# CSCI 468 Compilers Capstone

Jacob Hargiss

**Section 1:** the src.zip file can be found at : https://github.com/HAjacob/csci-468-spring2024/blob/main/capstone/src.zip

**Section 2:** this is the unfortunate part. I got very busy with getting everything done for all my classes for the past couple months and forgot I needed to find a partner until this last week before finals, at which point it was too late. So, instead I just did my partners portion for my own compiler wherein I made 3 tests for my compiler located in "partner tests" and the documentation I am submitting is my own and not my partners. As the rest of this capstone course was independent, I am hoping you will not judge me too harshly for my oversight. I feel that my documentation is adequate for familiarizing a user with the basics of how Catscript works. I went over every expression and statement possible as well as provided copious examples on what works and what causes an error. As for my 3 tests, I focused on areas that the normal classwork tests didn't. my tests were mostly on testing incorrect type errors as well as combining multiple different statements into 1 more realistic function. What I found in doing this is that while my code is adequate for compiling single statements, it struggles to compile more complex compound statements especially where if-statements are involved. This is definitely something I would have liked to address if I had the time to make this a functional compiler for the function. The tests can be seen below:

```
public class PartnerTests extends CatscriptTestBase{
    @Test
    public void varStatementAssignedToWrongType() {
        //This Test is to determine if the type assignment to the variable results in correct error handling
        assertEquals(ErrorType.INCOMPATIBLE_TYPES, getParseError("var x : int = \"hello\""));
        assertEquals(ErrorType.INCOMPATIBLE_TYPES, getParseError("var x : string = 1"));
        assertEquals(ErrorType.INCOMPATIBLE_TYPES, getParseError("var x : bool = 1"));
        assertEquals(ErrorType.INCOMPATIBLE_TYPES, getParseError("var x : bool = \"true\""));
    }
    //This test verifies that variables work properly in higher and lower scopes
    @Test
    void returnStatementWorks() {
        assertEquals("1\n2\n3\n", executeProgram(
                "function foo(x : int){\n" +
                        "print(1)\n" +
                        "print(2)\n" +
                        "print(3)\n" +
                        "}\n" +
                        "foo(9)"
        ));
        assertEquals("11\n", executeProgram(
                "Var output : int = 0\n" +
                        "for(y in [1,2,3]){\n" +
                        "output = y + x\n" +
                        "}\n" +

                        "print(output)"
        ));
        assertEquals("11\n", executeProgram(
                "function foo(x : int) : int {\n" +
                        "Var output : int = 0\n" +
                        "for(y in [1,2,3]){\n" +
                        "output = y + x\n" +
                        "}\n" +
                        "return output \n" +
                        "}\n" +
                        "print(foo(9))"
        ));
    }
//this is a test of the ByteCode generation on more complicated expressions
    @Test
    void printStatementWorksProperly() {
        assertEquals("10\n", compile("print(1 + 2 + 7)"));

        assertEquals("false\n", compile(
                "Var x = true\n" +
                        "if(x == true){\n" +
                        "print(x)\n" +
                        "}"
        ));
    }
}
```

# CatScript Language Documentation

The CatScript language is a programming language designed to be similar in structure to JavaScript for use in the CSCI-468 Compilers course at Montana State University. CatScript is a bare bones Dynamically typed language, meaning the variable types are determined at runtime, not at compile time. This results in a programming language where types do not have to be specified in code. It is meant to strike a balance between having good readability and good writability. Due to It's role as an educational tool, it lacks many creature comforts of a normal programming language but allows students to have an easier time building the compiler for it.

# Features:

## Expressions:

### Additive Expressions:

The additive expression allows the user to program summation and subtraction operations between integer type expressions. As well as adding and subtracting integers, the additive expression can be used to concatenate string type expressions, including adding an integer to a string value

Examples:

```
print(1 + 10)
    Result: 11

    print("ham " + "sandwich")
    Result: "ham sandwich"

    print(1 + " more")
    Result:"1 more"
```

### Unary Expressions:

The Unary Expressions allow the user to get the inverse of a boolean type expression using "not" or the negative of an integer type expression using "-".

Examples:

```
print(-(1 + 2))
Result: -3

print(not(true))
Result: false
```

### Comparison Expressions:

Comparison expressions allow the user to compare 2 expressions of integer type with ">", ">=", "<", and "<=". These comparison instructions can only be applied to integer expressions.

