# CSCI 468 - Compilers

## Spring 2024


## Brian Schumitz

## Gage Nesbit

## Sam Roelofs

## Section 1: Program

The code is contained within the source.zip file in this directory.

## Section 2: Teamwork

Throughout the semester, each of us has built a compiler for the Catscript coding language. While the construction of this compiler has been done individually, the teamwork came in after the compiler had been created.

My team consisted of three members. The bulk of each member's project was made for themselves, however, one team member provided added material for other team members. Team member 1 provided tests and documentation of features of the language for team member 2. Team member 2 did the same for team member 3, and team member 3 did the same for team member 1 in turn.

This allowed for the team member receiving the other team member's work to verify if their compiler worked the way they wanted it to, as well as work with the team member providing their documentation to ensure the documentation provided is of the quality desired.

*The Tests I received:*

```
public class CapstoneTests extends CatscriptTestBase{
    Sam
    @Test
    void varAssignedFromAdditiveVarExpression() {
        assertEquals( expected: 2, evaluateExpression( src: "-(-1)+1"));
        assertEquals( expected: 0, evaluateExpression( src: "-(-1)-1"));
        assertEquals( expected: 5, evaluateExpression( src: "-(-(-1)*10)/-2"));
    }

    Sam
    @Test
    void functionDeclarationWithBooleanStatementWorksProperly() {
        assertEquals( expected: "[1, 2]\n", executeProgram( src: "function foo(x:list) { if(true){print(x)} }\n" +
                "foo([1,2])\n"));
        assertEquals( expected: "false\n", executeProgram( src: "function foo(x:list) { if(false){print(x)}else { print(false) }}\n" +
                "foo([1,2])\n"));
    }

    new *
    @Test
    void nestedIfStatementWorksProperly() {
        assertEquals( expected: "1\n", executeProgram( src: "var x=1 if(x>0){ print(1) }else {if(true){print(5)}}"));
        assertEquals( expected: "5\n", executeProgram( src: "if(false){ print(1) }else {if(true){print(5)}}"));
        assertEquals( expected: "6\n", executeProgram( src: "if(false){ print(1) }else {if(false){print(5)}else{print(6)}}"));
    }

}
```

## Section 3: Design Pattern

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
5 usages   Carson Gross *
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;
    }
}
```

The above method has been written using the memoization design pattern. This was used to cache types used in lists for future lists. Using this, instead of creating a new object to be assigned as the type for the list, the compiler checks if the list type desired has already been used in a list and uses that object. This increases efficiency when creating lists.

# Section 4: Technical Writing

**Features:**

*Catscript Types:*

object: The Catscript object type which may hold any values of the types below.

int: The Catscript integer type.

string: The Catscript string type.

bool: The Catscript boolean type

List: The Catscript list type is able to hold values of the type of anyone and only one of the above. An important note is that within Catscript lists are immutable.

**Variable Statement:**

A way to initialize variables a value within Catscript. Variables must be given a value upon initialization. This is done such that a variable can never be null. This reduces the risk of null pointer exceptions.

Variable statements are not required to be given an explicit type as one can be inferred by looking at the type of the value on the right of the equals, this type will become the type of the variable. You can however give an explicit type. It is important to note that the type of the value that is being assigned to the variable must match the type of the variable.

Ex:(Inferring the type of int)

```
Var x =1
```

Ex:

```
Var x : int=1
```

Ex:

```
Var x : string= “hello world”
```

**Assignment Statement:**

A way to assign a new value to a variable within Catscript. The new value to be assigned must be assignable to the type of the variable. For more information on which types are assignable to which types, you can look above for documentation on Catscript types. A complex expression can be used as the assignment value as this will be interpreted before the value is assigned to the variable.

The validation of types within the assignment statement will occur after the parsing. This validation is not only critical to ensure type issues don't arise but also validates that the variable is a valid symbol name and has been initialized.

Ex:

```
x= 1
```

Ex:

```
x = “hello world”
```

**Print Statement:**

This is the only way to interact with the output buffer of Catscript. As the only form of I/O care was taken to make sure that the behavior and performance were baked right into the language. By giving the print statement we were able to have maximum control over print.

The print statement does perform and looks like a function though under the hood its behavior is fixed. A user cannot create a function or variable called print, it is a reserve word for the Catscript runtime. Doing either will cause a parsing error resulting in a nonfunctional program.

Ex:

```
print(1)
```

Output:

```
1
```

**For Loops:**

Catscript for loops are an easy-to-understand iterative loop. Unlike the C-style loops in Catscript, you cannot have an incremental value that loops over a specified range. As one of Catscript's control flow statements the for loop gives programmers adds substantial capabilities.

All statements held within the body will be executed during each iteration of the for loop as determined by the interactive loop variable. The for loop body constitutes all statements held between the two curly braces. In Catscript a for loop can hold any other type of statement except a function definition statement.

Ex:

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

Output:

```
1
2
3
```

**If statement:**

The if statement is another form of control flow statement. Though this unlike a loop or function invocation is a conditional control flow. The if statement takes a boolean value as its expression. Once the expression is evaluated if the expression is true the “true statements” within the if will be executed. If the expression is evaluated to be false then the statements within the else will be executed if one is present.

An if statement within Catscript may have an else statement, though one is not required. Unlike other languages, the if statement only has capabilities for one if and one else. To simulate an else-if case another if statement can be placed in the else, where that if statement can be thought of like an else-if.

Ex:

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

Ex:

```
if(x==true){
        print(true)
}
else{
        print(false)
}
```

**Function definition statement:**

This is how functions are defined within Catscript. Much like with the variable statement functions can either have their type inferred or they can have explicit type given. Functions can be just like a variable statement of the type object, string, int, or bool, they are also able to be of the void type where they do not need to return any value.

Additionally, they have to be given zero or more parameters. These parameters can have inferred types or explicit types, you can even mix and match where some parameters have an explicitly given type and somewhere a type will be inferred. The body of a function is defined as all the statements bound by the curly braces.

Ex:

```
function foo(){
    print(“foo”)
}
```

Ex:

```
function foo( bar) : bool {
    print(bar)
    return true
}
```

Ex:

```
function foo( bar : string ) : bool {
    print(bar)
    return true
}
```

**Function call statement:**

This is how functions are invoked within Catscript. This is the third and most sophisticated version of control flow. Within a function call, a new scope is pushed onto the symbol table. To invoke a function you need to give the name of the function followed by a parenthesis and any arguments then the closing parenthesis.

It is critical to note that you must send the proper amount of arguments to the function or it will cause an error. Additionally, the arguments that are sent must be of the type specified in the function definition statement if a specified type was given. If a function returns an expression a function invocation can be nested with any other statement as per the grammar and treated like any other expression.

Ex:

```
foo()
```

Ex:

```
foo(1, “string”)
```

**Equality expressions:**

How two expressions of equality are evaluated. There is an equals operator “==” and a not equals operator “!=”. These two operators will return true or false. Within Catscript this can be used with all types.

Ex:

```
print(   2 == 2  )
print(  false == “string”  )
```

Output:

```
true
false
```

Ex:

```
print(   2 != 2  )

print(  false != “string”  )
```

Output:

```
false

true
```

**Comparison expression:**

This is how values are compared within Catscript. There is a greater than operator “>”, a greater than or equal operator “>=”, a less than operator “<”, and a less than or equal to operator “<=”. Comparison expressions can only be used with the int type.

Ex:

Ex:

```
print(   2 >1 )

print(   1>2  )
```

Output:

```
true

false
```

Ex:

Ex:

```
print(  2 >=2 )

print(  1>=2  )
```

Output:

```
true

false
```

Ex:

```
print(   2 <1 )

print(   1<2  )
```

Output:

```
false

true
```

Ex:

```
Ex:

print(  2 <=2 )

print(  2<=1  )
```

Output:

```
true

false
```

**Additive expression:**

Within Catscript the “+” operator has two functions. To add integers together as well as concatenate strings. The types of each operand must match or there will be an error. The “-” operator deals with subtraction between two integers.

Ex:

```
print(1+2)

print(“hello”+”world”)

print(3-2)
```

Output:

```
3

helloworld

1
```

**Factor expression:**

This expression is responsible for multiplication with the “*” operator and division with the “/” operator between two integers.

Ex:

```
print(2*3)

print(6/2)
```

Output:

```
6

3
```

**Unary expression:**

This expression is responsible for inverting boolean values with the “not” reserve word. To negate an integer value the “-” operator is used.

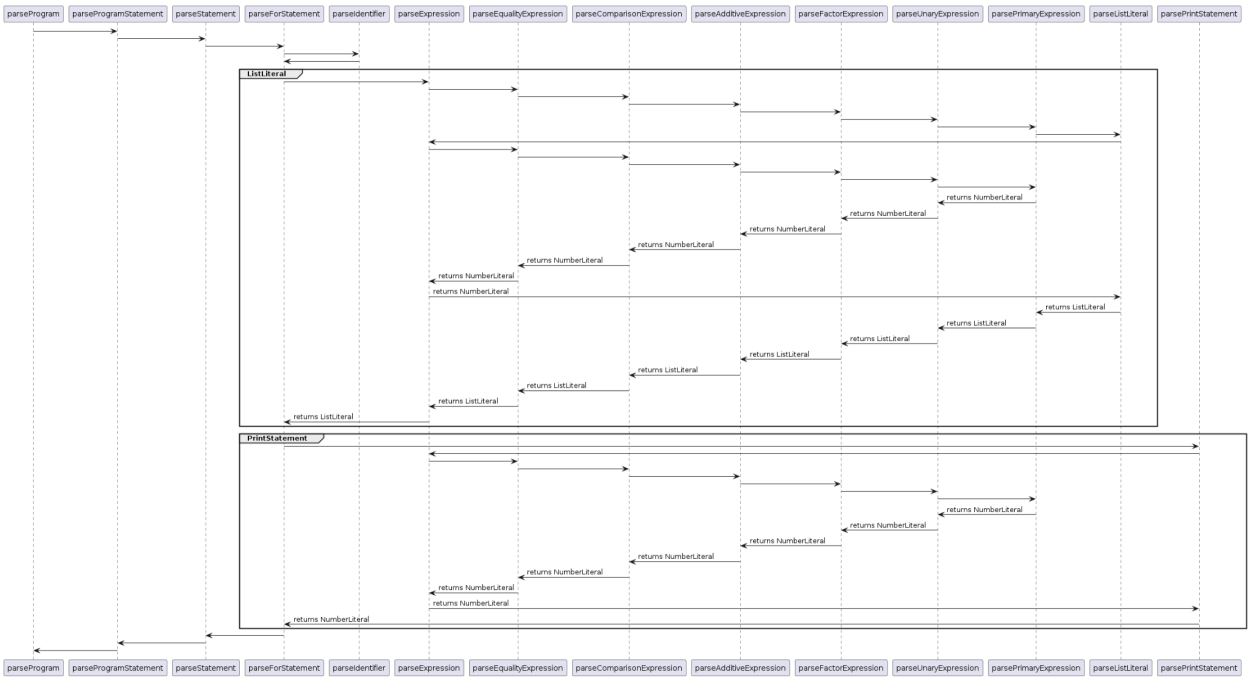
Ex:

```
print(not false)

print(- (1)  )
```

Output:

```
true

-1
```

# Section 5: UML




Above is a sequence diagram showing how the compiler would parse:

```
for(x in [1,2]) {
        print(x)
}
```

## Section 6: Design Trade-Offs

The parser of this compiler was made using recursive descent parsing. This is a type of top-down parsing. We used this instead of the more commonly taught parser generator approach. A parser generator is given a grammar and creates a parser to check if a sequence of characters fits in the grammar supplied

Generator parsers involve a lot less written code than recursive descent, and therefore is a lot easier to get right. However, parser generators come with their own downsides. The syntax used by parser generators can be obscure and harder to understand than if the parser was written by hand. Furthermore, recursive descent ties well into the recursive nature of grammars.

Other types of parser designs are used, but parser generators and recursive descent were the most prominent for the course, as we used recursive descent in the course and parser generators are commonly taught in other universities.

## Section 7: Software Development Life Cycle Model

This project was done using Test Driven Development. In this case, while writing each part of the compiler, we had associated tests to ensure each bit of the part we were working on functioned properly. For example, during the tokenizer, we had tests to check if the tokens were being split correctly.

This model was very helpful, as it ensured the parts of the compiler worked correctly. This prevented issues later on where a previous section wasn't properly written, causing confusion and errors in different sections relying on previously poorly written code.

This model also helped with navigating what would otherwise be a very daunting project. The tests guided what specifically needed to be done for each part, which made it easier to remember each section of the code that needed to be written. While the tests are usually written after the code rather than before it, this was something that stuck and was worth mentioning.