

CSCI 468 Compilers Captsone Documentation

Spring 2023

Team Members

Mitchell Froelich
Brendan Harrington

Section 1: Program

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

Section 2: Teamwork

The distribution of work was sorted into the manner below:

Team member 1: Compiler Engineer and Coder - 90%

Team member 2 (Brendan Harrington): Technical Documentation and Quality Assurance (Testing) - 10%

Team member 2 test examples:

```
@Test
void forStatementWorksWithNestedIfStatement() {
    assertEquals("1\n", executeProgram("for(x in [1, 2, 3]) {" +
        "if(x < 2){\" +
        "    print(x) \\n}\" +
        "}")));
}

@Test
void forStatementWorksWithTripleNestedIfStatementWithElse() {
    assertEquals("1\\n2\\n3\\nhump day!!\\n4\\n5\\n6\\n", executeProgram("for(x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) {" +
        "if(x <= 6){\\n\" +
        "    print(x)\\n\" +
        "    if (x >= 3){\\n\" +
        "        if(x < 4){\\n\" +
        "            print(\"hump day!!\")\\n\" +
        "        }\\n\" +
        "    }\\n\" +
        "}\\n\" +
        "})"));
}
```

```

        "}")));
    }

@Test
void differentReturnTypeGivesError() {
    List<ParseError> errors = getErrors("function foo(x : int) : int {\n" +
        "return \"test\" " +
        "}\n" +
        "print(foo(9))");

    assertError(errors, 0, ErrorType.INCOMPATIBLE_TYPES, 2, 7);
}

private void assertError(List<ParseError> errors, int errorIndex, ErrorType errorType, int line, i
    if (errorIndex >= errors.size()) {
        fail("No error at index " + errorIndex);
    } else {
        ParseError parseError = errors.get(errorIndex);
        assertEquals(errorType, parseError.getErrorType());

        if (line > 0) {
            assertEquals(line, parseError.getLocation().getLine());
        }
        if (lineOffset > 0) {
            assertEquals(lineOffset, parseError.getLocation().getLineOffset());
        }
    }
}
}

```

Section 3: Design pattern

Once pattern that was used in the compiler was the design of memoization. This pattern was implemented in the Catscript Type under the getListType method. This is done to cache types instead of simply creating new ones, to reuse resources instead of taking up more. This especially beneficial to use multithreaded environment.

Example of where this was used in CatscriptType:

```

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

Catscript Documentation

This document is a guide to help new users of Catscript understand some features of Catscript.

Introduction

Catscript is a simple scripting language that was created in CSCI 468. Catscript is a statically typed scripting language that focuses on recursive decent parsing. Catscript is a simple and easy to learn as it has only six data types, and simple features that share similarities to many other programming languages. Catscript's syntax is very similar to Java, requiring curly braces around statement bodies. Catscript is designed to be a simple and easy to learn scripting language that can be used for various purposes, including web development and general scripting.

Here is an example of Catscript:

```
var x = "foo"  
print(x)
```

Data Types

Catscript has the following six data types:

- int - The range of values that can be stored in a Catscript int is -2^{31} to $2^{31}-1$.
- string - This data type stores text values. A string value is enclosed in double quotes in Catscript.
- boolean - This data type stores true or false values.
- list - This data type stores a collection of values. A list can contain values of any data type, including other lists.
- null - This data type is used to represent the absence of a value.
- object - This data type is similar to Java's Object data type and can be used to represent any other data type.

Operators

Catscript supports the following operators:

- Arithmetic operators: (+, -, *, /)
- Comparison operators: (==, !=, <, >, <=, >=)
- Logical operators: !

The + operator also allows for string concatenation as shown below:

```
var x = "foo"
var y = "bar"
print(x + y)
```

Features

Variables

In Catscript, variables must be declared before they can be used. The syntax for declaring a variable is as follows:

```
var <variable_name> = <expression>
```

Assigning Values to Variables:

Variables can be explicit typed by declaring the <type> after the variable name

The syntax for an explicitly typed variable is as follows:

```
<variable_name> : <type> = <expression>
```

Variable Statement example

```
var age = 20
var name = "John"
var is_student = true
var last_name : string = "Smith"
```

Functions

In Catscript, a function can be declared using the `function` keyword, followed by the name of the function, a list of parameters enclosed in parentheses, and an optional return type.

The syntax for declaring a function is as follows:

```
function <function_name>(<parameter_list>) [: <return type>] {  
  // Function body  
  return <return_value>;  
}
```

The `<function_name>` is the name of the function, and it should be a valid identifier (it cannot share names with variable identifiers). The `<parameter_list>` is a comma-separated list of parameters that the function expects. Each parameter should have a name and can be explicitly typed. The types can be any of the six data types supported in Catscript.

Parameter List:

The parameter list is a comma-separated list of parameters that the function expects. Each parameter should have a name. If the parameter has an explicit type it should be declared by having the type declared after a colon following the name. The types can be any of the seven data types supported in Catscript.

The syntax for a parameter with an explicit in a parameter list is as follows:

```
(<parameter_name> : <parameter_type>, ...)
```

Return Statement:

A function can return a value using the return statement. The return statement will usually be followed by the value that the function returns. If the function has a return type declared, the value returned must be of the type declared.

The syntax for the return statement is as follows:

```
return <value>
```

Example Function

```
function foo(x: int, y){  
  for (i in x){  
    print(y)  
  }  
}
```

```
foo(3, "Hello, World!")
```

Control Flow Statements

If Statements

Catscript also has if-else statements as one of the control flow statements. The expression in the parenthesis is evaluated, and if it's true, the statements inside the if block are executed. If the expression evaluates to be false, the statements inside the if block are skipped. if-else statements contain an optional else block.

The syntax for an if statement is as follows:

```
if (<expression>){  
  //Statements  
}
```

The `<expression>` can be any valid Catscript expression that evaluates to a boolean value. For example, the following are all valid conditions:

- `x > y`
- `name == "John"`
- `age >= 18`

In some cases, you may want to execute a different set of statements if the condition in the if statement is false. In these cases, you can use an if-else statement.

The syntax for an if-else statement is as follows:

```
if (<expression>){  
  //True Statements  
} else {  
  //else statements  
}
```

Nested If Statements

You can also nest if statements inside other if statements to create more complex logic. The syntax for a nested if statement is as follows:

```
if (<expression_1>){  
  //Statements to be executed if expression_1 is true  
  if (<expression_2>){
```

```

        //Statements to be executed if expression_1 and expression_2 are evaluated as true
    }
    else{
        //Statements to be executed if only expression_1 evaluates to be true
    }
} else {
    //else statements to be executed if expression_1 evaluates to be false
}

```

Example If Statement

```

var x = 1
if (x > 0) {
    print("I have a dollar")
} else {
    print("I have no money")
}

```

For loops

Catscript also includes the for statement as another control flow statement used to iterate over a sequence of values and execute a set of statements for each value.

The syntax for a for statement is as follows:

```

for (<variable> in <expression>) {
    //Statements
}

```

The `<variable>` is a variable that takes on the value of each item in the `<expression>` during each iteration of the loop. The `<expression>` is any iterable data type, such as a list, that contain a collection of values to iterate over. The curly braces `{}` enclose the statements inside the for block.

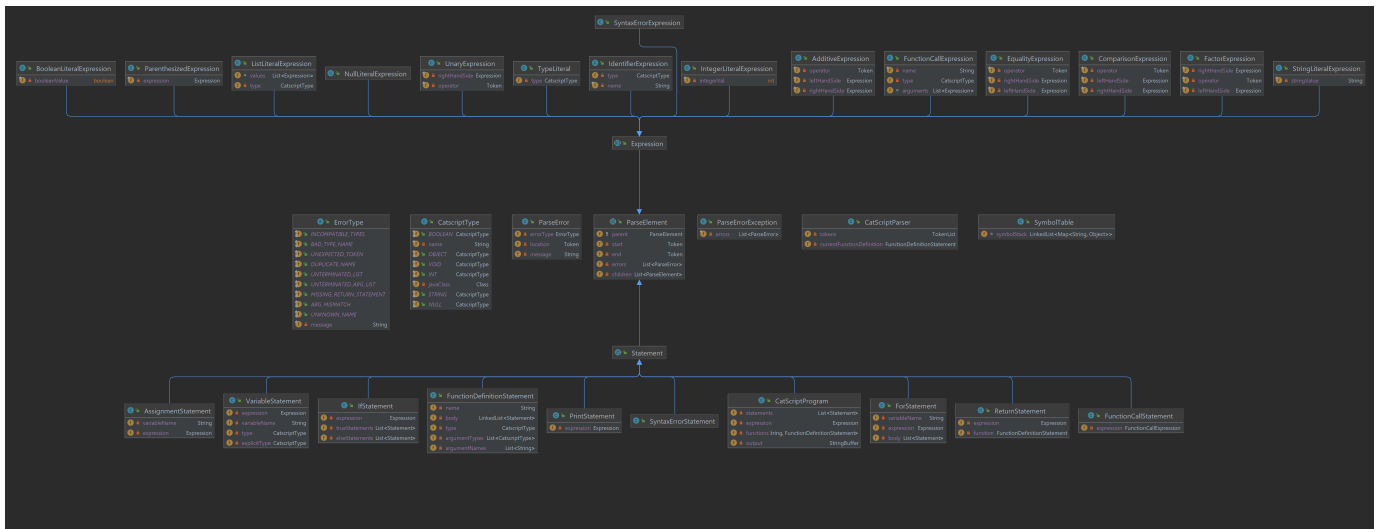
Example For Statement