Examples:

```
(1 > 2)
Evaluation: false

(1 < 2)
Evaluation: true

(1 > 1)
Evaluation: false

(1 < 1)
Evaluation: false

(1 >= 1)
Evaluation: true

(1 <= 1)
Evaluation: true
```

## Equality Expressions:

Comparison expressions allow the user to compare 2 expressions to check if they are equal. If they are of different types, then the expression will return false.

Examples:

```
(1 == 1)
Evaluation: true

(1 != 1)
Evaluation: false

(1 == 2)
Evaluation: false

(1 != 2)
Evaluation: true

("1" == 1)
Evaluation: false

("1" ! 1)
Evaluation: true

("1" == "1")
Evaluation: true

("ham" == "sandwhich")
Evaluation: false

(true == true)
Evaluation: true

(true == false)
Evaluation: false

(true != false)
Evaluation: true

(1 == null)
Evaluation: false

(1 != null)
Evaluation: true

(null == null)
Evaluation: true
```

## Factor Expressions:

The Factor Expressions allow the user to multiply and divide 2 integer type expressions. If the left hand side expression is not the same type as the right hand side or they are not of type integer, it will result in an Incompatible Type Error

Examples:

```
print(1 * 2)
Result: 2

print(10 / 2)
Result: 5

print(10 * "baked beans")
Result: Incompatible Type Error
```

## Parenthesized Expressions:

Parenthesized expressions allow the user to define order of operations of a complex expression. This enables more complex mathematical equations and other operations to be preformed correctly without regard to the default evaluation scheme.

Examples:

```
(2 * 2 - 2)
Result: 2

(2 * (2 - 2))
Result: 0
```

## Identifier Expressions:

Identifier Expressions define the type of expression,

Examples:

```
int:
        Denotes a variable as being of type integer

string:
        Denotes a variable as being of type string

bool:
        Denotes a variable as being of type boolean

object:
        Denotes a variable as being a generic type
        Any type can be passed to a variable of type object

null:
        Denotes a variable has a missing or unassigned value

void:
        Denotes a function has no type so no return value be expected
```

## Type Literal Expressions:

These expressions exist to define the type of a function or variable. They include "int", "bool", "list", "null", "object", and "string'

Examples:

```
Var x : int
Result: x can contain any valid integer value

Var x : string
Result: x can contain any valid string value

Var x : bool
Result: x can contain "true" or "false"

Var x : null
Result: x has no type. When a value is assigned, the variable type will be set.

lists:
        used with an implicit type or an explicit type
        Examples:
                Implicit type:
                Var x = [1, 2, 3, 4]
                        Type: list<int>
                        Results: x[0] : 1
                                         x[3] : 4

                Explicit type:
                Var x : list = ["apples", "oranges", "grapes"]
                        Type: list<string>
                        Results: x[0] : "apples"
                                         x[2] : "grapes"

                Var x : list<string> = ["apples", "oranges", "grapes"]
                        Type: list<string>
                        Results: x[0] : "apples"
                                         x[2] : "grapes"

                Explanation:    When defining the explicit type of list, the type of items held
                                       in the list can also be specified also be specified by including
                                       a type litteral expression between '<' and '>' characters in the
                                       form of: list<'type literal'>
```

# Statements:

## Variable Definitions:

This statement allows the user to define a new variable of a given name with or without an explicit type reference and assign. The user is able to assign a value to it at time of definition or leave it unassigned.

Examples:

```
Var x : int = 1
Evaluation: new variable named "x" is set with a type of "int" and the value of 1

Var x = 1
Evaluation: new variable named "x" is set with a value of 1 and its type is
                set to the type of that litteral expression used, in this case "int"

Var x : string
Evaluation: new variable named "x" of type "string" with no set value

Var x
Evaluation: new variable named "x" with no set type and no set value
```

## Variable Assignment:

This statement allows the user to set or change the value of a variable already defined. If the type of the variable is already set then an error will be triggered if the user tries to set a different type of value to the variable. This statement does not implicitly define a new variable if one does not exist with the given name

Example:

```
Var x : int
x = 10
Evaluation: assigns the value of 10 to variable named "x"
```

## For Loop:

This statement allows for iteration through a list. The user defines a variable used to hold the current element of the list, starting at element 0. The list then operates on every statement in it's body until it reaches the last element at which point, it does 1 final execution of the body statements, then exits the loop

Example:

```
For ( x in ["Apples, "Oranges", "Bannanas"]){
        print(x)
}
Result: "Apples"
                "Oranges"
                "Bannanas"
```

## If Statements:

These statements allow for the user to execute specified code if some boolean expression is true. This type of statement only supports boolean expressions, if any expression is used that does not evaluate to a boolean type an error is triggered.

