

# Compilers Portfolio

CSCI 468 – Spring 2023

Cory Janes and Dillon Shaffer

# CSCI 468 - Compilers Capstone

Teamwork

Design Patterns

Compositional Patterns

Behavioral Patterns

1 Introduction

2 Expressions

1) Equality Expression

2) Comparison Expression

3) Additive Expression

4) Factor Expression

5) Unary Expression

6) Identifier Expression

7) Primary Expression

3 Statements

1) Print Statement

2) Assignment Statement

3) If Statement

4) For Statement

5) Function Definition Statement

6) Return Statement

4 Type System

Assignable

Non-Assignable

UML

Design Trade-Offs

Weak Type System

Hidden Reference Types

Test Driven Development

## Teamwork

The team I worked with for this project consisted of myself, Dillon Shaffer, and Cory Janes. We spent time writing documentation for each others languages and testing/fixing bugs in each others code throughout the development process. We did this by writing tests for each other and trying to find issues in each others compilers by writing programs.

## Design Patterns

Given that CatScript is a simple language, there are only a few design patterns that go into its development. Once you exclude guard patterns there are only two or three unique structures in the language.

## Compositional Patterns

- **Composite Pattern**

The composite pattern is used in the structure of Statements and Expressions. Both are base types that inherit a shared `ParseElement` type. `ParseElement`s for a tree-like structure that

The utilities of this pattern can be seen in the `verify`, `execute`, `compile`, and `transpile` methods.

- **Iterator Pattern**

While we don't really create an official iterator in the CatScript compiler, both the Tokenizer and Parser use iterator-like behavior.

- The tokenizer has a while loop checking that there are more characters in the input. This loop matches the next character or set of characters and creates a token based on it.
- The parser is similar to the tokenizer in that it has a while-loop checking that there are more tokens. This loop calls the `parse` method to generate AST nodes.

## Behavioral Patterns

- **Facade Pattern**

The `ByteCodeGenerator` class, used during bytecode generation, is a facade over the `org.objectweb.asm` library. We provide some abstractions to simplify the generation of functions and creation of operation codes.

- **Tree-Construction Parser**

The `CatscriptParser` class uses recursive consumption of tokens to form our Abstract-Syntax Tree (AST). This is simple and efficient method for creating tree-like structures from linear data structures. We see this in practice with the consumption of tokens (stored in a linked list) as they are moved into a tree of expressions.

## 1 Introduction

CatScript is static but weakly typed programming language that used a recursive descent parser to evaluate expressions, statements, primitive types and compiles those into JVM bytecode.

## 2 Expressions

Expression: any valid unit of code that resolves to a value. With a recursive descent parser, the expressions are evaluated in a specific order given below:

### 1) Equality Expression

- Implements `equal to` and `not equal to`.
- Operators are `==` and `!=`.

**Examples:**

```
int1 == int2
var == 25
val() != 25
```

## 2) Comparison Expression

- a. Implements `greater than`, `less than`, `greater than or equal to`, `less than or equal to`.
- b. Operators are `>`, `<`, `>=` and `<=`.

**Examples:**

```
int1 > int2
```

```
int2 < int3
```

```
int3 >= int4
```

```
int4 <= int5
```

## 3) Additive Expression

- a. Implements `addition` and `subtraction`.
- b. Operators are `+` and `-`.

**Examples:**

```
x + y
```

```
int2 - int3
```

## 4) Factor Expression

- a. Implements `multiplication` and `division`.
- b. Operators are `*` and `/`.

**Examples:**

```
int1 * int2
```

```
int2 / int3
```

## 5) Unary Expression

- a. Implements `not` and `negative`.
- b. Operator is `-` or `not`.

**Example:**

```
-5
```

## 6) Identifier Expression

Implements variables with a stored string. The value of the variable is located by using the symbol table in the program.

