

John Chiki partnered with Felicia Jayasaputra

Capstone Document for 468 Compilers

Section 1: Program

Please include a zip file of the final repository in this directory.

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 this project, team member 1 was responsible for providing a few tests for the test suite as well as documentation of the language. Their estimated time on the project was 5-10 hours. Team member 2 was responsible for building the parser, tokenizer, and compiler. Their estimated time on the project was 50-100 hours.

The tests that team member 1 was responsible for are as follows:

Written by Felicia Jayasaputra

```
@Test
void nestedIfStatement() {
    assertEquals("18\n", executeProgram(
        "var x = 18 if(x > 10){ "+
        "    if (x < 20) {" +
        "        print(x) " +
        "    } } else { " +
        "        print( 10 ) " +
        "    }" ) );
}

@Test
void functionWithPrint() {
    assertEquals("1\n", executeProgram("function hi() {print(1)}" + "hi()"));
}

@Test
void printVarStatementWork() {
    assertEquals("3\n", executeProgram("var x = 3 " + "print(x)"));
}
```

Section 3: Design pattern

Caching can allow for the compiler to run with a faster compile time and the ability to be lightweight since the compiler can run faster since it is not making more objects than it needs to, and

references already created objects. This can decrease the system requirements and thus allow for the compiler to run on lower hardware. The disadvantage of the memoization implementation here is that it is not thread safe, and thus limited to a single thread. This could be solved with a hash map.

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

Markdown: written by Felicia Jayasaputra

Introduction

Welcome to Catscript Programming Language. Catscript is a programming language.

that allows a user to perform simple programming with different features and functionalities.

Expressions

The Additive Expression

The Additive expression computes addition and subtraction for mathematical operation.

The addition also handles concatenation of strings.

The symbols are as follows:

- - Addition
- - Subtraction

```
print(8 + 7) // Output: 15
```

```
print(8 + "a") // Output: 8a
```

```
print("a" + 12) // Output: a12
```

```
print("hello" + "jazmin") // Output: hellojazmin
```

```
print(8 - 7) // Output: 1
```

The Factor Expression

The Factor expression computes multiplication and division for mathematical operation.

The symbols are as follows:

- - Multiplication
- / Division

```
print(8 * 7) // Output: 56
```

```
print(8 / 2) // Output 4
```

The Unary Expression

The unary operators are operators that require only one operand that produce a new value.

Two types of unary operators in Catscript:

- NOT Operator (not)
- Unary Minus Operator (-)

Logical NOT Operator (not) will invert the logical boolean value.

If x is true, then (not) true will make it to false

Unary Minus Operator (-) is a unary minus operator that change the sign of the argument.

```
int a = 8
int b = -a // This will make the value of b = -8
```

The Equality Expression

The Equality expression decides if one operand is equal or not equal to another operand in the expression. The symbols are as follows:

- != not equal
- == equal to

```
if (8 == 8){
  print("Equal") // Output: Equal
}

if (true == true){
  print("Equal") // Output: Equal
}

if (8 != 7) {
  print("Not Equal") // Output: Not Equal
}
```

```
if (true != null){  
  
print("Not Equal")    // Output: Not Equal  
  
}
```

The Comparison Expression

The Comparison expression decides if one operand is greater than, greater or equal, less or equal, and less than.

The symbols are as follows:

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

```
if (8 > 7){  
  
print("Greather than")    // Output: Greather than  
  
}  
  
if (8 >= 8){  
  
print("Greather or equal to")    // Output: Greather or equal to  
  
}  
  
if (8 < 9) {  
  
print("Less than")    // Output: Less than  
  
}  
  
if (8 <= 9){  
  
print("Less or equal to")    // Output: Less or equal to  
  
}
```

Type Literal

The type literal consist of different types of data that can be used. The types are as follows:

- int - represent an unsigned 32-bit integer
- string - a sequence of combined characters
- bool - for checking true/false conditions
- object - every type can be an object

- null - represent an empty value
- list - used to store ordered collection

Statements

For Statement

The for statement iterates over all the values until there are no more values left.

