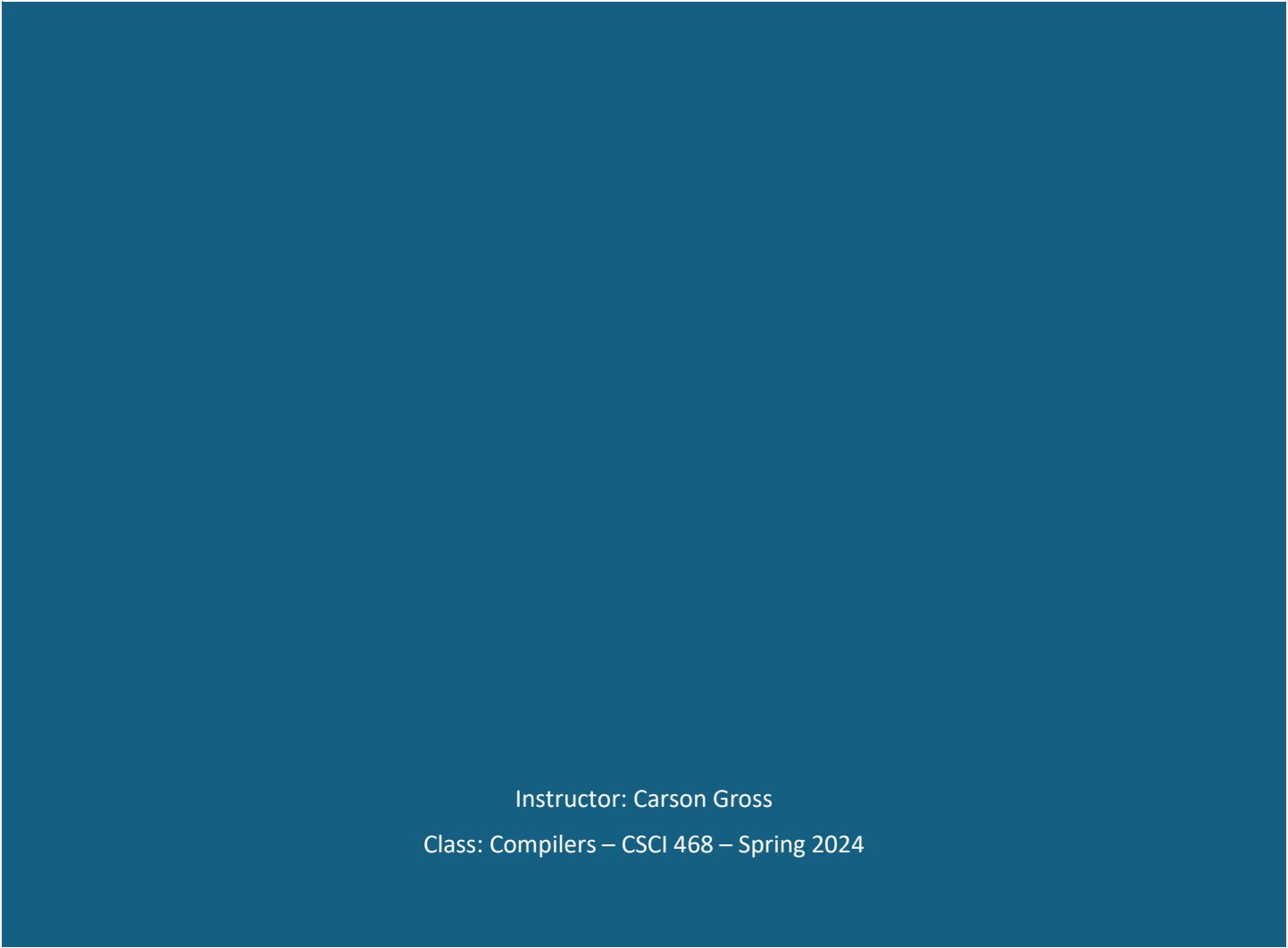




BRIAN SCHUMITZ SENIOR CAPSTONE DOCUMENT

Instructor: Carson Gross
Class: Compilers – CSCI 468 – Spring 2024



Section 1: Program

There is a ZIP file of my CatScript compiler code called source.zip in the /capstone/portfolio directory of my CSCI 468 GitHub repository.