## 7) Primary Expression

Implements the basic expressions:

- i. Identifier
- ii. Integer literal
- iii. String literal

- iv. Boolean literal
- v. List literal
- vi. Null literal
- vii. Parentheses
- viii. Function call

## 3 Statements

Similar to the recursive descent of the expressions, the statements in CatScript have a specific order to change the program state:

### 1) Print Statement

- a. Called with the 'print' string, followed by an expression
- b. Returns value to standard output in CatScript

**Examples:**

```
print("Detroit Lions")  
print(detroitLions)
```

### 2) Assignment Statement

Assigns or alters the value of a defined variable.

**Examples:**

```
int97 = 2023  
goatRB = "Barry Sanders"
```

### 3) If Statement

A conditional statement that requires certain symbols in a particular order to be

Functional:

- i. Starts with the `if` keyword
- ii. Followed by a `(` symbol
- iii. Followed by an expression
- iv. Followed by a `)` symbol
- v. Followed by a `{` symbol
- vi. Followed by statement
- vii. Followed by a `}` symbol
- viii. Conditional `else` keyword can follow or `else if` as well

**Example:**

```
if (detroitLions == "Super Bowl Champs") {
    print("2024!");
} else {
    print("then 2025 for sure!");
}
```

## 4) For Statement

A statement that iterates through objects, it is similar to the if statement in that it requires certain symbols to be functional:

- i. Starts with the 'for' keyword
- ii. Followed by a '(' symbol
- iii. Followed by a variable
- iv. Followed by an expression
- v. Followed by a ')' symbol
- vi. Followed by a '{' symbol
- vii. Followed by statement
- viii. Followed by a '}' symbol

**Example:**

```
for (var in variables) {
    print(var);
}
```

## 5) Function Definition Statement

The function definition statement defines functions in CatScript. These functions can return a type or return nothing, a void function, when called with a valid function name elsewhere in the program. The format need for functionality:

- i. Starts with a valid function name
- ii. Followed by a '(' symbol
- iii. Followed by argument(s) (multiple separated by a comma)
- iv. Followed by a ')' symbol

**Example:**

```
function lionsWin(year: int): boolean {
    return false
}
```

## 6) Return Statement

The return statement in CatScript requires a valid function and is called inside of the function itself. The statement will return a value when the following requirements are met:

- i. Starts with a 'return' keyword
- ii. Followed by an CatScript type to be returned

**Example:**

```
fn double(value) {  
  value = value + value;  
  return value;  
}
```

## 4 Type System

The CatScript's type system is statically typed and the types are divided into two categories, assignable and non-assignable types:

### Assignable

#### 1. Integer

There is a single 32-bit signed integer in CatScript.

#### 2. Strings

CatScript strings behave similar to often used programming languages, allowing newline and quotation characters, `\n` and `\"` respectively.

#### 3. Booleans

CatScript has the traditional Boolean type, `true` and `false`

#### 4. Lists

CatScript provides component specifiable lists:

- `list` specifies a list of objects
- `list<type>` specifies a list of only `type` elements

#### 5. Elements

The CatScript language assigned all of the types to an object.

