

CSCI 468 - Compilers

Spring Semester 2023

Alex Krings

Nicholas Ceccanti

Section 1: Program

Program source code:

See `source.zip` in this directory.

Program specification:

```
Catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

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

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

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

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

variable_statement = 'var', IDENTIFIER,
                    [ ':', type_expression, ] '=', expression;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

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

function_body_statement = statement |
                        return_statement;
```

```

parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

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

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

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

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

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

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

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

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

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

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

```

Section 2: Teamwork

Our team worked on the project by splitting the roles of the two partners into primary, documentation and testing engineer. Team member 1 was the primary engineer, writing all the code for the project. Team member 2 was the documentation and testing engineer, writing the unit tests and documentation for the Catscript language. Team member 1 contributed approximately 50% of the total project and team member 2 contributed approximately 50% of the total project in terms of total time spent.

Section 3: Design Pattern

One of the design patterns used in the compiler was the memoization pattern. This pattern was used in `getListType()` method of the `CatscriptType` class. The memoization pattern is similar to caching, where something is stored in an easier to get place if it is retrieved because it might be retrieved again later. This prevents the retrieved object from being constructed again a second time. This trades runtime speed for slightly higher memory usage. In the Catscript compiler, we memoize the Catscript types in a hashmap. This method is used to retrieve the type of a Catscript list. If the hash map already contains the specific type it will just retrieve it from the hash map, if it is not already contained in the hash map, the program will retrieve the type and then save that type into the hash map. This pattern speeds up the `getListType()` program by trading the memory space taken up by the hash map for the speed of a hashmap to retrieve a new `CatscriptType` object. The `getListType()` is a frequently called method and therefore the use of the memoization design pattern increases the runtime speed of the compiler.

The memoization design pattern source code, located at lines 37-46 in `CatscriptType.java`:

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

Section 4: Technical Writing

Catscript Guide

This document serves as a living technical document and guide for the Catscript scripting language.

Introduction

Catscript is a simple scripting language that implements many of the main features of other scripting languages.

Below is a simple example which creates a function called `printHelloWorld`, which prints the following string `Hello World`, and then we call said function.

```
function printHelloWorld() {  
    print("Hello World")  
}
```

```
printHelloWorld()
```

Comments

Comments are supported in Catscript, but it is only line by line. Just like in Java, `//` comments out the rest of the line.

```
// This is a comment
```

Assignment Operation and Variables

The assignment operation is very simple and requires the identifier/variable name followed by an `=` and then whatever it is being assigned to. This assigns an item to the identifier/variable name.

```
variable_name = new_assignment or x = y
```

When creating a new variable, you must include the keyword `var` followed by the name of the variable, and then you use the equal sign to assign the variable to that item. Here we created a new variable `x` and set it equal to `10`.

```
var x = 10
```

Since we didn't specify the type explicitly, Catscript automatically assigns the type to the variable. In this case `x` would be of type `int`. Optionally, you can specify the type when creating the variable. This is done by adding a colon `:` after the name of the variable, and then the type.

```
var temp : string = "Hello World!" or var booleanVar : bool = false
```

Catscript Types

Catscript is statically typed and has the following types:

- `int` - a 32 bit integer
- `string` - a Java-style string
- `bool` - a boolean value (`true/false`)
- `list` - a list of value with the type '`x`'
- `null` - a null type

- object - any type, any value

Throughout a variable's lifetime, it must follow and maintain the same type. For example, type `int` can't be assigned to type `string` and vice versa. `object` and `null` are the slight exceptions to this rule and are explained below.

Integers

These are your normal whole numbers. And since it is 32 bits, integers have a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647.

Lastly, integers can be denoted by using the `int` keyword.

```
var x : int = 10 or var x = 10
```

Strings

Strings are denoted using the `string` keyword and must be enclosed between a pair of double quotes.

```
var stringOne = "Hola" or var stringTwo : string = "Bonjour"
```

Concatenation operations are implemented and a `+` is used to perform the operation on the string.

```
var stringOne = "Hello"
var stringTwo = "World"

var concat = stringOne + stringTwo
print(concat) // This would output "HelloWorld"
```

Catscript, like JavaScript, also allows for concatenations between strings, ints and `null`. The resulting output between the two would be of type string.

```
var string = "Temp"
print(temp + 1) // This would print "Temp1"
print(null + "a") // This would print "nulla"
```

Booleans

Boolean values are your traditional `true` and `false` values that are denoted with the `bool` keyword.

```
var boolExample = true or var anotherOne : bool = false.
```

List

Lists are extremely similar to java arrays. The biggest similarity that they share is the fact that the lists are covariant. This means that list one would be assignable to list two, if the list type of list one is assignable to the type of list two. However, lists are immutable, so once one is created, it cannot be changed.

