

Catscript Compiler

Alex Maliziola, Matthew Vandeberg

Montana State University

CSCI 468 Compilers

Carson Gross

May 8, 2024

Section 1: Program

<https://github.com/mvande/csci-468-spring2024-private/blob/main/capstone/portfolio/source.zip>

Section 2: Teamwork

My teammate (teammate 1) and I (teammate 2) split this program about 95/5 in terms of work done with me taking on the bulk of the work. I was responsible for writing all of the code for the Catscript compiler over the course of a few months. This included writing the tokenizer, expressions, and statements that are essential for the Catscript compiler. Most of my time was spent implementing new code to pass the provided tests and trouble shooting my code when it failed to pass any tests. My teammate was responsible for completing the technical document for this project as well as a couple final tests to verify the full functionality of the Catscript compiler. The technical documentation was written over the course of a couple months while the compiler was being developed. The final suite of tests were created by my teammate during the final weeks of development to go beyond the basic implementation tests that were already provided.

Section 3: Design pattern

Code snippet can be found at lines 35-40 in the source code file

src/main/java/edu/montana/csci/csci468/CatscriptType.java

```
35 //Design Pattern for Capstone Project
    2 usages
36 static HashMap<CatscriptType, ListType> cache = new HashMap<>();
    6 usages
37 public static CatscriptType getListType(CatscriptType type) {
38     cache.computeIfAbsent(type, listType -> new ListType(type));
39     return cache.get(type);
40 }
```

The design pattern I used in my capstone project was memoization and my implementation of it can be seen in the screen capture of my source code. I decided to use memoization here so that I wouldn't have to compute the type of a list every time I needed to check a given list's type. By using memoization to keep track of all the list types in a Catscript program I no longer have to spend time and resources checking the type of the values in a list to know the type of the list. Memoization also allows me to use less code to achieve the same goal. It's also very helpful when there is more than one list of the same type in a Catscript program since I only have to compute the type of list for one of the lists and then lookup the type of the other lists later from the cache.

Section 4: Technical writing

Catscript

Introduction

Catscript is a simple scripting language. Catscript can be compiled into Java bytecode so that it can be run on a number of machines. Here is an example of some simple catscript code:

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

Grammar and Features

Catscript Program

A program in catscript is written using zero or more program statements (see below). An example of a catscript program can be seen in the introduction.

```
catscript_program = { program_statement };
```

Statements

Program Statement

Program statements form the base of any catscript program. A program statement consists of either a statement or a function declaration statement (see both below).

```
program_statement = statement |  
                  function_declaration;
```

Examples of program statements:

```
var y = 1

function foo(x : bool) { print(x) }
```

Statement

A statement consists of one of the basic functional components of the catscript language. These components include control flow statements such as for loops, if/else statements, and function calls as well as statements for making variables and assigning values to them. Refer to a specific type of statement to see an example of each.

```
statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;
```

For Statement

For statements are used to loop over the values of a list and perform the statements in the body of the for statement. A for statement has a local variable that iterates through the values of the given list. This local variable can then be used inside the body of the for statement. Once the for statement reaches the end of the list, the for statement finishes and the rest of the code under the for statement is executed. It should be noted that nested for loops are possible in catscript.

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
               '{', { statement }, '}';
```

Example:

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

Result: 1 2 3

If Statement

If statements are a form of control flow that allows for conditional code execution. In catscript, an if statement consists of an expression (that reduces down to a boolean value), a body made up of statements executed if the expression is true, and an optional *else* branch to be executed if the expression is false. Additionally, more branches can be created by adding an *else if* condition to the if statement. Due to the nature of the catscript grammar, nesting if statements inside other if statements is possible just like in other similar languages.

```
if_statement = 'if', '(', expression, ')', '{',
               { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

Examples (assume x is an existing variable of type int):

```
if (true) { print("if") }

if (x >= 10) { print(x) } else { print(-1) }

if (x >= 2) {
    print(1)
} else if (x == 1) {
    print(2)
} else {
```

```
print (3)

}
```

Print Statement

A print statement is used to print out a given value or the result of a given expression.

Every data type (except `object`) can be used in a print statement. In addition, as long as an expression results in a single value, that expression can be used inside the parentheses of a print statement to print out the resulting value.

```
print_statement = 'print', '(', expression, ')'
```

Examples:

```
print(1)

print(true)

print("hello world")

print([1, 2, 3])
```

Variable Statement

Variable statements are used to create variables for data storage in a catscript program.