### Non-Assignable

#### 6. Null

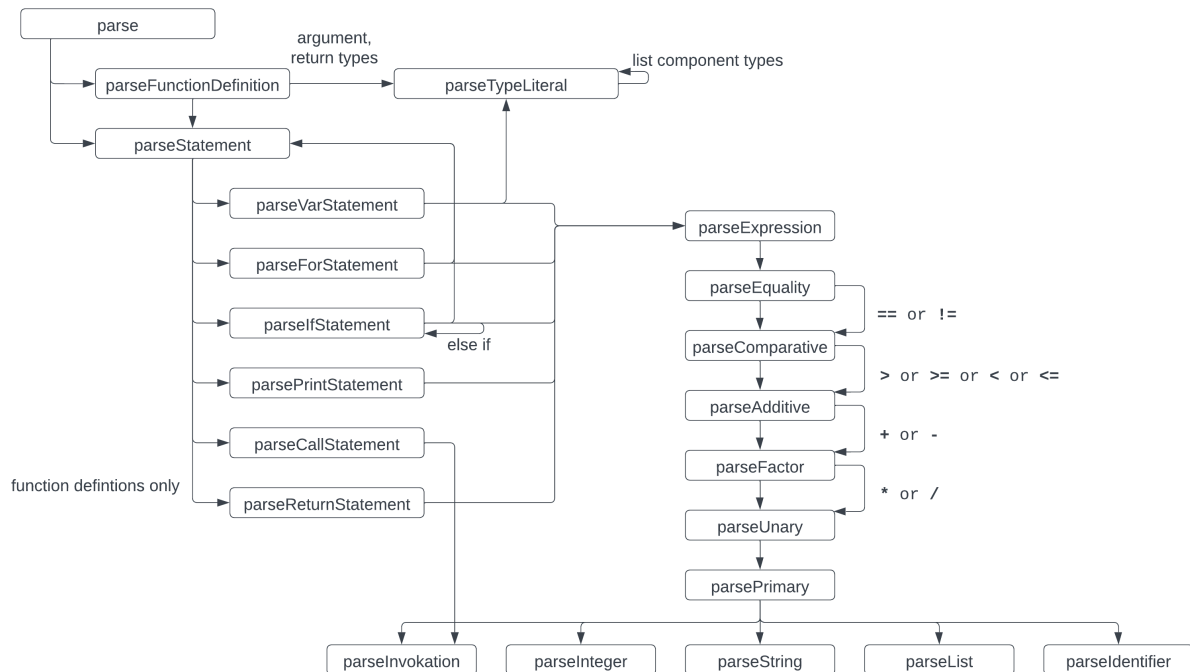
The `null` type in CatScript is non-assignable because it cannot be used in type declarations.

#### 7. Void

The void type is non-assignable because variables cannot be declared with the type void and visa-versa.

## UML

The most complicated part of the compiler is the parser. Given the complex grammar rules and embedded rules, it is not easy to just write a parser (unless you already know what you're doing). To combat this issue, we designed our parser before we wrote it 😊



## Design Trade-Offs

Although CatScript is technically a scripting language, it is not Turing complete. Given that this language was created in only three months, some corners had to be cut.

- There is not support for custom types (classes)
- There is no module/import system

## Weak Type System

CatScript has a weak type system, this makes it simpler to write code without more advanced language features and without an IDE. This comes with the upside of smaller code bundles but slower run-time.

## Hidden Reference Types

CatScript has reference types similar to Java, passing values under the `object` type boxes them and creates a reference. The primitive values `int` and `boolean` are not boxed until they are passed into



object parameters or variables. This has the side effect of using extra memory but easy data sharing.

## Test Driven Development

The CatScript compiler is developed solely based on testing. There is a series of tests for each major stage of the compiler. Our compiler has five stages—tokenizing, parsing, execution, compilation, and transpilation. Each stage has a set of tests that ensures that its respective stage is working properly. There are some edge cases that are not covered and there are some cases that are hard to test, but for the most part the compiler should work as expected.

We also added the following tests:

```
@Test
public void ifStatementWithElseIfParses() {
    IfStatement expr = parseStatement("if(x > 10){ print(x) } else if { print( 10 ) }", false);
    assertNotNull(expr);
    assertTrue(expr.getExpression() instanceof ComparisonExpression);
    assertEquals(1, expr.getTrueStatements().size());
    assertEquals(1, expr.getTrueStatements().size());
}

@Test
void assignmentTypeError() {
    assertEquals(ErrorType.INCOMPATIBLE_TYPES, getParseError("var x = 30\n" +
        "x = false"));
}

@Test
public void parseStringLiteralWorks() {
    StringLiteralExpression expr = parseExpression("\"Calvin Johnson\"");
    assertEquals("Calvin Johnson", expr.getValue());
}
```