Another cool feature of Catscript lists is they can be nested and there is no limit to the dimensionality. When creating a list, the elements of the list have to be the same type, unless the list is of type object. An int list can only contain integers, a string list can only contain strings, but an object can contain both int and string.

Specifying that it is of type list is a little different. The first part is the same, where you use a colon : followed by list. However, the next step is to declare what type the list is using left and right braces with the type in the middle <type>.

```
var x = [1, 2, 3] or var y : list<string> = ["a", "b", "b", "a"] or var z = [1, 2, [2,4]]
```

Object

Objects in Catscript behave very similarly to those in Java. Essentially, anything with type object can have any type assigned to it, including lists of type object.

These are denoted using the keyword object.

```
var x : object = "I am a string" or var y : object = 1.
```

Null

The null value in Catscript can only be assigned to type null. Only null can be assigned to something of type null, but everything else (int, string, boolean, etc.) can be assigned to a null value.

This is denoted with the keyword null.

```
var nullValue : null = null or var nullInt : int = null.
```

Catscript Features

Print Statement

The print statement is a core feature of any language and shares similarities with Python. To print something, you use the keyword print followed by a left parentheses, what you want to output, and then a right parentheses. If you are printing a message, the message

needs to be contained in a pair of double quotes.

```
print("Hello World")
```

You can also call functions inside of print statements, perform arithmetic operations, comparison operations, and print variables. When printing different sets of items, like a message followed by a variable, you have to separate with a comma.

```
var x = 1
var y = 2

print("Math operations") //This will print 'Math Operations'
print("Var x is equal to", x) // This prints the message followed by the
var x
print(x > y) // This statement is false, so it will print false

print(foo(3, 1)) // This will output 4 It calls the function foo which
takes in two ints and returns the addition of the two.
function foo(x : int, y : int) {
    return x + y
}
```

Simple Arithmetic

Catscript has basic arithmetic and math operations. There is adding +, subtract -, multiply *, and divide /. Like other operations it performs multiplication and division operations before addition and subtraction.

You can use parentheses to create math equations and the order of operations to your choosing.

```
var x = 1 + 1 // equals 2 or var a = (1 + 3) * 4 // equals 16
```

There is no implied multiplication, i.e. 4(1+1) throws a parse error.

Comparison and Equality Operations

Comparison operations are similar to those of Java and Python. When performing a comparison, it returns either true or false. There are six total comparison operations: greater than >, greater than or equal >=, less than <, less than or equal to, <= equals ==, and not equals !=.

```
print(1 == 4) //false or print(1 >= 1) //true
```


Catscript supports two unary expressions `not` and `-`. The `not` keyword is the equivalent to the logical NOT operator. The `not` keyword flips the Boolean value given in the expression, and `not` can only be used with Boolean values. `-` can be used to flip the sign.

```
if(not x) {  
    //do something  
}
```

```
var x = -1  
print(-x) //1
```

Function Declaration and Calls

Function Declaration

In order to create a function you must use the keyword `function` followed by the name/identifier. Following the identifier is a left parenthesis (with arguments in the middle if desired, separated by a comma, and then enclosed with a right parenthesis). Next the body of the function has to be contained in a pair of brackets {}.

If there are no arguments you just leave the () empty.

```
function myFirstFunction(){  
    print("Hooray!")  
}
```

Similar to Python if there are any arguments, you put the identifier of the variable followed by a colon followed by the type. If there are multiple arguments, you separate them with a comma.

```
function functionWithArgs(x : int, y : string) {  
    print(x) // Prints whatever value was passed to x  
    print(y) // Prints whatever value was passed into y  
}
```

Returns

Another important aspect of functions is the `return` feature. In order to `return` from a function, you must specify a type in the function declaration. Whatever type is specified, then that type must be returned. Another important note is that if you return within a control branch (if/else statement), there must be a `return` statement to satisfy the other branches.