A variable statement consists of a variable name, an optional data type for the variable, and an expression that should reduce down to one value. This expression can be a simple data value (string, integer, boolean, etc) or it can be a function call to a function that returns a value to be stored in the variable. If a type is not specified for a variable, then the type will be determined from the expression's type.

```
variable_statement = 'var', IDENTIFIER,

                    [':', type_expression, ] '=', expression;
```

Examples:

```
var x = 1  
  
var y : bool = true  
  
var z : list<int> = [1, 2, 3]
```

Assignment Statement

Similar to the variable statement except the variable must already exist in the given scope to be used in an assignment statement. Once a variable is assigned a value, the variable's type is set and cannot be changed to a different type using an assignment statement. For example, a variable that has been declared with an integer value or of an integer type cannot be assigned a boolean value afterward. However, a variable with a defined type and a null value can be assigned a value of the variable's type. Also, a function call can be assigned to a variable as long as the function being called returns a value.

```
assignment_statement = IDENTIFIER, '=', expression;
```

Examples (assume x, y, and z are existing variables):

```
x = 1  
  
y = false  
  
z = [0, 4, 5]
```

Return Statement

Return statements return a value from a function.

```
return_statement = 'return' [, expression];
```

Examples (assume x and a are vars in this context):

```
return  
  
return x
```



```

return 12

return a >= 9

return "hello"

```

Function Statements

Function Declaration Statement

Function Declarations declare functions with parameters and an optional return type for later calling.

```

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
    [ ':' + type_expression ], '{', { function_body_statement }, '>';

```

Function Body Statement

The function body is composed of one or more statements that perform specified tasks within the scope of the function. Additionally, if a function has a specified type then the function body must contain a reachable return statement that returns a value with that same type. If the function has no type, then an empty return statement in the function can be used to interrupt or end the execution of the function.

```

function_body_statement = statement |
    return_statement;

```

Function Call Statement

See documentation for *function call expression*.

```

function_call_statement = function_call;

```

Function Parameters

A parameter list is just a list of locally-scoped variables that exist inside a given function. Parameters can be of different types or no type at all.

```
parameter_list = [ parameter, {',' parameter } ];
parameter = IDENTIFIER [ , ':', type_expression ];
```

Expressions

Type Expression

A type expression is used to identify data types. This type of expression is used in the following statement types: *parameters*, *function declarations*, and *variable statements*. Not all types are compatible with each other, for instance both `int`'s and `bool`'s are compatible with `string`'s, but `int`'s are not compatible with `bool`'s. This means the code, `print(1+"a")`, will produce the string "1a" however, `print(1+true)` results in a parse error. It is important to keep these type compatibilities in mind when using *type expressions* in catscript.

```
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [ , '<' , type_expression, '>' ]
```

Expression

An expression is the basis for any simple instruction in the catscript language. Expressions include basic tests such as comparison and equality tests as well as basic operations such as add, subtracting, multiplying, and dividing. Function calls and lists are also expressions in the catscript language. An expression could even be as simple as a basic string, integer, or boolean value.

```
expression = equality_expression;
```

Equality Expression

Equality expressions are used to compare two expressions to each other. Equality expressions always result in a boolean value (true or false) based on the input expressions. There are two ways to compare expressions, *equals* (==) and *not equals* (!=). The *equals* comparison results in a true value if the two expressions on either side are the same but will result in a false value if the two expressions are not the same. The *not equals* comparison does the inverse of the *equals* comparison. In addition to basic values, lists and variables can be compared to other lists or basic values.

```
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
```

Examples:

```
1 != 0
1 == 2
"hello" == "hello"

x == y
```

Results: true, false, true, depends on the values of x and y

Comparison Expression

The comparison expression is similar to the equality expression in that the comparison expression results in a boolean value depending on the input expressions and the specified comparison. There are four comparisons that can be performed: *greater than* >, *greater than or equal* >=, *less than* <, and *less than or equal* <=. These comparisons can only be made between two `int`'s or variables of `int` type.

```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };
```

Additive Expression

An additive expression is used to add or subtract two or more integers together as well as concatenating two or more strings. Adding and subtracting integer values is pretty simple, however string concatenation is more complicated. Both boolean values and integer values can be concatenated with string values without any type casting.

However, integer values and boolean values cannot be added or concatenated together.

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Examples:

```
1 + 2  
"asdf" + "ghjk"
```

Results: 3, "asdfghjk"

Factor Expression

A factor expression is used to multiply or divide two or more integer values together. An error will be thrown if non-integer values are used.

