

## Program

source code:

<https://github.com/Gearhartlove/CatScript-Compiler/tree/master/capstone/source.zip>

## Teamwork

Team Member One and I worked together in providing documentation for each other's codebase as well as implementing the logical operators '`||`' and '`&&`' into CatScript through four different tests. Five percent of the project's time was spent working with Team Member One, as most of the compiler work involved writing code and passing test checkpoints individually.

Team Member One's documentation is a CatScript guide. It details typing, operators, storing and printing values, branching, looping, variables, and function specifics. Each piece of information is followed by in-depth examples written in CatScript. The documentation is included in the Technical Writing portion of this document.

The logical operators '`||`' and '`&&`' are implemented for CatScript. This changes the grammar by **factor\_expression** recursively requiring a **logical\_expression**, and moving the **unary\_expression** which was present into the **logical\_expression**. This fits into the grammar as follows.

```
factor_expression = logical_expression { ("/" | "*" )
                                logical_expression };
```

```
logical_expression = unary_expression { ("&&" | "||" )
                                unary_expression };
```

The tests written for this feature include tests for tokenization, expression parsing, evaluation, and ByteCode compilation. The tests written in the code are included on the next page.

## Tokenization

```
@Test
public void logicalExpressionTokenization() {
    assertTokensAre("true && false", TRUE, LAND, FALSE, EOF);
    assertTokensAre("true || false", TRUE, LOR, FALSE, EOF);
}
```

## Parsing

```
@Test
public void logicalExpressionWorks() {
    LogicalExpression expr = parseExpression("true && false");
    assertTrue(expr.isAnd());
    expr = parseExpression("true || false");
    assertFalse(expr.isAnd());
}
```

## Evaluation

```
@Test
void logicalExpressionEvaluatesProperly() {
    assertEquals(true, evaluateExpression("true && true"));
    assertEquals(false, evaluateExpression("false && true"));
    assertEquals(true, evaluateExpression("true || false"));
    assertEquals(false, evaluateExpression("false || false"));
}
```

## Compilation

```
@Test
void logicalExpressionCompilesProperly() {
    assertEquals("true\n", compile("true && true"));
    assertEquals("false\n", compile("true && false"));
    assertEquals("true\n", compile("false || true"));
    assertEquals("false\n", compile("false || false"));
}
```

After the grammar was updated, after the tests were written, and after every error was handled, the compiler now compiles catscript with logical operators! An example of the feature implemented in unison is shown by the following valid catscript.

```
var x : bool = true || false // evaluates to true
```

## Design Pattern

CatScript uses the memoization pattern to memoize type access in the `getListType()` method in the `CatscriptType.java` code. This pattern "accelerates performance by caching the return values of expensive function calls" ([cloudsavvit.com](https://cloudsavvit.com)). A `HashMap` is created which stores the catscript type. If the queried `CatscriptType` exists in the `HashMap`, the cached result is returned. If the queried `CatscriptType` does not yet exist, then a new `CatscriptType` is instantiated and added to the `HashMap`. This removes redundant instances of `CatscriptType` `ListTypes` existing, when they are already instantiated.

The source code for this feature found in `CatScriptType.java` is below.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);
    if (listType == null) {
        listType = new ListType(type);
        cache.put(type, listType);
    }
    return new ListType(type);
}
```

# Catscript Guide

---

## Introduction

---

Welcome to the Catscript Guide! Catscript is a simple scripting language that statically is typed and statically scoped. It supports strings, integers, lists and booleans. And, it is *technically* turing complete via the `if` statement and the `for` loop!

This language was completed as an assignment for CSCI 468: Compilers at Montana State University. This document is to help explain how to use Catscript for whatever you'd like to do in the JVM.

## Features

---

### CatScript Typing

---

#### Booleans

Catscript supports basic booleans. We can get them in a few ways

```
var x : bool = true           // via direct assignment
var y : bool = 1 < 2          // or doing some calculations
var z : bool = null == null  // note: null == null!
```

We can also chain boolean expressions to get new booleans, in our version of catscript, we have a logical expression

```
var x : bool = true || false // evaluates to true!
```

#### Integers

Catscript supports integers.