Section 2: Teamwork

Team Member 1, Primary Contributions:

As team member 1, my main contribution to the project was writing most of the code for the CatScript compiler. Our professor, Carson Gross, provided us with the infrastructure for the project. However, we had to code the important parts of the compiler on our own. The main sections that we had to write a significant amount of code for were Tokenizing, Parsing, and generating ByteCode.

Tokenizing, also called scanning or lexical analysis, is the process of breaking text into individual tokens. This process acts in a loop, and typically involves discarding meaningless white space and comments. Parsing is the process of using a grammar to turn the stream of tokens into a parse tree based on a grammar. In our case, CatScript uses Extended Backus-Naur Form (EBNF) for its grammar. We took the recursive descent algorithm approach for our parser. Finally, for the back end of the compiler, we generated code in the form of JVM ByteCode.

My time spent implementing the code for the CatScript compiler amounted to around 95% of the project work.

Team Member 2, Primary Contributions:

Team member 2 mainly contributed to the project by writing documentation. The documentation came in the form of a CatScript guide. This guide, which can be seen in section four, acts as a user manual for the CatScript programming language. It provides brief descriptions of CatScript's type system, features, and expressions along with examples.

Additionally, team member 2 created test cases for my compiler. The tests that my partner wrote, seen below, provided valuable edge cases that tested the proper functionality of my parser. The tests pointed out issues in my code that I hadn't noticed prior, and I was able to get them fixed.

Team member 2's contributions, documentation and code tests, amounted to about 5% of the project work.

Compiler tests:

```
//This tests checks that variables assigned from additive or factor expressions
//are assigned/stored correctly
BrianS
@Test
void variableAssignedFromAdditiveAndFactorExpressionsWorks() {
    assertEquals( expected: "3\n2\n", executeProgram(
        src: "var x = 1\n" +
            "var y = 2\n" +
            "var z = x + y\n" +
            "print(z)\n" +
            "z = x * y\n" +
            "print(z)");
    }
}
```

```
//This test checks that additive expressions can be sent as arguments in a function call
BrianS
@Test
void functionCallWithAdditiveExpressionPassedAsArgumentWorks() {
    assertEquals( expected: "2\n3\n4\n", executeProgram(
        src: "function foo(x) { print(x) }\n" +
            "var y = 1\n" +
            "foo(y+1)\n" +
            "foo(y+2)\n" +
            "foo(y+3)");
    }
}
```

```
//This tests checks that for functions returning an int the return statement can have an additive expression.
BrianS
@Test
void returnStatementWithAdditiveExpressionWorksProperly() {
    assertEquals( expected: "80\n", executeProgram(
        src: "function foo() : int {\n" +
            "    var x = 42\n" +
            "    return x + 38\n" +
            "}\n" +
            "print( foo() )");
    }
}
```

Section 3: Design pattern

Memoization pattern in compiler code:

```
//Memoization pattern occurs here
3 usages
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
5 usages  ± BrianS +1
public static CatscriptType getListType(CatscriptType type) {
    if (cache.containsKey(type)) {
        return cache.get(type);
    }
    else {
        ListType listType = new ListType(type);
        cache.put(type, listType);
        return listType;
    }
}
```

Design patterns are general solutions for common problems in software development. Every design pattern can be thought of like a blueprint that can be customized and implemented to solve design problems in your code. Patterns are also useful tools because they provide a common language that helps teams communicate more efficiently and effectively.

In the development of the CatScript compiler, we implemented the memoization pattern, seen above. The memoization pattern addresses scenarios where a method is frequently called with identical inputs that always result in the same outputs. To reduce inefficiency, we can store given inputs and their associated output into a data structure like a hash map. By storing these inputs-output pairs, we can check if the method has already been run on some given input. This way, the method is only ever executed once for some unique input.

Specific to the CatScript compiler, we used the memoization pattern to optimize type access. Without the design pattern, each time the `getListType` method is called with the same type, we are unnecessarily creating duplicate `listType` objects. However, by using the memoization pattern, we were able to remove this inefficiency. Now, each time the `getListType` method is called, we check a cache to see if we already have a `listType` object associated with the type given as an argument. If that association already exists in the cache, then we will return the existing `listType` object. If that association does not exist, then we will create a new `listType` object, add the new association into our cache, and return the new `listType` object.