```

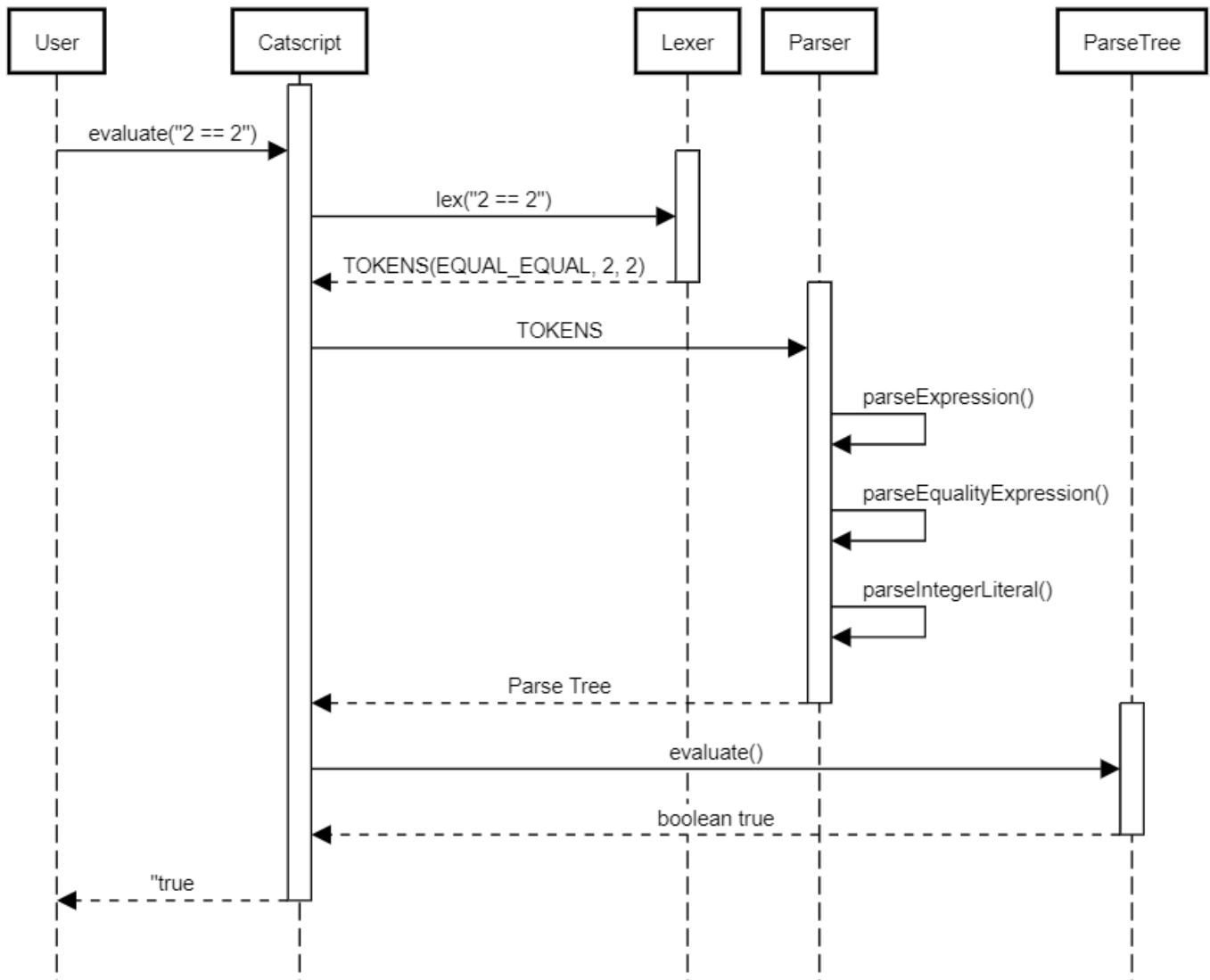
var names = ["Alice", "Bob", "Charlie"]
for (name in names) {
    print(name)
}

```

Section 5: UML.



Catscript Equality Sequence Diagram



Section 6: Design trade-offs

We created a parser by hand, rather than with a parser generator. This allowed for greater understanding of grammars, but this also took far more time than it would if we did in fact use a generator. This allowed us to learn how parsers work in the background. If we had used a parser generator, we would have been more familiar with the industry standard.

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project. We were given tests to drive the coding of all parts of our compiler. This included both the tokenizer and the parser. We were also given the grammar to build off and use to create our syntax expectations. The following is some advantages and disadvantages of using Test Driven Development:

Advantages

Code as Needed:

```
One of the main advantages of this model is that you
only write the code that you need for the assignment given.
You create a test to drive implementation of a feature.
It also keeps the code as simple.
```

Modular Design:

```
TDD is also a modular design, meaning you go from one feature
to the next, saving time. Since you also write the tests
first, you also quickly verify components as you go through them.
```

High Test Coverage:

```
A test for each feature builds high confidence in your code
working most, if not all, of the time.
```

Easy to Refactor:

```
Every feature is formally tested, so you can make changes to large areas
of the code and the code will pass the same way assuming everything is
correctly layed out. As you improve, you can change your code to be better,
and you still end with the same result.
```

Disadvantages

Slower Process to Develop:

You have to devise the tests for the code before you can even start with a line of the code to pass them. This takes time that could go toward the coding process instead.

Tests Must Change:

As requirements shift, you also will need to change the tests to match, not just the code for the tests. You can rewrite it as much as you want, but if a fundamental change is made, you need to change your tests and code to match.