```
var x : int = 0           // 0
var y : int = 1 + 1       // 2
var z : int = 1000 / 10    // note: Catscript does not support floating
                          // point, so it will always do integer division
```

#### Strings

The classic 'Hello World' program is possible in Catscript! Strings are backed up by the JVM's String class, and we can do concatenation through the `+` operator

```
print("Hello World!")      // sends 'Hello World!\n' to stdout
print("Hello, " + "Strings") // sends 'Hello, Strings\n' to stdout
```

#### Lists

Lists in Catscript are immutable, which means that we can store multiple types in the same list. However, be careful because trying to assign a list of `object` to a list of `int` will fail because we cannot guarantee that all items in the list are `int` types

```
var x : list<int> = [1, 2, 3, 4]
var y : list<string> = ["Hello", "World", "!"]
var z : list<object> = [null, "Hello", 1]
var w : list<int> = z // will fail because of the above
```

#### Null

Catscript supports null types.

```
var x : object = null
```

#### Object

Catscript supports a basic object type. Most of the time this is used in type inference but you can just declare all things to be of type `object`. A notable exception is you cannot assign `list` types to `object`

```
var x : object = "Strings!"
var y : object = false
var z : object = 1
```

## Void

Catscript also supports a void type, although it is not assignable to any variable. It is used for functions that do not return any values.

```
function foo() {
  print("This function doesn't return anything!")
}
```

## Type Inference

So far in this guide we have been telling the compiler exactly what type we are going to use for our variables i.e `var x : int = 0`. However, we can just let the compiler figure out whatever the variable's type is supposed to be by just saying `var x = 0`. The compiler will figure out what the type of the variable should be through *type inference*, by just checking what the type of the variable is on the right hand side of the expression. So, in the rest of this guide, we'll be taking advantage of some the features in the compiler to minimize the code we're writing.

## Variable Assignment Errors

Catscript will not let you assign dubious types to variables. For example

```
var x : string = "Hello World!"
x = 0    // <- will fail
```

Because `int` is not assignable to `string` But you can do:

```
var x : string = "Hello World!"
x = null
```

Because `null` is assignable to any type.

As a general guide, anything is assignable to `object`, `null` is assignable to any type, and each type is assignable to itself.

## Catscript Operations

Catscript supports very basic operations. We have addition, subtraction, division and multiplication. **Note** : Since Catscript does not support floating point types, all division operations will act like integer division.

### Adding

Addition is done via the `+` operator. As you saw in the `Catscript Typing` section, you can see that we can add two numbers like so:

```
var x = 1 + 1
```

Please note that the `+` operator is *overloaded*. If you were to do

```
var x = "1" + 1
print(x)
```

this *will* be treated as string concatenation. The output of this program would be 11.

### Subtracting

Subtraction is done via the `-` operator. You can do subtraction like so:

```
var x = 2 - 1
```

### Multiplication

Multiplication is done via the `*` operator. You can do multiplication like so:

```
var x = 10 * 2
```

## Division

Division is done via the `/` operator. Please note that this does *integer division*, and will not produce floating point values.

```
var x = 10 / 5
var y = 10 / 3 // y == 3 in this case
```

# Storing and Printing Values

## Variables

Variables are the bread and butter of any programming language. Let's take a look at how CatScript handles them.

### Declaration

We can declare variables using the `var` keyword, as you have seen in almost all the previous examples. You can tell the compiler that the variable has a specific type like so:

```
var x : int = -10
```

but you can also let the compiler just figure it out.

```
var x = -10
```

### Assignment

We can mutate any variable that is currently in scope through the `=` operator. For example:

```
var x = 1
x = x + 1
```

## Print Statements

Print statements are a good way to log data to the console. You can print any variable that is in scope, or any expression supported by CatScript.