If the function is `void` (i.e. no type is specified) then you are able to use a `return` (with no associated value) to break out of the function. Unlike a function that returns a value, it is not necessary to add a `return` to all control branches. An example of this is shown below.

In order to specify a return type for a function, you add a colon `:` after the `()` arguments followed by the type you would like to return.

```
function function_name(optional arguments) : type { <body> }
```

```
function foo(x : int, y : int) : bool {  
    if(x != y) {  
        return false  
    } else {  
        return true  
    }  
}
```

```
function foo(x : int, y : int){  
    return x + y  
}  
var temp = 0
```

```
temp = foo(1, 2)  
print(temp) // Will output 3
```

```
//void function  
function foo(x : int){  
    if(x != 0){  
        return  
    } else {  
        // Since the function is void, there is no need to  
        // to add a return statement here  
        print("Hello there!")  
    }  
}
```

```
// return function  
function foo2(x : int) : string{  
    if(x > 10){
```

```

        return "Greater than ten"
    } else {
        return "Less than ten"
    }
    //You could also return here instead of the else
}
print(foo(1))

```

Function call

When calling a function it invokes the body of the desired function. First you use the name of the function followed by parentheses and any arguments that are needed to be passed in. Just like a declaration, you separate the arguments by a comma ,.

It is important to note that you must pass the same number of arguments expected as well as the same type.

```

function noArgFunc(){
    print("I have no args!")
}

```

noArgFunc() // Calls and invoked the function

```

function iHaveArgs(x : list, y : int){
    print(x)
    print(y*2)
}

```

```

var a = [1, 2, 3]
var b = 10

```

iHaveArgs(a, b) // This will call the function, which will then print x
//and perform a math operation on b and output it's product

If and Else Statements

If statements are very similar to those of Java. The main difference being you can't perform multiple checks inside an if, there is no or || operator. To create an if statement you start with the keyword **if** followed by a left paren (the conditional statement). You then create the body of the **if** statement between curly braces {<code>}.

Like other **if** statements, if the conditional is true it will perform the body of the if. If false, it will do nothing, unless you create an **else** statement! To have an **else** you need to place it after the closing curly brace of the **if** statement. Followed by its own set of curly braces.

In the conditional statement, you must perform some sort of comparison check.

```
if(x == 0){
    print("hello")
}

if(x > y){
    print("X is greater than y")
} else{
    print("Y is greater than y")
}
```

For Loops

The for loop is probably the most different compared to other languages. A for loop iterates through a list rather than a set number or condition.

