return Statement in Catscript is designed to terminate a function and may also return a value. It must be executed within a function; if placed outside, the program will generate a Syntax error. Here's how to create a Catscript return statement:

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

Type System

Catscript features a basic type system. Valid types are as follows:

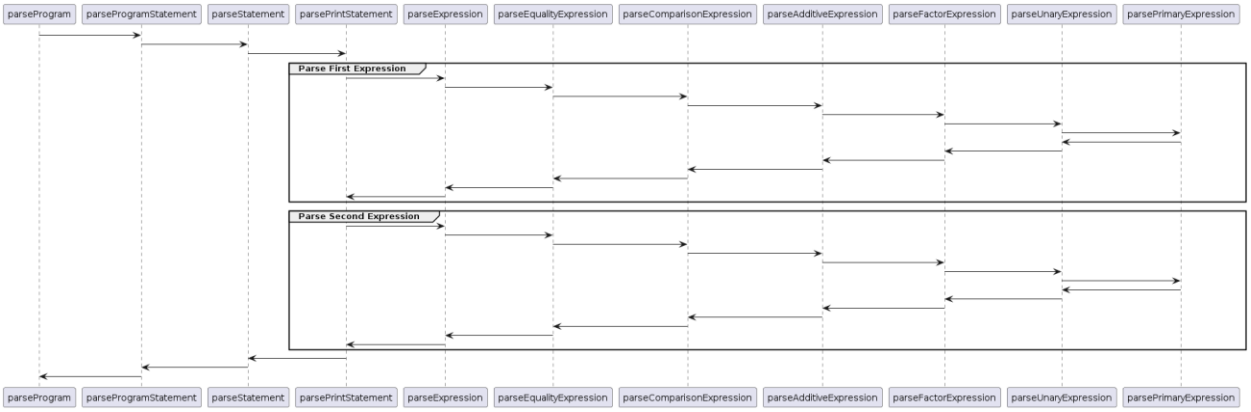
- `int` - a 32-bit integer
- `string` - a Java-style string
- `bool` - a boolean value
- `list` - a list of values with the type 'x'
- `null` - the null type
- `object` - any type of value

Section 5: UML.

Include a UML diagram for parse elements

The UML Diagram below demonstrates the sequence of how recursive descent is used to parse the expression `print(4+3)`

UML Diagram



Recursive descent is a popular compiling algorithm used to generate a tree. The bottom-most expressions will be evaluated, then the results will bubble up to the branch above until there is only one expression left. In this example, `4+3` needs to be evaluated before the result can be printed. In my recursive descent hierarchy, `parseEqualityExpression` calls `parseComparisonExpression` before the equality expression can be fully evaluated, and `parseComparisonExpression` calls `parseAdditiveExpression` before the comparison is fully evaluated, and so on. This descent continues all the way until the last function `parsePrimaryExpression`. Through this recursive descent, the algorithm checks every possible expression. In the UML Diagram above, the `4` gets parsed in the first **Parse First Expression** section. This will bubble up to the additive expression on the left hand side, then the right hand side `3` needs to be evaluated. This is encompassed in the **Parse Second Expression** section. This right hand side will bubble up to the `parseAdditiveExpression`, where it will be evaluated, then will be returned to `parsePrintStatement`. Once the

recursive descent is complete, the print statement will be called to execute the correct answer of 7.

Section 6: Design trade-offs

In the world of compilers, there are two major options: recursive descent and parser generator. I manually implemented recursive descent for this compilers project. Understanding the differences between recursive descent and parser generators is important for choosing which option is more suitable for implementation.

First, when it comes to the parsing process, recursive descent has full control while parser generators do not. For example, ANTLR is a parser generator tool used for reading, processing, and executing structured text. Since ANTLR does most of the heavy lifting, you cannot control the full parsing process ANTLR performs. If you write a recursive descent tree yourself, you can control every aspect of the parsing. Next, recursive descent is easier to read while parser generator's code can be longer and much more difficult to decipher. This is very important when it comes to debugging. Parser generators include less developer-written code, and it is not necessary to understand concepts like the tokenizing API, but in recursive descent there is a lot more handwritten code, and understanding tokenization is a prerequisite for recursive descent. Recursive descent is a very simple and profound algorithm which makes it more appealing to many people in the industry, but parser generators are generally loved in academics because of its complexity. There is a general rule to never edit generated code since the work will be overwritten once the generator is executed again. Some are tempted to edit generated code because developers don't have full control over the parsing process when it comes to parser generators, but this problem does not exist in recursive descent. Finally, recursive descent is not as efficient as parser generators. Overall, I believe the full-control and simplicity of recursive descent outweighs the complexity of parser generators.

Section 7: Software development life cycle model

There are pros and cons about using Test-Driven Development. Starting with the pros, Test-Driven Development provided a finite number of explicit goals. The tests helpfully provided the results my code was expected to produce which made debugging the code easier. Test-Driven Development also allows for quick, repetitive testing. There were several times when I attempted to fix a function in my code and ended up breaking several other tests in the process, and I was able to tell which aspects of my code broke. If Test-Driven Development was not used in this project, it would have been harder to find which aspects of my code were broken. Test-Driven Development dramatically reduced my debugging time. A con for Test-Driven Development includes being overly-reliant on the tests themselves. My tests are all passing but when I attempted to code using my Catscript Compiler, the code did not always work as expected. I had to fix several issues that Test-

Driven Development did not check for. Another con is that Test-Driven Development heavily relies on the types of tests used. Yes, JUnit allows to test for coverage. This is very helpful to determining how much of your code is being tested by the asserts, but even this is not perfect. This exerience taught me that it's important to perform multiple types of testing rather than being heavily reliant on one form of testing.