```
print("Strings") // strings
print(1)         // integers
var x = null
print(x)         // even null values o.0
print([1, 2, 3]) // and lists! nicely formatted in the console
```

This program prints the following to `stdout` :

```
Strings
1
null
[1, 2, 3]
```

# Branching

## If Statements

If statements are the way that you can perform branching in your code. We can write if statements in CatScript using the `if` keyword

```
var x : int = 0
if (x < 5) {
  print(x+ " is less than 5")
}
```

You can also use the `else` keyword to perform other actions if the initial condition ( `x < 5` in this case) is not true

```
var x : int = 0
if (x < 5) {
  print(x+" is less than 5")
}
else {
  print(x+" is greater than or equal to 5")
}
```

# Looping

## For loops

For loops are great for iterating through a list. We can do a basic for loop like so:

```
// We can iterate through any constant list
for (x in [1, 2, 3]) {
  print(x)
  if (x == 2) {
    print("2 is in the list!")
  }
}

// even ones with all kinds of elements in them
for (x in ["Hello", null, 1]) {
  print(x)
}
```

Note that `x` in this case is a copy of whatever is in the list on the `n`th iteration, so changing `x` **will not** change items in the list. We do this so that our typing system remains sound even when we have a list of `object` s.

# Functions

## Definitions

We can declare functions via the `function` keyword. Here is an example:

```
function foo() {
  print("Foo!")
}
```

## Returning Values

We can tell the compiler that our function returns a value like so:

```
function foo() : string {
  return ""
}
```

First we say the function will return a `string` (on line one with the `: string` syntax), and then whenever we execute a `return` statement, the expression immediately following the `return` is, well, returned :-). (example on line 2)

Catscript also supports recursion. Let's take a look at a basic recursive function.

```
function foo(x : int) {
  print(x)
  if (x > 0) {
    foo(x-1)
  }
}
```

## Function Calls

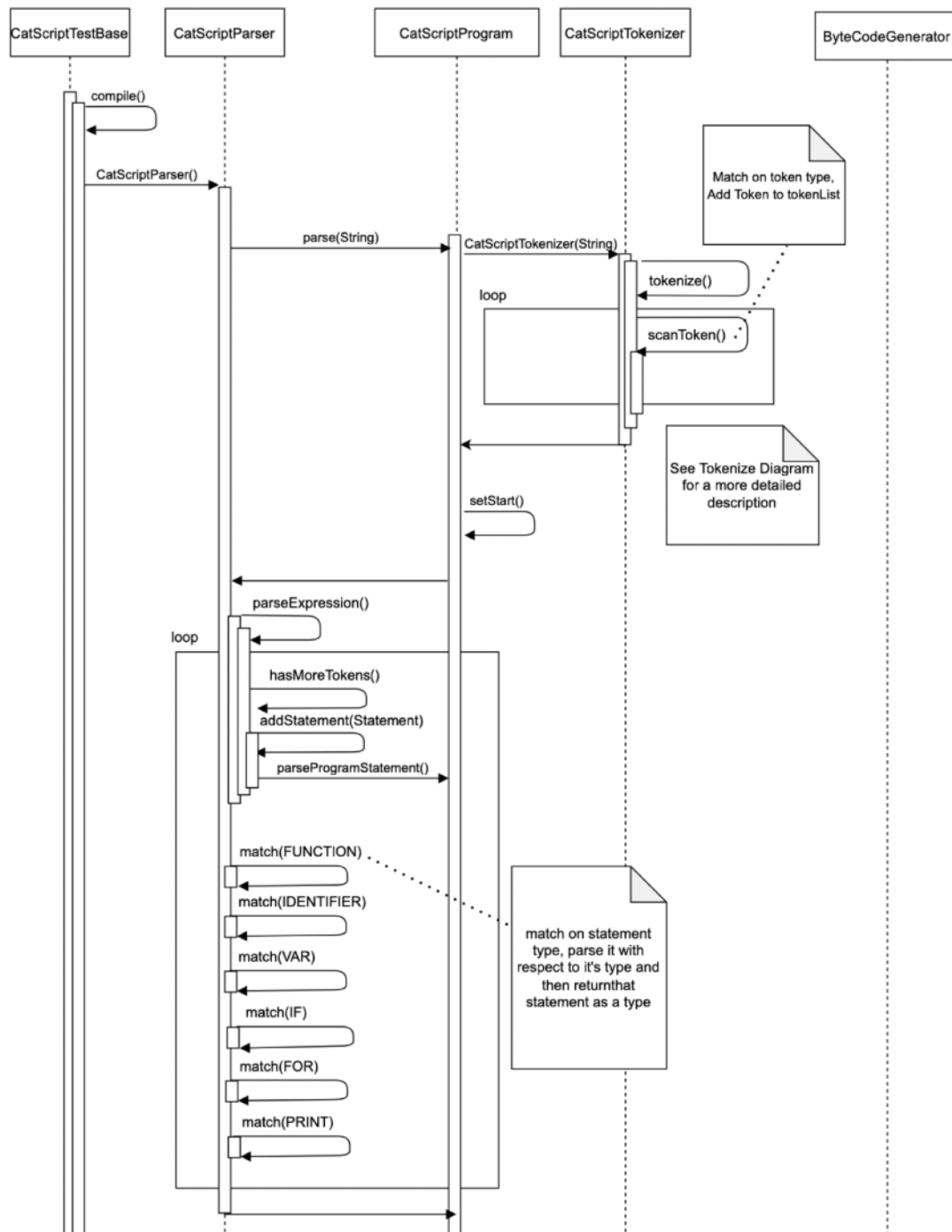
We can call a function by just doing `foo(args...)`, you can see in the example above that we can call the same function within the function's own definition. Here is a non-recursive example.