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

Examples:

```
1 * 2  
9*6  
10 * 15
```

Results: 2, 54, 150

Unary Expression

Unary expressions are used to negate boolean values/expressions and integer values. The *not* keyword is used for boolean expressions/values and turns a true value to false and vice versa. The minus sign turns a positive integer into a negative integer and vice versa. Unary expression can be stacked so `not not true` is a valid unary expression which is the same as `true`. Similarly, `--1` is the same as `1`.

```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```

Primary Expression

A primary expression is either an identifier (variable name, function name, non-key phrase, etc.), string value, integer value, boolean value, null value, list literal, function call, or parenthesized expression. A parenthesized expression is an expression surrounded by parentheses and is used to establish a hierarchy for evaluating expressions in catscript. An example of a parenthesized expression would be `1 + (1 * 2)`.

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |  
list_literal | function_call | "(" , expression , ")"
```

List literals consist of a list of values surrounded by square brackets with each value separated by a comma. These values can be in the form of an expression as long as it evaluates to a singular value.

```
list_literal = '[' , expression , { ',' , expression } ']' ;
```

Function Call Expression

Function calls work by referencing the name of a declared function along with any specified parameters. If a function call does not include an argument list despite the

function having parameters or the data types in the argument list do not match the data types in the parameter list then catscript will throw an error. If a function does not have any parameters, then the function call should not have any arguments between the braces.

```
function_call = IDENTIFIER, '(', argument_list , ')'
```

Examples:

```
function foo(): { print("foo") }

foo()

function bar(x: int, y: bool) { if(y) { print(x) } }

bar(1, true)
```

An argument list is basically the same as a list literal except the data is not enclosed in brackets.

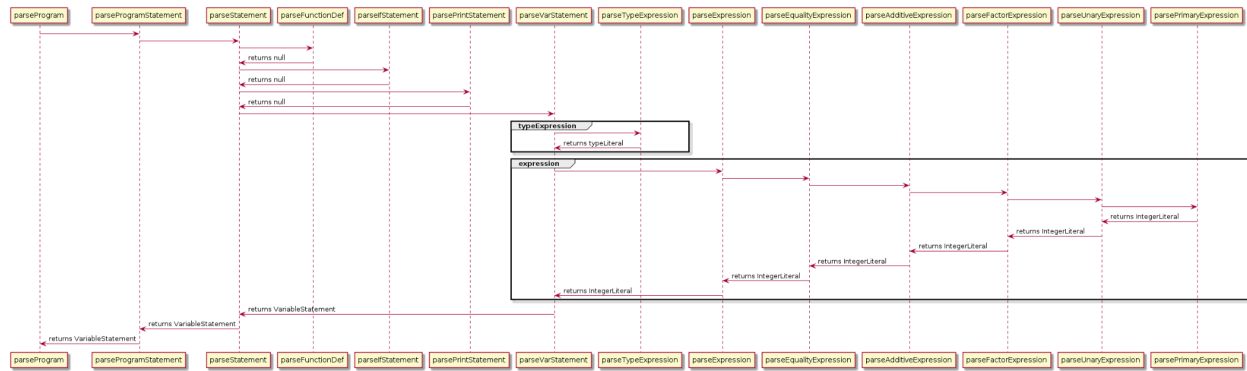
```
argument_list = [ expression , { ',', expression } ]
```

Types

- `int`: Represents any 32 bit integer
- `string`: Java-style string immutable string
- `bool`: Boolean value (true or false)
- `list<x>`: An immutable list of values of type 'x'
- `null`: Represents the *null* value
- `object`: A base type of any value

Section 5: UML

UML diagram demonstrating the recursive parse tree for the statement `var x:int = 1`



Section 6: Design trade-offs

One of the trade-offs made in this project was choosing to design a parser using recursive descent instead of a parser generator. One of the benefits of using recursive descent is that it expresses the natural recursive nature of the Catscript grammar in a much more obvious way. Since parser generators generate the code to parse a grammar instead of using hand-written code, parser generators are generally harder to debug compared to recursive descent. Additionally, the code generated from parser generators is harder to interpret and make sense of by humans. The problems associated with understanding the code and methodology behind parser generators, especially compared to the code and methodology of recursive descent, make it less appealing to use for the catscript parser. It should be noted that parser generators do use less code than recursive descent, however the disadvantages of parser generators, especially when examining the goals of this project, make recursive descent the obvious choice for designing a parser for the catscript language.

Section 7: Software development life cycle model

For this project a Test Driven Development (TDD) model was used to write the Catscript compiler. The TDD model consists of a five-step loop that allows for constant iteration of a project until every goal has been achieved. The five steps can be broken down as follows: write tests for some of the requirements, run the new tests, rewrite the code base to satisfy the new tests, re-run the tests to verify, refactor code base to be more readable and simpler, and finally repeat with a new subset of the requirements that haven't been met yet. This process is advantageous for large projects since it allows a project to be broken into much smaller tasks that can be tackled one at a time. TDD is also very useful since writing your own tests makes you pay more attention to the details of the project's requirements.

The part of TDD that was the most useful for me was being able to break up the many requirements of the Catscript compiler into manageable, testable sections that could be tackled one at a time and then built upon as the project came together. Prior to completing this project the only experience I had with TDD was in CSCI 366 so it was good to gain more experience with it through this project. Based on my current experience using TDD in various classes and projects, I'm going to be using TDD to develop many of my future projects.