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

If Statement

The If statement executes your program only when the condition is met.

```
x = 20;  
  
if(x > 10){  // check to see if x is greater than 10  
  print(x);  // If x is greater than 10, print the number. Output: 20  
}
```

If Else Statement

If else statement - gives an alternative way if the condition is not met. If the condition is met, it will execute the if statement body statement. However if the condition is not met, it will execute the else body statement.

```
x = 20;  
  
if(x > 10){  // check to see if x is greater than 10  
  print(x);  // If x is greater than 10, print the number. Output: 20  
} else {  
  
  print("number less than 10");  // If x is less than 10. Output: number less  
  than 10  
}
```

Variable Statement

The variable statement assign a specific value to a name and use that name to represent that specific value. It can be assign with an implicit or explicit type.

- `var x = 10` (Implicit type)
- `var x : int = 10` (Explicit type)
- `var x : bool = true` (Explicit type)
- `var x : list = [1, 2, 3]` (Explicit type)

```
var x = 10

print(x)  // Output: 10

var x : bool = true

print(x)  // Output: true

var x : list<int> = [1, 2, 3]

print(x)  // Output: 1 2 3
```

Print Statement

The print statement is used for printing a specific values.

```
print(1)  // Output: 1

print("Catscript")  // Output: Catscript
```

Return Statement

The return statement is used for returning the value of the specific function code when it finished executing.

```
function hi(){

return 8  // return the value 8

}

print(hi())  // Output: 8

function hi() : int {

return 10  // return the value 10 which is an integer
```

```
}  
  
print(hi()) // Output: 10
```

Function

Function is a segment of code that is executed to perform a specific actions.

It consists of different parts that make up a function, which is
the function call, function parameter and function definition statement.

```
function hi(a, b, c) {  
    print (a);  
}
```

NOTE:

- hi(a, b, c) is function call
- a, b, c is function parameter
- function x() {} is function definition statement

Section 5: UML.

Code for the sequence diagram below.

“

title Catscript Addition Sequence Diagram

participant User

participant CatScript

participant Lexer

participant Parser

participant ParseTree

activate CatScript

User->>CatScript:evaluate ("1 + 2 * (3 + 7)")

activate Lexer

CatScript->>Lexer: lex("1 + 2 * (3 + 7)")

CatScript<--Lexer: tokens

deactivate Lexer

activate Parser

CatScript->>Parser: parse(tokens)

Parser->>Parser: parseExpression()

Parser->>Parser: parseAdditiveExpression()

Parser->>Parser: parseFactorExpression()

Parser->>Parser: parseIntegerLiteral()

CatScript<--Parser: parseTree

deactivate Parser

activate ParseTree

CatScript->ParseTree: execute()

CatScript<--ParseTree: "21"