```
// a cute little wrapper function around 'print'
function foo(x : int) {
  print(x)
}
foo(1337)
```

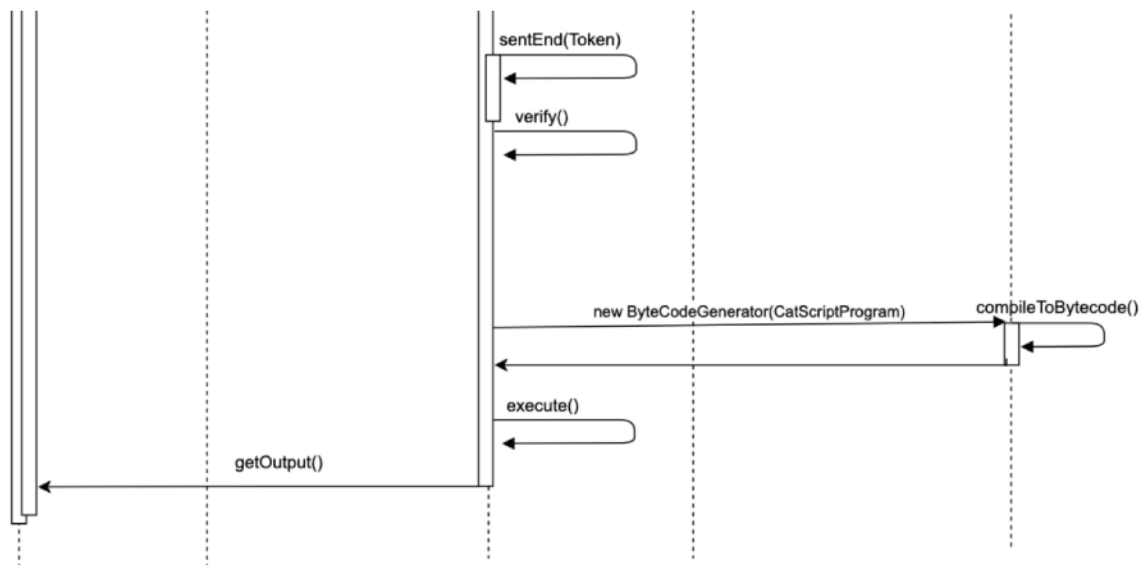


# UML

## CatScript Compiler Overview, page 1



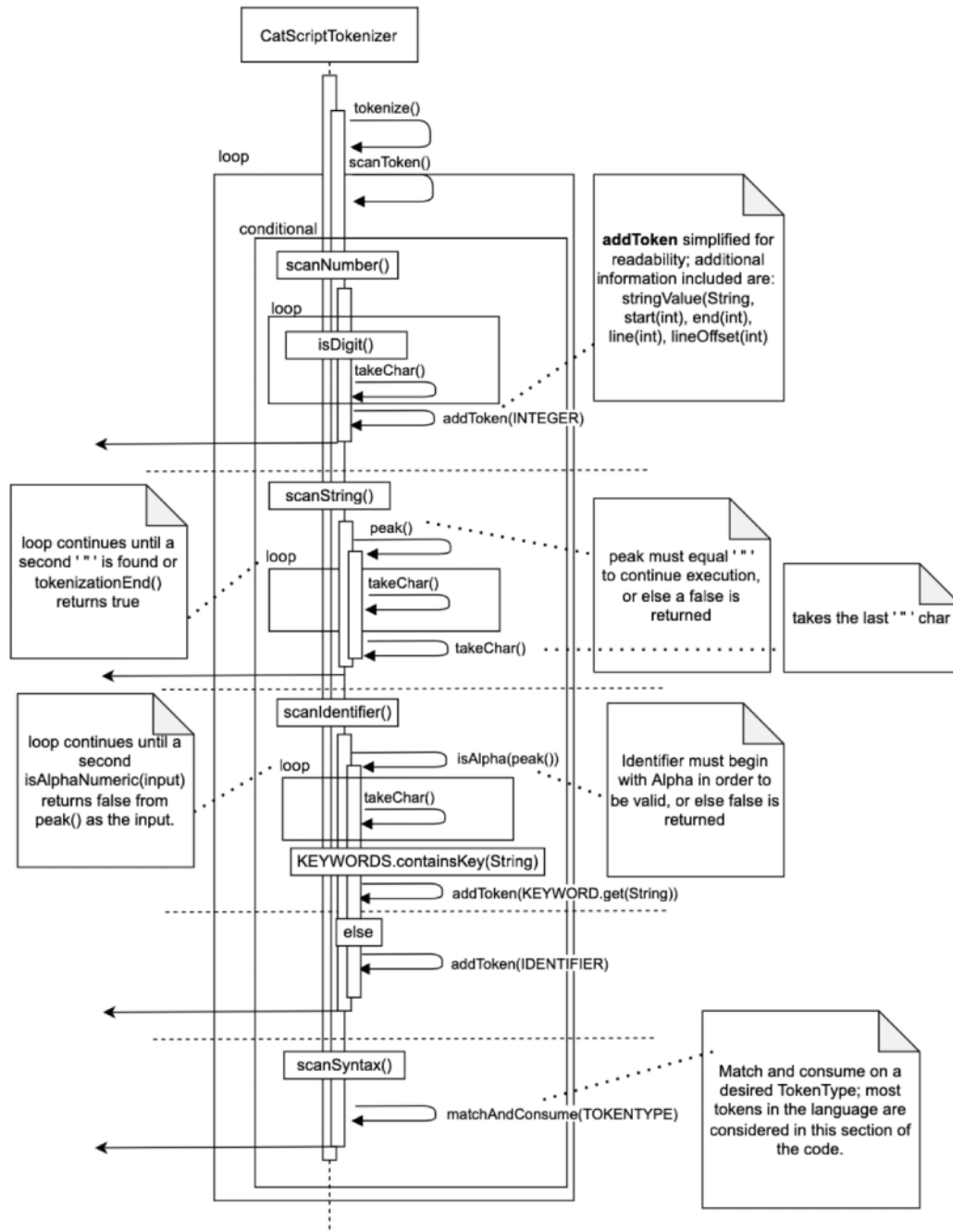
## CatScript Compiler Overview, page 2



The CatScript Compiler begins by considering code the user has written. In this UML diagram, this first step takes place in the **CatScriptTestBase**, where code has been written and is soon to be compiled. The **compile()** method starts the process by instantiating a **CatScriptParcer**. This object will then instantiate a **CatScriptProgram** object, which will then instantiate a **CatScriptTokenizer**. The Tokenizer scans the code the user has written and returns a **TokenList**, consisting of all of the tokens in the language. More about the **CatScriptTokenizer** will be described in the next UML diagram.

After tokenization, the parsing of expressions and statements begins in the aforementioned **CatScriptParcer** object. The parsing begins by first trying to parse an expression. If no expression exists, then a statement is parsed. The statement(s) match on specific 'statement' **Tokens**, creates an instance of that statement as an object, and then adds that object to the **CatScriptProgram**. After parsing has ended, the program is verified which includes whether variables are named correctly or exist and also static type checking. This verification is done on each statement or expression that exists in the **CatscriptProgram**. If verification raises no issues, the code is then compiled to Java ByteCode by instantiating a **ByteCodeGenerator** object with the **CatScriptProgram** as a parameter. The Program is finally executed, which is compared to a **CatScriptTestBase** test.

