

Compilers Portfolio

CSCI 468 – Spring 2023

Cory Janes and Dillon Shaffer

Section 1: Program

The link to our zipped file in the project directory: *capstone/portfolio/source.zip*

Section 2: Teamwork

Most of the work was done on an individual basis, but there was plenty of ideas and concepts that Dillon and I discussed throughout the compiler project. Communication and teamwork became more involved as the semester drew to a close and we collaborated with test writing, documentation structure and finalizing our capstone.

Dillon and I created additional unit tests for each of our projects to test the robustness of our coding in the project. Dillon's tests can be found in the directory *test/java/edu.montana.csci.csci468/partnertests/CatscriptParserLocationTest* and below are a few of the tests:

@Test

```
public void integerLiteralExpression() {  
    IntegerLiteralExpression expression = parseExpression("1");  
    MatchToken token = new MatchToken().start(0).end(1).line(1).offset(0);  
    assertEquals(token, expression.getStart());  
  
    token = new MatchToken().start(0).end(1).line(1).offset(0);  
    assertEquals(token, expression.getEnd());  
}
```

@Test

```
public void stringLiteralExpression() {  
    StringLiteralExpression expression = parseExpression("\"hello\"");  
    MatchToken token = new MatchToken().start(0).end(7).line(1).offset(0);  
    assertEquals(token, expression.getStart());  
    assertEquals(token, expression.getEnd());  
}
```

@Test

```
public void stringLiteralExpressionWithEscapes() {  
    StringLiteralExpression expression = parseExpression("\"\\\\\"hello\\\\\"");  
    MatchToken token = new MatchToken().start(0).end(11).line(1).offset(0);
```

```

    assertEquals(token, expression.getStart());
    assertEquals(token, expression.getEnd());
}

```

@Test

```

public void identifierLiteralExpression() {
    IdentifierExpression expression = parseExpression("foo", false);
    MatchToken token = new MatchToken().start(0).end(3).line(1).offset(0);
    assertEquals(token, expression.getStart());
    assertEquals(token, expression.getEnd());
}

```

@Test

```

public void booleanLiteralExpression() {
    BooleanLiteralExpression expression = parseExpression("true");
    MatchToken token = new MatchToken().start(0).end(4).line(1).offset(0);
    assertEquals(token, expression.getStart());
    assertEquals(token, expression.getEnd());

    expression = parseExpression("false");
    token = new MatchToken().start(0).end(5).line(1).offset(0);
    assertEquals(token, expression.getStart());
    assertEquals(token, expression.getEnd());
}

```

@Test

```

public void nullLiteralExpression() {
    NullLiteralExpression expression = parseExpression("null");
    MatchToken token = new MatchToken().start(0).end(4).line(1).offset(0);
    assertEquals(token, expression.getStart());
    assertEquals(token, expression.getEnd());
}

```

Section 3: Design Patterns

One of the design patterns used in Catscript is the Memoization method. Also known as tabling, memoization is a technique that stores expensive function calls and returns that cached result when the same input is used or inputted in the program. The benefit of this method is a speed boost by remembering results and bypassing the function calls needed for those results. Below is a code snippet with a memoization example:

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {

    CatscriptType listType = LIST_TYPES.get(type);

    if (listType == null) {

        listType = new ListType(type);

        LIST_TYPES.put(type, listType);

    }

    return listType;