Example:

```
print("first")
if ( true == false ){
        print(true)
{
print("last")
Result: "first"
                "last"

print("first")
if ( true != false ){
        print(true)
{
print("last")
Result: "first"
                "true"
                "last"
```

## Print:

The print statement allows the user to specify a specific expression to be outputted to the console. This works with expressions of all types including function calls

Example:

```
print ( 100 )
Result: "100"

print ( "apples" )
Result: "apples"

print ( [1, 2, 3] )
Result: "[1,2,3]"
```

## Function Definitions:

This statement defines a function that can be called later by a function call. The function definition can include input variables with implicit or explicit types. The function definition can also declare a specific return type

Examples:

```
function foo(){
        print(1)
        print(2)
}
Result: Creates a functioned named "foo" that takes in no variables and executes all
                statements withing the function declaration when calle.d

function foo(x){
        print(x)
}
foo(7)
foo("hello")
Result: Creates a functioned named "foo" that takes in a single cariable named x of
                generic type and executes all statements withing the function declaration when
                called.
                Output:
                        "7"
                        "hello"
Explanation: Because x is a generic type, it will be defined at runtime, so when foo(7)
                         is called, that instance of the function defines x as type int and when
                         foo("hello") is called that instance of the function defines x as type
                         string.

function foo(x : int){
        print(x)
}
foo(7)
foo("hello")
Result: Creates a functioned named "foo" that takes in a single variable named x of
                generic type and executes all statements withing the function declaration when
                called.
                Output:
                        "7"
                        Error: Incompatible Types
Explanation: Because x is specified as being of type int, it is expecting the value
                         taken in from the function call to be of type int, thus this creates an
                         error since strings cannot be implicitly parsed to integers in catscript.

function foo(a, b, c){
        print(a)
        print(b)
        print(c)
}
foo(1, 2, "apples)
Result: Creates a functioned named "foo" that takes in a single cariable named x of
                generic type and executes all statements withing the function declaration when
                called.
                Output:
                        "1"
                        "2"
                        "apples"
Explanation: The function definition statements can declare multiple input values.

function foo() : void {
        print(1)
}
Result: Creates a functioned named "foo" with the return type of void that takes in no
                variables and executes all statements withing the function declaration when
                called. Since the return type is void, there cannot be any retrun statements in
```

```
                the function.

    function foo() : int {
            return 1
    }
    Result: Creates a functioned named "foo" with the return type of int that takes in no
                    variables and executes all statements withing the function declaration when
                    called. Since the return type is int, there must be a retrun statements in the
                    function that returns a value of type int, otherwise there will be an
                    Incompatible types error
```

## Return:

This statement is used to send the resulting value of a function to the function call that triggered it to run. The return type can be explicitly specified at function definition. An error will result if the value being returned does not match the type of the function.

Example:

```
    function foo(){
            return 10
    }
    print( foo() )
    print( foo() - 1 )
    Result:
            "10"
            "9"

    function foo() : string{
            return 10
    }
    print( foo() )
    Result:
            Error: Incompatible Types
```

# Function Calls:

Function calls can behave as both statements and expressions. when used on their own, they are treated as statements, when used in conjunction with other expressions they are treated like expressions. When used like "foo()" on their own they are statements, when used like "Var x = foo()" they are handled as expressions.

