CSCI 468

Compilers

Spring 2024

Team Members:
Andrew Turenne
Coleman Lewis

Section 1: Source Code

Please see the attached source.zip file.

Section 2: Teamwork

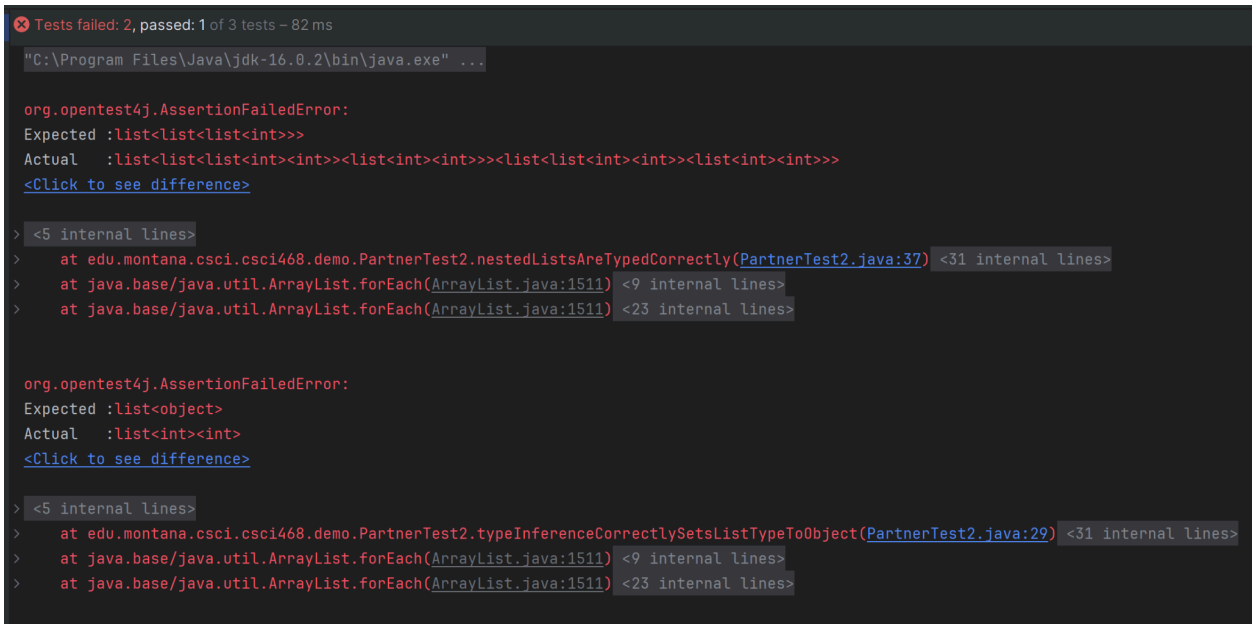
My teammate wrote 3 additional tests to further verify the correctness of the compiler.

```
public class PartnerTest2 extends CatscriptTestBase {
    @Test
    public void functionAllowsReturnInsideIfElseStatement() {
        FunctionDefinitionStatement expr = parseStatement( source: "function x() {if(1 < 10) {return true} else {return false}}");
        assertNotNull(expr);
        IfStatement ifStatement = (IfStatement) expr.getBody().get(0);
        ReturnStatement returnStmt1 = (ReturnStatement) ifStatement.getTrueStatements().get(0);
        ReturnStatement returnStmt2 = (ReturnStatement) ifStatement.getElseStatements().get(0);
        assertNotNull(returnStmt1);
        assertNotNull(returnStmt2);
        assertFalse(expr.hasErrors());
    }

    @Test
    public void typeInferenceCorrectlySetsListTypeToObject() {
        ListLiteralExpression expr = parseExpression( source: "[1, false, null, 3]");
        assertEquals( expected: 4, expr.getValues().size());
        assertEquals( expected: "list<object>", expr.getType().toString());
        assertFalse(expr.hasErrors());
    }

    @Test
    public void nestedListsAreTypedCorrectly() {
        ListLiteralExpression expr = parseExpression( source: "[[[1,2],[3,4]], [[3], [4], [5]], [[6]]]");
        assertNotNull(expr);
        assertEquals( expected: "list<list<list<int>>>", expr.getType().toString());
        assertEquals( expected: 3, expr.getValues().size());
    }
}
```

While the first test passed, the second and third did not.