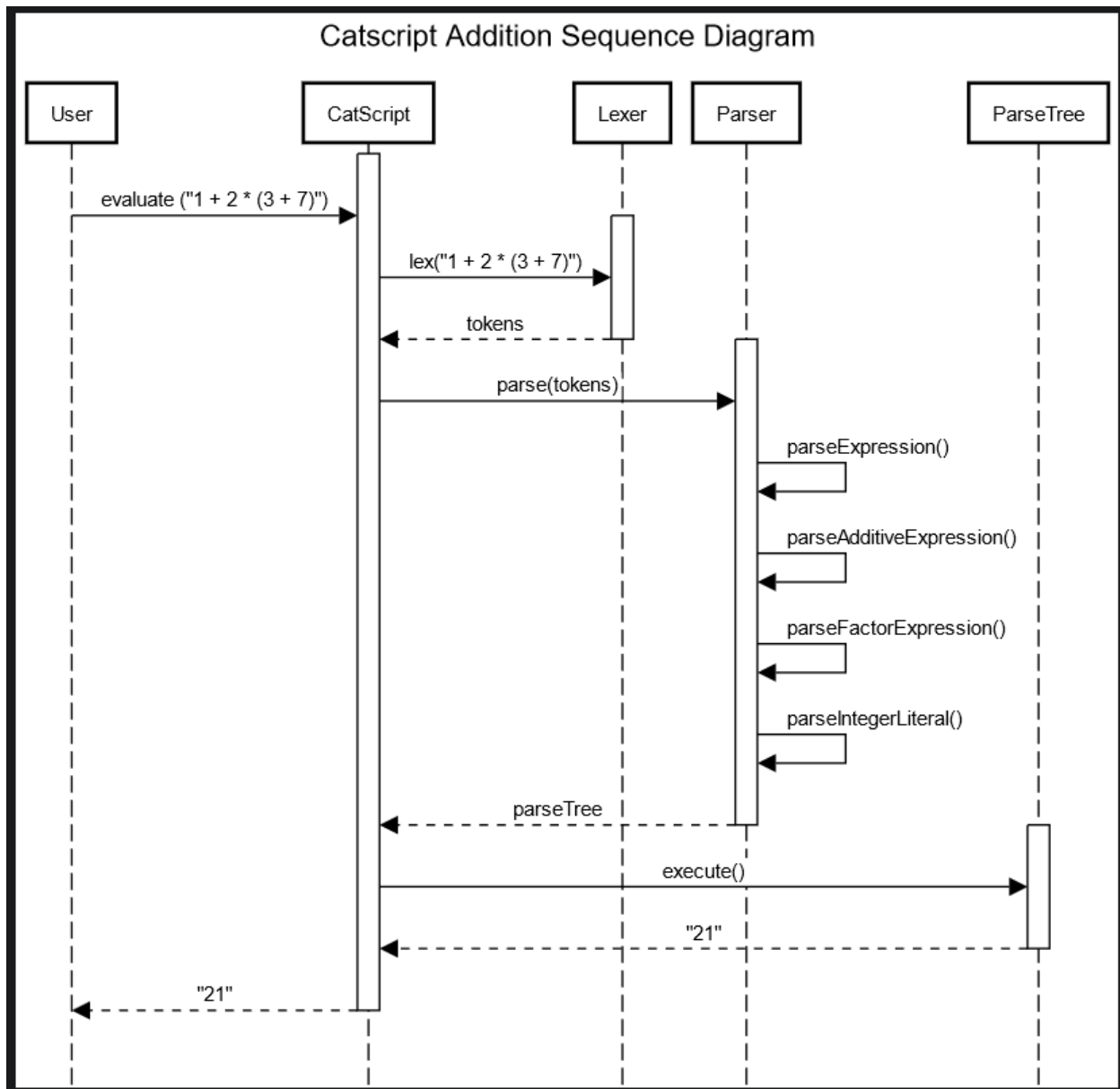
deactivate ParseTree

User<--CatScript: "21"

deactivate CatScript

”

This is a sequence diagram for an arithmetic evaluation in the language.



Section 6: Design trade-offs

When building a parser, there are really 2 popular ways to do so, by hand or with a parser generator using a regex. For this project, building by hand was chosen for several reasons, one big one being to learn more about how a parser works and if built by hand, it is far easier to debug and to read. Parser generator parsers are significantly larger than a by hand recursive decent parser, and much harder to read, as well as being very specific. This makes them less appealing and was also a consideration into why the by hand method was chosen. Another reason for a by hand build recursive decent parser, was that it is more general even if it is given vague rules, where a generated parser may struggle to be.

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project, this helped us by knowing what functionality was required to work, as well as knowing how the features interacted with each other. This also allowed us to make sure that all functionality was working at all stages of the project. The creation of tests is the worst part of TDD, since knowing what makes a good test can be difficult. As well as coming up with the tests being time consuming, the making of the tests is the large downfall of TDD for a project.