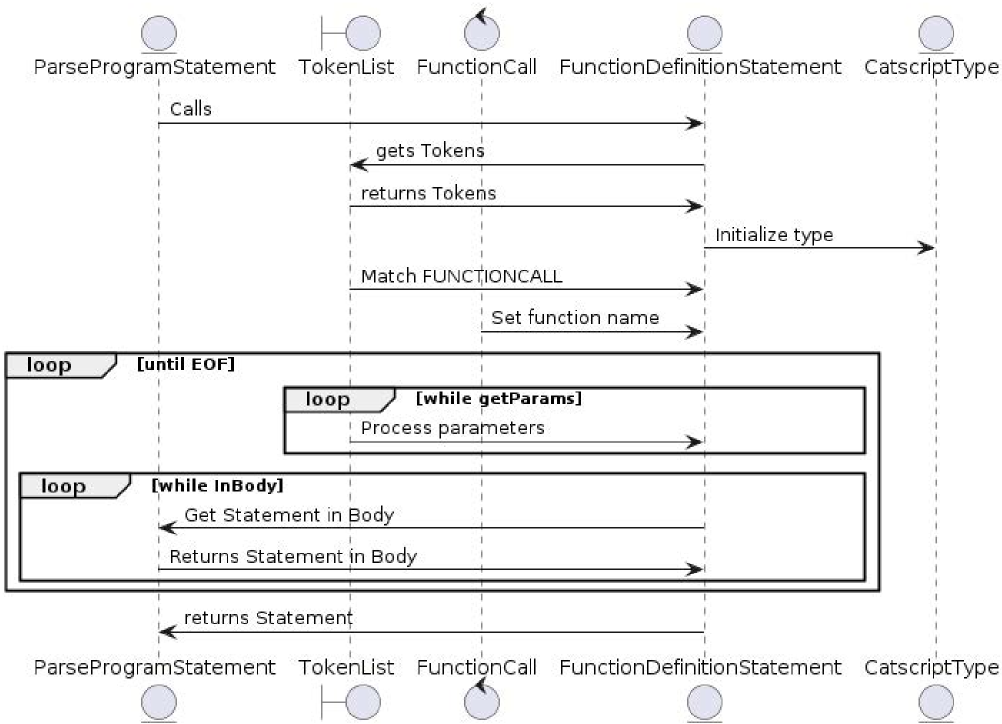
Examples:

```
foo()
    Result: executes function "foo" with no input

    foo(7)
    Result: executes function "foo" with the input of integer 7

    foo([1, 2, 3])
    Result: executes function "foo" with the input of list [1, 2, 3]

    foo(1, 2, 3)
    Result: executes function "foo" with the inputs of 1, 2, and 3 coresponding to 3
                    different input variables of the function
    Var x = foo()
    Result: executes function "foo" with no input and assigns the value of what ever the
                    function returns to variable x
```

Section3: For this part, we were instructed to all use the same design pattern, that being memoization. memoization works by storing variables already created so they dont have to be created again when needed. This saves on system resources and compute time. It is definitely a better method than just duplicating variables every time we call a function. when implementing this feature, I accomplished this by establishing a hashmap, then if the variable didn't exist already, I would create one and add it to the hash-map. Otherwise my function would just call a get operation on the hash-map to get the variable I need.

Section 4: Not Applicable

Section 5:




Here I have the UML Diagram for my function definition statement parser function. It works by first being called by the program statement, then it checkes the tokenList for a "function" identifier and for a functionCall token (I added this to the tokenizer to make things easier), if it sees one, then it goes on to evaluate the tokenlist as containing a function definition. If it doesnt see one, it checks for every other statement possible. at the end of this recursion chain, the parseFunctionDefinition method returns its statement to the parseProgramStatement function.

Section 6: for this Course, we were given the design for a Recursive-Descent parser. However, a parser generator could have also been used for this same task. Parser Generators work as a tool that analyzes rules for a language model using a context free grammer and uses that to parse out the code. While parser generators are an easier to use method, Recursive-descent offers the ability to look ahead into the list of tokens while parsing. As well, Recursive descent also offers the ability to alter the parsing at the base level to tweak how different sequences are interpretted. parser generators will not do this, so any small changes require changing the grammer of the language. This makes Recursive Descent more flexible in it's aproach to parsing

Section 7: For this class, we used Test Driven Design. In this aproach, tests are formed to put the program through it's passes to make sure it can pass a base level of functionality. We used sepcifically, the test first variety of this methodology. This means that the first step was to build code that passed the tests. I am not a fan of this design method. It is too easy to pass tests based on a fluke result or miss a neccisarry part of the functionality and end up building the rest of the code on a bad foundation. This reults in alot of having to go back and make repairs on code that is not compatible with the next phase of tests. As an Example, during our last round of tests, I had 3 tests that I couldnt get to work. my code that compiled the bytecode was correct, but 2 steps earlier I had an oversight in my parser that caused parse errors during these tests. As a result, to fix the problem I would have had to almost entirely rebuilt the code that parses my function definitions. In my opinion, there are better ways to start the process of building a new piece of software. For example, I think a better approach would be to start with a UML Diagram and build a foundation on that structure before integrating testing more near the end.