# CatScriptTokenizer



The **CatScriptTokenizer** scans Strings the user has written and interprets those into predefined Tokens important to the CatScript Language. The heart of the algorithm is the a **while loop** which scans each character until no more characters exist. Inside of the loop, there are a multitude of conditions for the desired character to match onto. This includes **scanNumber()**, **scanString()**, **scanIdentifier()**, and finally **scanSyntax()**. Most of these conditions do what they say, however **scanSyntax()** is more general and accounts for most of the tokens in the CatScript Lexicon. All match on specific criteria, such as numbers of `'"` characters, or even **isAlphaNumeric()** conditions. In each specific case listed above, a token is returned and added to the **CatScriptProgram TokenList**.

## Design Trade-offs

While programming CatScript, a number of design decisions were made. The most prevalent is why we chose a recursive descent parser, as opposed to a more conventional parser generator (such as [ANTLR](#)). The primary reasons for programming a recursive descent parser by hand are educational, readability, and debuggability.

Writing a parser from scratch without a generator is an incredibly revealing process. It requires a strong grasp on the grammar, tokenization, as well as parsing of the user's code. This opportunity is educational because everything needs to be programmed. Nothing is given to you, nothing is generated for you. That means that each token needs to be matched in tokenization and each statement or expression needs to be matched in parsing. There are many steps before the evaluating semantics even begin. Recursive descent pacers help reveal these subjects in a much brighter light.

Because no code is generated, the readability of the system can be meticulously refined and consistent throughout the scope of a project. This translates to function descriptors, parameters names, and even language consistency throughout a project. This allows any other person to easily understand the conventions and jargon of a project quickly.

