

## **Section 2: Teamwork**

---

We were all required to team up with a partner in the class. They wrote the documentation for CatScript and 3 tests for our parser. We did the same for them. This way, the work of writing the parser was done individually while other jobs were handled by a peer. This is similar to the interaction in the workplace where different jobs are done by different people.

## **Section 3: Design Pattern**

---

## Section 4: Technical Writing

---

Catscript is a c-like programming language designed for educational purposes. Despite its simplicity it does support some advanced language features such as type coercion, recursion, and statement separation without semicolons. A catscript program consists of either one or more statements, executed from top to bottom, or a single expression.

Catscript is a simple, scripting language with C-like syntax. It supports basic control flow, subroutines, and integer math. It uses a strong type system but also supports type inference.

## Types

---

Catscript is a strongly typed language. It supports a small set of types.

- [int](#): 32 bit, signed whole number
- [string](#): Immutable collection of plain text characters.
- [boolean](#): true or false
- [List<T>](#): List of values of type T. Can be iterated by a [for loop](#).
- [object](#): Any type
- [void](#): absence of type. Only allowed as the return type of a [function definition](#).

## Features

---

### strict typing

Catscript is a strongly typed language, which means that variables and function parameters cannot be reassigned with values of a different type.

### type inference

Catscript supports type inference. This means that the types for variables and function return values don't always need to be explicitly stated, unless desired.

### functions

Subroutines with scope, parameters, and return values. Recursion is also supported.

```
function abs(num: int): int {  
    if (num < 0){  
        return -num  
    } else {  
        return num;  
    }  
}
```

```
var number = -3
var absolute_value_of_number = abs(number)
```

## variables

Like most languages, catscript supports named variables which can be assigned and reassigned values.

```
var greating = "Goodbye Moon?"
greeting = "Hello World!"
```

## standard out

Use the print statement to print to standard out.

```
print("Goodbye Moon?")
```

print doesn't need to be given a string value either.

```
print(42)
```

## comments

Catscript supports single line comments which are ignored by the compiler.

```
// the variable known as x
var x =
// the value given to x
3
```

## expressions

---

### function call

Calls a function with the given arguments and outputs the result

```
foo(2, 2 + 3)
var result = bar()
var result2 = biz(1, 2 + result)
```

### literal expression

Simply returns the literal value given in text format.

#### *string literal*

Returns a string containing the character between the quotes. Supports the same escape sequences as java strings, such as `"\n"` (linebreak), or `"\t"` (tab), or `""` (double quote).

```
"Hello World!"
"line1\nline2\nline3"
"\do it to learn about how to write a compiler\" - Dennis Ritchie"
var s: string = "Hello World!"
```

*integer literal*

```
42
var num: int = -42
```

*boolean literal*

```
true
false
```

addition and subtraction

multiplication and division

equivalence

## Statements

### For Loop

Iterates over a list, running a block of code on each value:

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

for( i in lst ) {
    print(i)
}
```

### If Statment

Evaluates a condition and runs its block of code if that condition evaluates to true. If the condition evaluates to false, the optional else block will run.

```
if ( a < 7 ) {
    print("a is less than 7")
} else {
    print("a is not less than 7")
}
```

### Print Statement

Prints to standard out the result of whatever is placed in between the parenthesis.

```
print("Hello world!")
print(true)
print(7)
print(2 * 21)
```

## Var statement

Declares a new variable. An optional type paramety may be provided to provide explicit type declaration; Otherwise, the type is inferred.

```
var foo = "bar"
var num: int = 9
var bln = true
var s: string = "Hello world!"
```

## Assignment

Assigns the result of the right-hand side to the variable on the left-hand side.

```
foo = "biv"
num = 2
bln = false
s = "World!, hello"
```

## Function call

Calls a function, discarding the returned value. This is the only statement that can also be treated as an expression; as long as the return type of the function is not void.

```
foo("bar")
updateDatabaseWidget("tom", "fredrickson", "324-234-2345")
executeFizzBuzz()
```

```
// as an expression
var URL: string = fetchURLString()
var factorialOf5 = fact(5)
var helloTimes5 = repeat("hello", 5)
```

## Function definition

Declares a new function and its behavior.