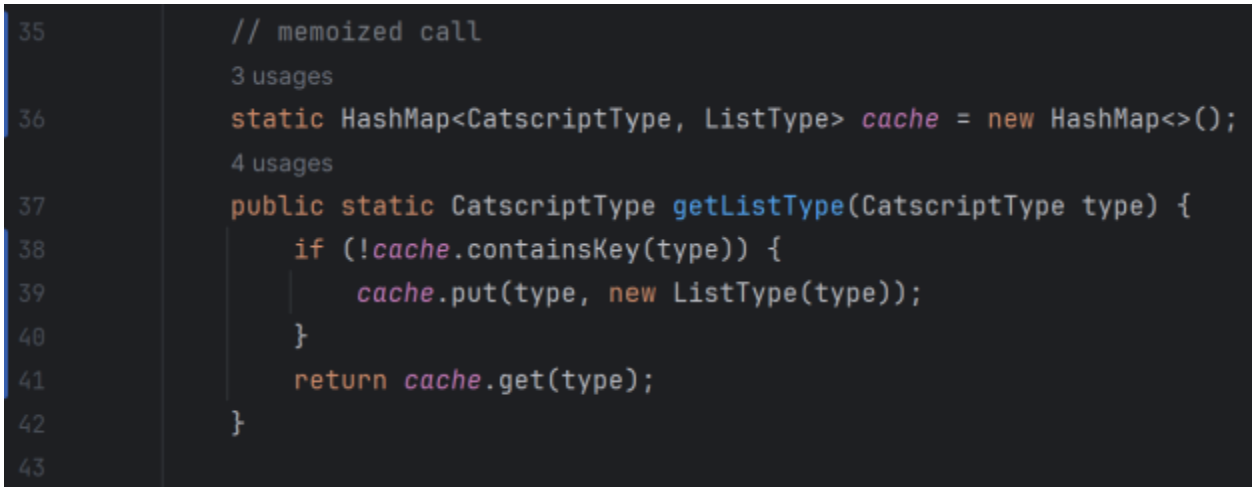


(Tests can be found in the attached PartnerTest.java) My partner's test seemed to have uncovered a bug in my list typing system. This issue was undetected by my tests and may have gone undetected for some time without these tests. Next, My teammate created the documentation for catscript. See the attached 'Catscript.md' The work on this project was split evenly between team members.

Section 3: Design Pattern

Within the CatscriptType class, there exists a function for getting a ListType, which extends CatscriptType, given a component type. Initially, this function created a new ListType object each time it was called. This, however, is wasteful, as multiple identical objects may be created.

To fix this wasteful method, we employ the memoization design pattern. This design pattern essentially caches data (objects) and when an object is requested which has already been made, the cached object is sent. If that object is not found in the cache, it is created and sent, then cached for future use.



In our implementation, the cache is represented with a hashmap, with CatscriptType objects as the keys and ListTypes as the values (Declared on line 36). On line 38, we check if the ListType with our given CatscriptType has been cached. This line exemplifies the memoization pattern concisely. If the value has not been cached,

we create it and cache it on line 39. We then return the ListType on line 41 (regardless of whether the ListType was just created or not).

Section 4: Technical Writing

Catscript Guide

Introduction

Catscript is a statically typed scripting language. Catscript supports a small type system of common data types such as integers, strings, and booleans. Catscript also includes essential control flow statements, such as for loops and if-else statements.

Here is a small example of a Catscript program:

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

The rest of this document will briefly illustrate the features and syntax of the Catscript language.