Recursive descent parsers are incredibly debuggable because they are written by hand and follow a grammar tightly. This favors Test Driven Development highly and enables the stepping-through of code when implementing and debugging features or bugs. This fact was proven

many times in the development of the Compiler. The ease of debuggability helps when developing more complex features for the language. When unexpected behavior is present, stepping through the code is often easy and a godsend.

After speaking about the advantages of implementing a recursive descent parser, there are also some disadvantages. Most notably, it is a slow process. When using a tool like ANTLR, only a grammar and some lightweight 'generator code' is required to implement much of the backend of the compiler. This process lasts much longer when writing a recursive descent parser, as often thousands of lines of code need to be written. Generators, for the most part, work when they are given good rules and guidelines to generate. Code written by a programmer is often plagued with errors and bugs, which again takes time.

## Software Development Cycle

A Test Driven Development software development life cycle model was used throughout the entire project. Hundreds of tests were provided which correlated to specific milestones during the compiler's development. For example, the tokenizer, parser, and bytecode compilation all included tests which must pass in order for the compiler to work. This process was seamless, easy to follow, and efficient when designing the various components of the compiler.

An example of a test in the source code is below.

```
@Test
public void returnStatementExprInFunction() {
    FunctionDefinitionStatement expr = parseStatement("function x() : int
{return 10}");
    assertNotNull(expr);
    assertEquals("x", expr.getName());
    ReturnStatement returnStmt = (ReturnStatement) expr.getBody().get(0);
    assertNotNull(returnStmt);
    assertTrue(returnStmt.getExpression() instanceof IntegerLiteralExpression);
}
```