```
function fact(num: int){

}
```

---

---

---

---

## Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

### Introduction

Catscript is a simple scripting language. Here is an example:

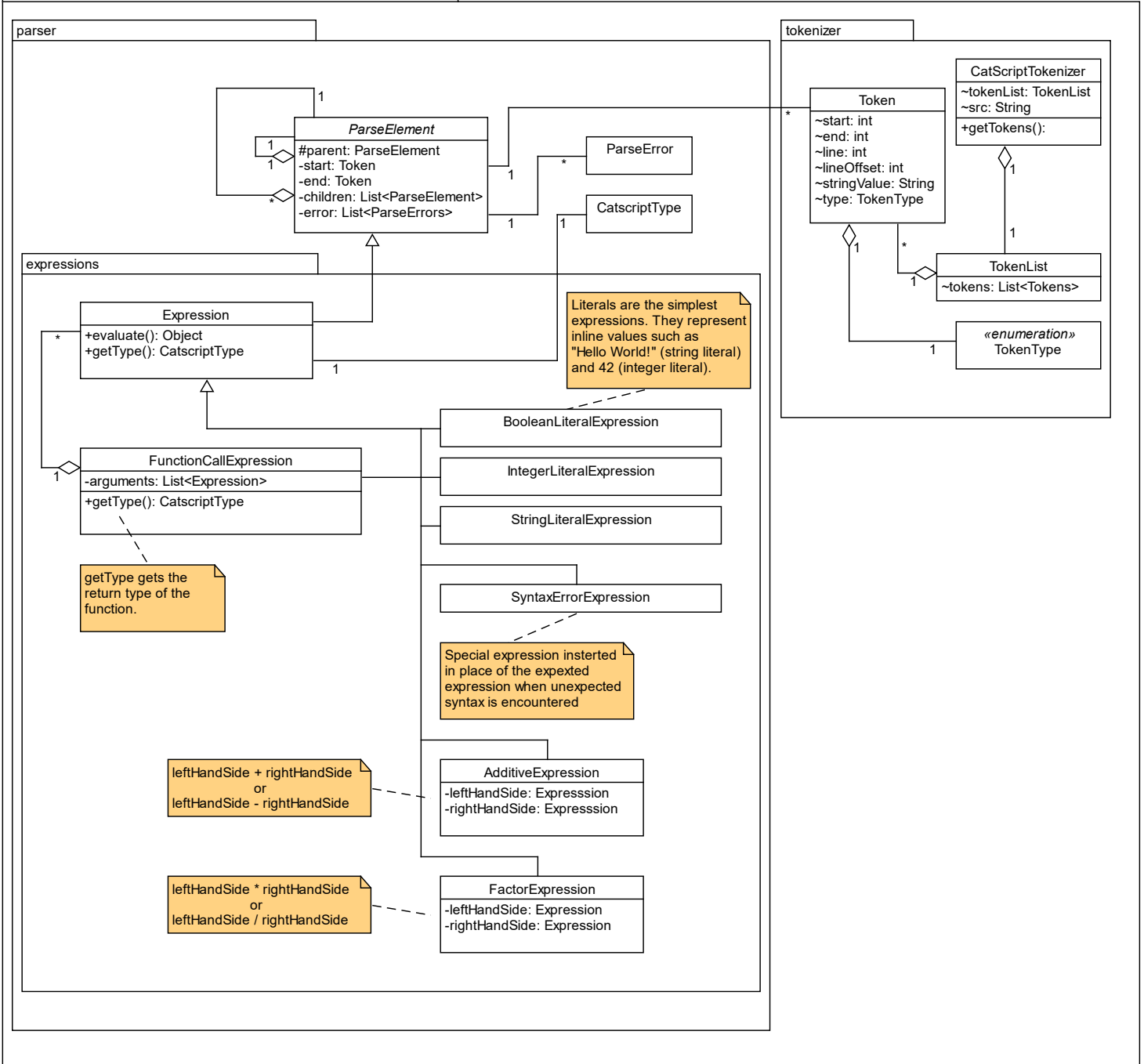
```
var x = "foo"  
print(x)
```

### Features

#### For loops

## Section 5: UML

edu.montana.csci.csci146





## Section 6: Design trade-offs

---

The primary goal of “separation of concerns” is to break up a design into appropriate pieces. Each piece has a job or “concern” that it should stick to without deviation. The benefits of this include big-picture readability and long-term maintainability of large code bases. The cost is a higher overhead complexity, especially in a small codebase like CatScript. This cost is the primary reason why CatScript does not adhere to this design philosophy as much as it could or should.

The separate concerns of parsing, evaluating, and compiling code would optimally be separated into different packages or files but instead we have chosen to define them all in the “parsing” package, despite them not all being related to parsing; awkward naming being one of downsides to this approach. The name, “parsing”, may not be wholly accurate but it would be difficult to find a more accurate name that isn’t too generic, primarily due to how generic the package itself is. For example: the names “components”, or “elements” might be a better fit, but they certainly aren’t very descriptive. The result of such naming is that big-picture readability suffers as someone looking for non-parsing related code might not know where to look.

Additionally, this layout doesn’t bode well for long-term maintainability since the packing of multiple concerns into the same class files will inevitably lead to bloated classes as more features are added. Without breaking these classes up, allowing them to grow horizontally across multiple files, they can only grow vertically by making the files bigger.

However, CatScript is not a long-term project, nor is it large enough for awkward naming to pose a substantial problem. CatScript is a learning project and the goal of such a project is never the final product but rather the experience gained. The goal of CatScript was to learn how to write a parser and focusing too much on achieving a more robust design could have distracted from that goal.

## Section 7: Software development life cycle model

---

Test driven development is a development model in which tests are written first before the code. This way, the tests provide clear and unambiguous requirements while also serving their original purpose of providing a simple way to test the code. This model works well for projects with clear requirements given up front with a reasonable timeframe; Unfortunately, this rarely applies perfectly in industry, where deadlines and requirements are flexible; although, one area where it could fit perfectly is education. In the classroom, requirements are usually fully known up front and the deadlines rarely change.

In our case, the requirements for Catscript were given to us in the form of unit tests, with checkpoints spread across the semester that required different sets of tests to pass. Most of the grade for the main project was determined by these checkpoints. This made grading simpler and reduced a significant source of stress for the students, since we could know ahead of time what grade we’d receive. Effort was still required to understand the material and write quality code that works. Additionally, while test driven development is rarely followed strictly, unit tests are still a popular way of ensuring code quality during the maintenance phase of development, so being exposed to unit tests and learning how to write them is a valuable experience for computer science students.

Of course, none of this is to imply that test driven development should not be used outside of the classroom. A development model may not always apply fully to every situation, but, as such, it need not always be fully applied. Some parts of a project might fit test driven development, and some might not. If the requirements of a class or method can be assumed to be immutable, then writing tests first might be a good idea.