Features

Type System

The types of all variables, functions, and parameters in Catscript are known at *compile time*.

Catscript supports the following types:

Int

A 32-bit integer.

String

A java-style string.

Bool

A boolean value.

Null

The type of the value `null`. All types are assignable from the null type.

Void

The void type is exclusive to functions. Used to identify functions that do not return a value. No type is assignable from the void type.

Object

Any type of value. Any other type can be assigned *to* object, but object is *not* assignable to other types. For example, a string can be assigned to an object, but not vice versa.

List

A list of values. A list can consist of a single type (ie: a list of ints), or multiple types. If a list uses more than one type of value, its type is inferred to be `list<object>`. Catscript *does* support nested lists.

Equality Expressions

Equality expressions evaluate to a boolean value. If using the symbol `==`, an equality expression will evaluate to `true` if both values are equal to each other. If using the symbol `!=`, the expression will evaluate to `true` if both values are *NOT* equal to each other.

```
1 == 1
true == false
"foo" != "bar"
[1, 2, 3] != [1, 2, 3]
```

Comparison Expressions

Comparison expressions evaluate to a boolean value. Comparison expression can only be used to compare the values of integers. Attempting to compare any other types will result in an error.

The following operators are available for comparison expressions: greater than `>`, greater than or equal to `>=`, less than `<`, and less than or equal to `<=`.

```
1 < 2
3 > 4
5 >= 5
6 <= 7
```

Additive Expressions

Additive expressions are used for performing addition/subtraction computations on integers, as well as performing string concatenation. If an additive expression consists of two integers, the expression will be evaluated as the sum of the integers. Otherwise, the two values will be concatenated as a string. Integers are the only type which can be subtracted.

```
1 + 1
3 - 2
"Hello" + " World"
```

Factor Expressions

Factor expression can be used to perform multiplication/division computations on integers. Integers are the only valid types for factor expressions, any other type will result in an error.

```
4 * 5
9 / 3
var x : int = 12 * 12 / 2
```

Unary Expressions

Unary expressions are expressions with a single operand. The unary operators in Catscript are: `-` and the keyword, `not`. `-` can only be applied to integers, and it is used to flip a positive integer to its negative counterpart. The `not` keyword is exclusive to bools and can be used to flip a bool's value from `true` to `false`, or vice versa.

```
-12
not false
```

Variable Statement

A variable statement is used to declare new variables. A variable must begin with the `var` keyword. From there, the variable is assigned the name which will be used to refer to it. Optionally, a variable's type can be declared explicitly, but Catscript can also infer the variable's type.

```
var x = "foo"
```

This example demonstrates the variable `x` being instantiated with inferred typing.

```
var x : string = "foo"
```

This example demonstrates `x` being declared with explicit typing. In both of these examples, the variable `x` would be declared as a `string` with the value `"foo"`

Assignment Statement

After a variable has been declared, its value can be changed using an assignment statement.

```
var x : int = 5
x = 9
```

In this example, a variable statement is used to declare `x` as the integer value `5`. The value is then reassigned to the value `9` using an assignment statement.

Print Statements

A print statement in Catscript is used to output a value to the display terminal. It can be called for any type or expression, as they will be temporarily converted to the string type before being printed.

```
print("Hello, World!")
var x = "Hello, World!"
print(x)
```

This example demonstrates how the print statement can be used to display values or variables. In the first line, the command `print("Hello, World!")` will result in `Hello, World!` being printed on the display. In the following line, the variable `x` is set to that same value. So, when `print(x)` is called, the same value, `Hello, World!`, is outputted.

For loops

For loops can be used in Catscript to perform a set of computations over a list of values.

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

In the example above, a for loop iterates over a list of integers, printing the value of each list item. The variable `x` is a placeholder within the for loop's body. When the code executes, `x` is replaced with the list item corresponding to the current iteration of the loop.