}
```

Section 4: CatScript Documentation

Introduction

The CatScript programming language is a simple, non-Turing complete scripting language. Due to a lack of assignment expressions, custom data types, and a module/package system—the language is largely incomplete at this time.

- ♦ Non Turing-Complete
 - The language does not support list assignment Basic
- ♦ Type System
 - Integer, String, List types
 - Generic Object, Null, and Void
- ♦ Function support & top-level statements Simple
- ♦ for-loops and complete if-statements

Features

The CatScript scripting language has the basic statements and expressions found in most C++/Java-based programming languages. It has support for basic if statements, for-statements.

Type System

The CatScript type system is statically but weakly typed. This means that all types are known by the compiler at compile time but the “boundaries” between different types are soft (see the **object** type).

Assignable Types

Integers

The syntax for integers in Catscript is similar to that in which you will find in most programming languages: Regex: `[0-`

`9]+`

Type Keyword: **int**

Type Size: 4 bytes

Given CatScript’s simple type system, there is only a single (32-bit) signed integer in the language.

Booleans

Once again, the syntax for booleans is similar to most Java/C++ inspired programming languages.

Regex: `true|false`

Strings

Like integers and booleans, the syntax for string tokens are similar to many other mainstream programming languages:

Regex: `“[(\\”)(\\n)|.]*”`

Type Keyword: **string**

As you can see in the grammar, CatScript strings allow quotation and newline escaped characters.

`\n` → ASCII 10 - Linefeed

`\"` → ASCII 34 - Quotation

CatScript does not provide anyway to use unicode escapes

Lists

Like other simple scripting and programming languages, CatScript the simple array syntax for lists.

Regex: `\[(<expr> (, <expr>)*)? \]`

<expr> is the regex for a CatScript expression

Type Keyword: **list** or **list<<type>>**

<type> is any CatScript type (including another list)

*If not specified, the component type of a list defaults to **object***

CatScript lists are read-only (this is the main feature making the language not Turing complete!)

Object

In CatScript, everything is an object. All types listed above this entry are assignable to object. It is the only type to which null can be assigned.

Non-Assignable Types

Void

The void type is not an assignable type (this means that values cannot be assigned to a variable of type void, nor can variables be declared with type void!) This type is used primarily in functions that return no value.

Null

The “null” type is a valid CatScript type, though it has few uses. It is intended for future CatScript updates in which functions will absolutely return a null value. It exists as an alternative to void since a Null type has size as opposed to the zero-size null type.

The null type is not an “assignable” type meaning that it cannot be used in type declarations

Syntax

Now that we have covered the type-system, we can talk more about the format of the syntax of the language. CatScript has three major types of AST elements—declarations, statements, and expressions— listed here hierarchically.

Declarations

In CatScript, there are only two kinds of declarations:

Function Declaration

In CatScript, functions are the only form of code abstraction. Since there are no object-oriented types there are no methods meaning CatScript is a purely non-pure functional language.

Regex: `<type>1 <ident>2 \((<ident>4 (: <type>)?5),*3 \) { <stmt>* }`

1. The return-type of the function. The function must return a value of this type.
2. The name of the function, `<ident> = [a-zA-Z][a-zA-Z0-9_]+`
3. A function can have zero or more arguments as seen here, they must be comma separated
4. Each argument has a name of its own, each argument must have a name unique from the other arguments of that function
5. Each function argument, may optionally be specified a type. If not specified, the type defaults to **object**

```
function doubleIt(value: int): int {
    return value + value;
}

function zero(): int {
    return 3; // lol why not :D
}
```

Statement Declaration

This declaration is really just a wrapper around some statement. Since CatScript is a scripting language, we find it useful to be able to execute statements at the global level.

Statements

Print Statement

In catscript, in order to print out values we must use the JVM. For this reason, the print function is abstracted into a language keyword.

Regex: **print(<expr>)**

Var Statement

The variable statement is responsible for creating variables.

Regex: **var <ident> (: <type>)? = <expr>**

Examples:

```
var y: int = 465
```

```
var z = [1, 2, 3]
```

If Statement

The if-statement is responsible for conditional branching.

Regex: **if (<expr → bool>) { (<stmt>)* } (else if (<expr → bool>) { (<stmt>)* })* (else { (<stmt>)* })?**

The if-statement allows support for else-if branches as well as a singular, final else branch.

Examples:

```
if (true) { print("hi") }
```

```
if (c == d) { ... } else if (true) { ... }
```

```
if (a >= b) { ... } else { ... }
```

For Statement

The for-statement in CatScript is a simple loop statement that allows iteration over elements in a list. The iteration variable is declared in the loop declaration and is read-only.

Regex: **for** (<ident> in <expr>) { (<stmt>)* }

Return Statement

The return statement is used to exit a function early and optionally return a value. If no value is specified, the function returns null.

Regex: **return** (<expr>)?

Expressions

Given that CatScript is a very simple language, it is missing around half of the operators that most OO languages have. These include the increment operators, access operators, index operators, bitwise, and boolean-algebra operators.

Binary Operators

CatScript supports the following binary operators:

- ♦ +, -, *, /: addition, subtraction, multiplication, division
- ♦ ==, !=, <, >, <=, >=: equality, inequality, less than, greater than, less than or equal, greater than or equal
- ♦ !, not: logical AND, logical OR, logical NOT

Precedence

1. Assignment
2. Equality
3. Comparison
4. Addition
5. Multiplication
6. Unary
7. Primary

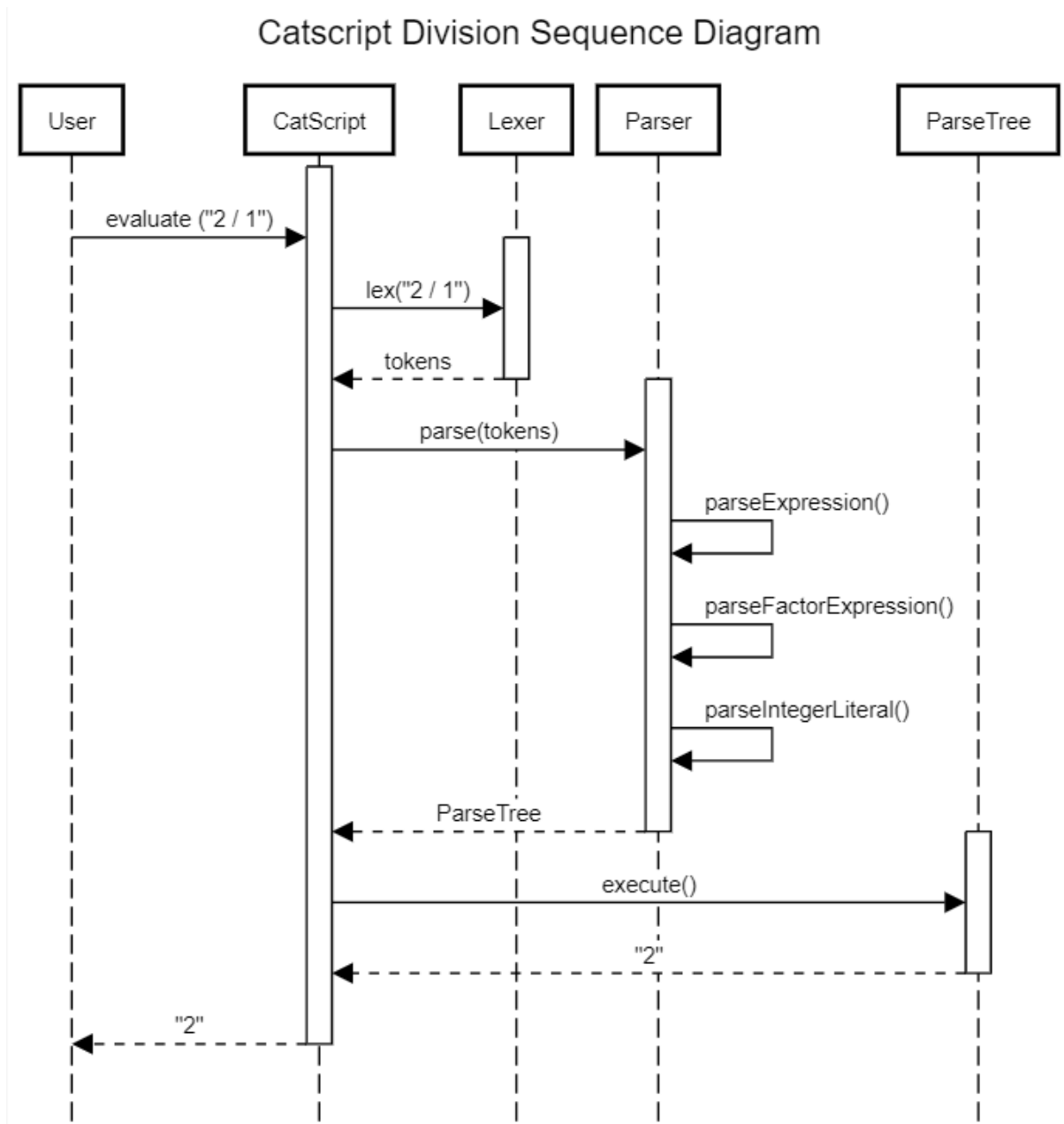
Expressions

Expressions are the core of any programming language. In CatScript, we have the following expressions: Literals:

- ♦ Integer, string, null, and list literals
- ♦ Variables: accessing the value of a variable function
- ♦ Calls: calling a function with arguments
- ♦ Binary operators: performing operations on two values
- ♦ Unary
- ♦ Operators: performing operations on a single value list
- ♦ Operations: list access and list membership operators

Section 5: UML

The UML for parse elements was given in the class and a portion of it was modified to show understanding of the subject, seen below:



- 1) The program is asked to evaluate a statement
- 2) The CatScript tokenized in the Lexer and sent to the parser
- 3) Through recursive descent, the Parser finds the appropriate function for the tokens
- 4) The function is executed and the ParseTree returns the outcome to the user

Section 6: Design trade-offs

CatScript has some trade-off because of its educational design. Simplifying the compiler creates some limitations while letting students learn the basics of its generalized process. CatScript uses a recursive descent algorithm instead of the often-taught parser generator. Recursive descent parsers are slower at parsing large expressions, but being loop-based, the parser has less problems. As mentioned, the recursive descent algorithm gives novices a better picture of how a compiler works with straight forward coding blocks and it can be easily debugged.

Another trade-off for ease of understanding is the depth of expression layers that are not reached in CatScript. Java has 12-14 expression layers, while CatScript has only 6 layers for its expressions.

Section 7: Software Development Life Cycle

The Test-Driven Development model was used in this project. This is a process that relies on software requirements that are turned into test cases. Unlike testing after a finished product, this is done during software development to test new code repeatedly against all test cases. This greatly increases the robustness of the software dynamically.