Section 4: Technical writing

CatScript Guide

Introduction

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

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

----[OUTPUT]----
foo
```

Type System

CatScript is statically typed, with a small type system as follows

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of value with the type 'x'
- null - the null type
- object - any type of value

Features

For loop statement

Description:

The user provides an expression with an unused variable and a list. Inside the For Loop, the user will put statements to be executed by the For Loop. The For Loop will iterate through the list and execute all statements inside the loop for each item within the list in the provided expression.

This allows for items in a list to be iterated through, so a range of numbers can be selected for statements to be executed a certain number of times. A list of items the user wants to use in statements can also be selected for processes.

Example:

```
for (var i in [1, 3, 5]) {
    print(i)
}
```

----[OUTPUT]----

```
1
3
5
```

If statement

Description:

The user provides an equality or comparison expression for the If Statement to verify. Inside the If Statement, the user will provide statements to be executed by the If Statement. The If Statement will first check if the expression provided yields true or false. If the expression is true, the If Statement will execute all statements within the If Statement. If it does not return true, the If Statement's body statements will be skipped over.

Optional: The user may provide an Else If or Else Statement. Else If Statements are similar to If Statements, but provide a separate expression for the statement to verify. Else Statements only require statements inside the Else Statement. If the If Statement's expression does not return true, the statements inside the Else If Statement that returns true, or, if no expression returns true and an Else Statement exists, the body statements of the Else Statement will be executed.

Example:

```
if (1 == 1) {
    print(true)
}
```

```
if (2 > 3) {
    print(true)
}
else {
    print(false)
}
```

----[OUTPUT]----

```
true
false
```

Print statement

Description:

The user provides an expression, variable, or value inside the Print Statement. When the Print Statement is executed, if an expression is in the Print Statement, the return value of the expression will be written for the user to read. If it's a variable, the value held by the variable will be written. If it's a value, the value will be written.

Example:

```
print("Hello, world!")
```

```
----[OUTPUT]----  
Hello, world!
```

Variable statement**Description:**

The user provides a string to be used as the variable's name and an expression to be used as the variable's value. This creates a variable in the memory of the program and the variable name cannot be used again.

Optional: The user may provide an explicit type. If an explicit type is given, the value provided by the user must match the explicit type. If no explicit type is given, the type will be inferred based on the value provided.

Example:

```
var number:int = 25  
var phrase = "This is a phrase"
```

Assignment statement**Description:**

These statements can be used to reassign values to already initialized variables. The statement only requires a new value to replace the variable's old value. The new value must match the type of the old value, regardless of if an explicit variable was used in the Variable Statement.

Example:

```
var x = "Original Value"  
x = "New Value"
```


Function definition statement

Description:

The user provides a name for the function followed by parentheses. Inside the parentheses, the user may provide zero or more strings to serve as parameters. These strings will be used as the variable names of the parameters, and the value will be determined by the Function Call Statement.

Optional: The user may give the function an explicit type. This is put after the parentheses and determines what type the function must return. Each parameter may have an explicit type as well. If the user does not provide a specific type, the type will be inferred.

Example:

```
function func1 (param1) {
    print(param1)
}

function func2 (param1 : string) {
    print(param1)
}

function func3 (param1 : string) : string {
    print(param1)
    return param1
}
```

Return statement

Description:

The user can use this to have a function return a value for use outside of the function. The CatScript type of the value being returned must match the explicit type of the function if one is given.

Example:

```
function concatenateTwoStrings (str1 : string, str2 : string) : string {
    concatenatedString = str1 + str2
    return concatenatedString
}
```

Function call statement

Description:

Function Call Statements are used to invoke functions that have already been defined with a Function Declaration Statement. The user will call the function with the same number of arguments as parameters the function is expecting. If the function has a return value, the value can be saved to a variable when the Function Call Statement is used.

Example:

```
var x : string = "Hello "
var y : string = "there!"
fullString = concatenateTwoStrings(x, y)
print(fullString)
```

```
----[OUTPUT]----
Hello there!
```

Expressions

Equality expression

Description:

Checks the equality of two values using the equals ("==") operator or the not equals ("!=") operator. Returns true or false.

Example:

```
print( true == true )
print( "hello" == "world")
print( 3 != 4 )
print( "hello" != "hello")
```