If(-Else) Statements

If statements allow code to execute conditionally. If a condition is evaluated as `true`, the body of the if statement is executed. Optionally, an else statement can be appended to the end of an if statement. In that case, if the condition of the if-statement evaluates as `false`, the else statement will be executed instead. Else-if statements can also be used to set an alternate set of conditions, which can execute an alternate set of computations.

```
if (x > 10) {
    print(x)
    }

else if (x < 10) {
    print(x + 10)
    }

else {
    print(10)
    }
```

In the example above, the comparative expression `x > 10` is considered. If `x` is greater than `10`, the body of the if statement is executed, meaning the value of `x` will be printed. If `x` is less than `10`, the body of the else-if statement is executed. In this case, the value of `x + 10` is printed. If neither of these conditions is true (ie: `x` is equal to exactly `10`), `10` will be the printed value.

Functions

In Catscript, functions live within the same namespace as variables. This means that variables and functions within the same scope can not share the same name.

Function Declaration

Functions are declared using the `function` keyword and a name for the function. Then, a set of parameters must be declared. If there are no parameters, use an empty set of parentheses `()`. The parameter list is followed by an optional return type, then a series of statements (including the return statement), which is contained within a set of braces.

```
function x(a : int, b : bool) : int {
    if (b) {
        return a
    }
    else {
        return a + 1
    }
}
```

The example above shows the most explicit use case for a function definition. Explicit declaration of return types and parameter types are optional.

Function Calls

Once a function is declared (see the Function Declaration section above), it can be called using a function call statement A function call consists of the function's name, followed by a set of arguments corresponding to the parameters of the function.

```
var x = 6
x = adder(x, 4)
isPositive(x)
```

Return Statements

A return statement is used to return a value from a function, effectively marking the end of the function's execution. Return statements are written using the `return` keyword. Return statements are optional in function with a `void` return type. Otherwise, functions must have full return coverage.

```
function badExample(x : int) : bool {
    if (x > 0) {
        print(x)
        return true
    }
    else {
        print(x)
    }
}
```

The example above shows a function that does *NOT* have full return coverage. In this example, if `x` is less than or equal to `0`, the else statement, which does not have a return statement, will be executed. Return coverage issues can be fixed by ensuring each possible code path returns a value.

```
function one() : int {
    return 1
}

function voidExample() : void {
    return
}

function badExampleFixed(x : int) : bool {
    if (x > 0) {
        print(x)
        return true
    }
    else {
        print(x)
        return false
    }
}
```

The examples above demonstrate how return statements can be used to return a value or break from a void function. The final example shows how the invalid example from before can be tweaked to form a valid Catscript function.

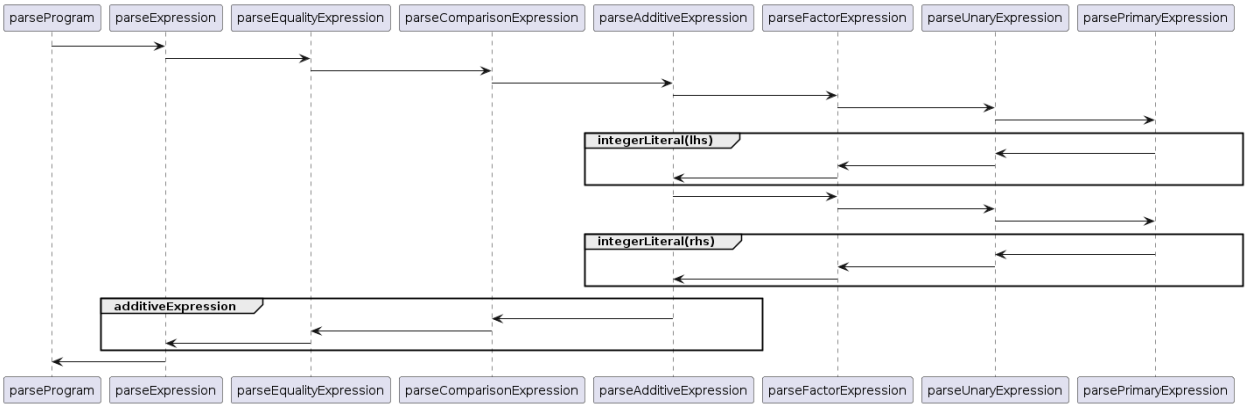
Keywords

The following words are keywords in Catscript, and can not be used as names for variables, function names, parameters, etc.

- else
- false
- function
- for
- in
- if
- not
- null
- print
- return
- true
- var

Section 5: UML

Sequence diagram for an additive expression with two integer literal expressions



As with all catscript code, the parser begins with parseProgram. In this case, the source code might look like '-10 - 8' for example. Initially, the parser invokes `parseExpression`. If an expression isn't identified, `ParseStatement` is invoked instead. The recursive descent approach continues with a series of calls to `parseXExpression` until `parseUnaryExpression` is reached. Within this recursive descent process, `parseExpression` method invokes `parseEqualityExpression`, followed by `parseComparisonExpression`, `parseAdditiveExpression`, `parseFactorExpression`, and eventually reaches `parseUnaryExpression`.

In parseUnaryExpression, if the left hand side integer literal has a negative sign, a 'minus' token will be found (for my example, it would find a negative sign). Regardless of if a negative number is found or not, 'parsePrimaryExpression' is called, and an 'integerLiteralExpression' is returned. The algorithm then returns back to the 'parseAdditiveExpression' call. The operator token (either 'plus' or 'minus') is matched

and consumed, in my example, the operator would be a minus. Next, the right hand side of the expression is parsed.

Just as was done with the left hand side, the recursive descent makes expression parsing calls until ‘parseUnaryExpression’ is reached. In my example, no ‘minus’ token is found, so ‘parsePrimaryExpression’ is called, which returns the right hand side ‘integerLiteralExpression’. Once again, the algorithm returns until reaching the ‘parseAdditiveReaction’. For this example, the ‘parseAdditiveExpression’ creates a new ‘additiveExpression’ object, which is then returned all the way to the top of the parse tree, because there are no more tokens besides ‘endOfFile’

Section 6: Design Trade-offs

Hand-writing a parser using a recursive descent algorithm presents several advantages over employing a generated parser. Firstly, it grants complete control and comprehension of the parsing process. Hand-written parsers allow for thorough understanding and facilitate debugging and modifications as necessary. Additionally, a notable benefit is the ability to customize your solution. Hand-written parsers can be tailored precisely to the project's requirements, optimizing performance and incorporating features that may be challenging to implement with a generated parser.

Furthermore, the readability and maintainability of hand-written parsers tend to be superior, as developers can structure the code in a manner that makes sense to them and others working on the project. Another advantage lies in error handling, where hand-written parsers offer more precise control over error messages and recovery strategies. Implementing a parser from scratch also serves as a valuable learning experience, deepening understanding of parsing techniques and enhancing programming skills. However, it's important to acknowledge that hand-writing a parser can be more time-consuming and prone to errors - particularly for complex grammars, compared to using a generated parser, which automates much of the process.

Section 7: Software Life-cycle

In this project, we implemented test driven development (TDD). In this strategy, the developers start with tests that their code should eventually pass. These tests will have a predetermined input, which is passed to the code, and a hardcoded output/result. A test passes when the code's output matches the hardcoded output. Working on a large piece of software can be daunting, but the test driven development model makes it easier to set goals and focus on small parts of functionality at a time. By simply looking at the list of failed tests, the developer can decide what still needs to be done, and can often infer what parts to complete first.

This development model provides a simple feedback cycle for completing code. When the developer changes some code, they run all tests again. Sometimes the new change makes previously passing tests now fail. Without the tests to drive development, however, this could go unnoticed for some time. With TTD the feedback is almost instant, the developer always has a clear direction in mind.