```
----[OUTPUT]----
true
false
true
false
```

Comparison expression

Description:

Compares two values using the less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). Returns true or false. May only be used on int type.

Example:

```

print( 1 < 2 )
print( 1 > 2 )
print( 1 <= 1 )
print( 2 >= 2 )
print( 1 <= 2 )
print( 1 >= 2 )

----[OUTPUT]----
true
false
true
true
true
false

```

Additive expression**Description:**

When using the plus (+) operator, if both values are integers, the operator will add the integers together. If both values are strings, the operator will concatenate the strings. The minus (-) operator only handles subtraction of two integers.

Example:

```

print( 1+1 )
print( "two" + "strings" )
print( 1-1 )

----[OUTPUT]----
2
twostrings
0

```

Factor expression**Description:**

The star (*) operator handles multiplication of two integers. The slash (/) operator handles division of two integers.

Example:

```

print( 2*3 )
print( 6/3 )

```

```
----[OUTPUT]----
6
2
```

Unary expression

Description:

Unary expressions handles negative integer values by using the minus (-) operator and handles logical negating of boolean values by using the “not” keyword.

Example:

```
print( not true )
print( -1 )

----[OUTPUT]----
false
-1
```

List literal expression

Description:

List literal expressions are used for creating lists of a certain type, and are immutable in CatScript. List expressions can be assigned to variables, or they can be declared inline in for-loop statements. Lists can be mutli-dimensional.

Example:

```
var list1 = [23, 52, 11]
var list2 = ["these", "are", "strings"]
var list3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for (var i in [1, 2, 3]) {
    print(i)
}

----[OUTPUT]----
1
2
3
```


Section 6: Design trade-offs

During our development of the CatScript compiler, a major design decision was whether to implement parsing using recursive descent or a parser generator, as these represent the two primary approaches to building parsers. Recursive descent is a method that works exceptionally well for Extended Backus-Naur Form (EBNF) grammars as the algorithm mirrors the recursive nature of such grammars. At runtime, the recursive descent algorithm traverses the parse tree recursively, which results in the efficient parsing of complex statements and expressions. In contrast, parser generators are tools that automate parser code generation based on a provided grammar. Parser generators are nice in that they offer convenience, but they often result in complex, less readable output.

Each method has distinct pros and cons. Recursive descent is favorable for its elegance, clarity, and readable code. However, its implementation can be time-consuming, and it may result in slower parsing performance compared to parser generators. On the other hand, parser generators are nice because they streamline the parser development process and produce highly optimized code. Unfortunately, they can also generate parser implementations that are difficult to comprehend.

Ultimately, we opted to implement recursive descent parsing for several reasons. This approach aligns seamlessly with EBNF-style grammars, which are inherently recursive. The resulting parser code is logical, straightforward, and easier to grasp compared to code generated by parser generators. Lastly, recursive descent is widely employed across industry for compiler development, further highlighting its practicality and effectiveness.

Section 7: Software development life cycle model

The Software Development Lifecycle (SDLC) is the process that development teams follow in order to design and build software products. Common steps that SDLCs include are planning, designing, implementing, testing, deploying, and maintaining. There are many SDLC models such as Waterfall, DevOps, and Agile. Every SDLC model has its pros and cons, and each model takes a slightly different approach from the other. Another SDLC model is Test Driven Development, which is what we used for this project.

Test Driven Development (TDD) is a software development practice where the focus is placed on designing unit tests *before* implementing any code. The scope of the TDD lifecycle is different from other SDLC models in that it tests code in small chunks rather than running tests against software systems as a whole. TDD aims to uncover bugs in code as early as possible, which makes the process of debugging and fixing them significantly easier. The TDD process is also iterative. Code is developed, tested, and refined in small chunks until all unit tests are passed. This decreases the likelihood of redundant code and results in more resilient software systems.

My experience developing the compiler using a TDD approach was the most enjoyable out of any SDLC I have had experience with. The preexisting test requirements that we needed to meet provided clear and concise development goals. The goals also translated perfectly into the process of writing accurate, non-redundant code throughout the compiler. Lastly, having concrete development goals to aim for, based upon the knowledge we gained throughout the semester, not only made the process clearer, but it was also quite enjoyable.