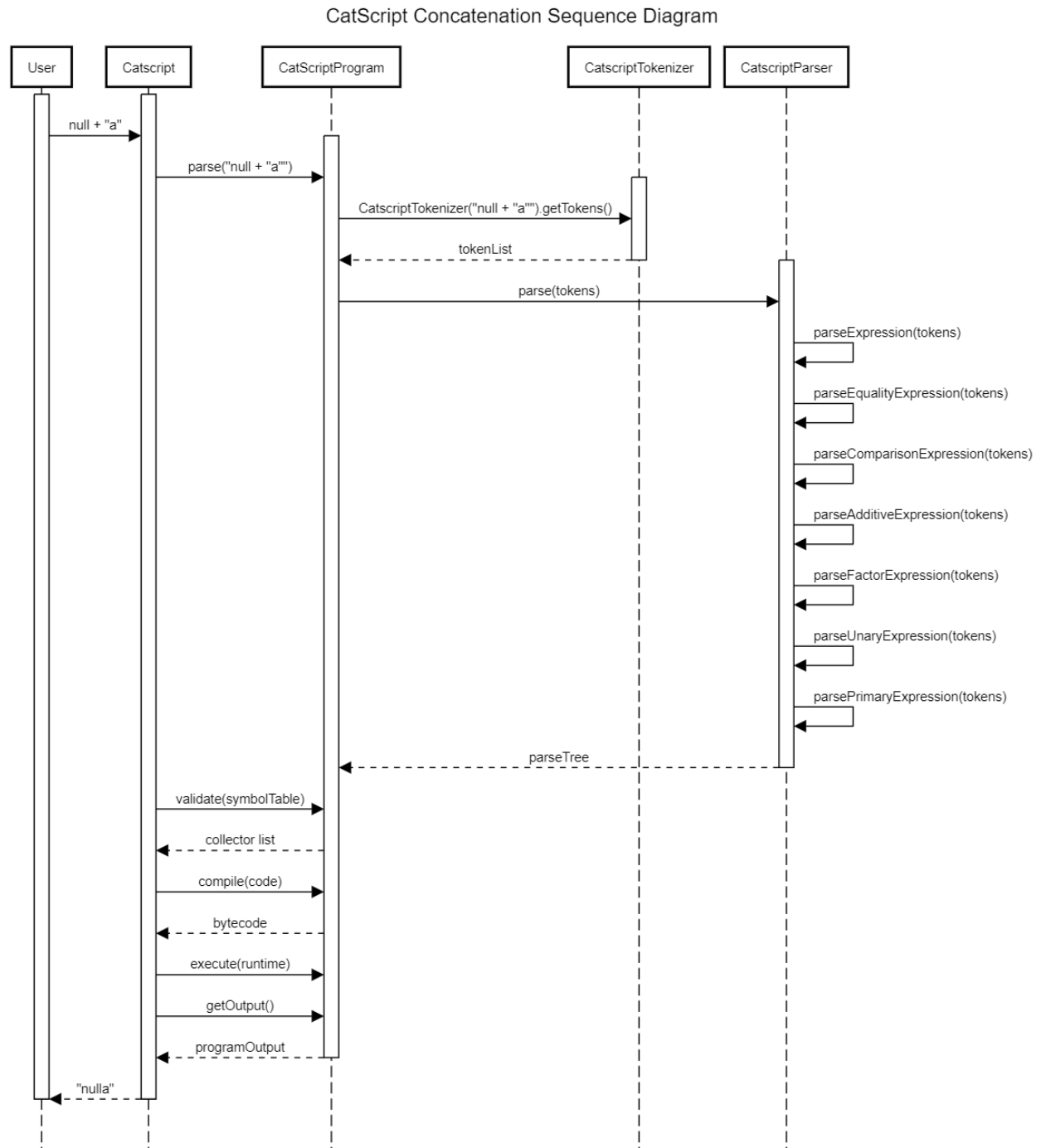
In order to create a for loop, you initiate it by using the keyword `for` followed by parentheses. Inside the parentheses is the iterator variable followed by the keyword `in` followed by the list that will be iterated through. Lastly, curly braces are used for the body of the for loop.

For loops will iterate for the length of the list and execute the code in its body every iteration. The iterator variable is set equal to whatever the current value in the list.

```
for(x in [1, 2, 3]){
    print(x) // This will print 1, then 2, and then 3.
}

var x = 0
for(a in [2, 4, 8]){
    print(x+a)
    // This will print 2, 6, 14
}
```

Section 5: UML



Section 6: Design Trade-offs

A major design tradeoff was the decision to use a recursive descent parser written by hand instead of using a parser generator. A parser generator is a piece of software that takes in

a specification of a language and it auto generates a parser in code given the specification. The decision was made to use recursive descent as opposed to a parser generator for two reasons. The first reason is that parse generators are tools already written by someone else and are very complex. The recursive descent parser needs to be written by hand and it's simpler but is able to do the same job as the parse generator. By writing the compiler by hand, we as students receive a better learning experience with respect to compilers and software engineering skills in general. By getting hands-on experience with writing code for the parser we gain a better understanding and improve our programming skills. Since the recursive descent parser is simpler and is more compact, it also allows us to include more unique features with the recursive descent parser than we would using a parser generator. The second reason for using a recursive descent, is that they are frequently used in industry. Many compiler classes in academia use parse generators while recursive descent parsing is used in industry. By learning an architecture that is common in real world use prepares us more for our career and to be efficient programmers in the future.

Section 7: Software Development Life Cycle Model

The development that we used for this project was test driven development. A series of unit tests around the grammar of the language. The test-driven development model is very convenient because it allows implementation requirements into code very early in development. By creating unit tests beforehand and writing the code around the unit tests, it makes the requirements very visible and easy to identify successes or failures. By using unit tests with an IDE it is also very easy to keep track of progress, as it remembers what test passed and failed in the previous run of tests. Also, the unit test suite allowed greater visibility of how changes affected the outcomes of other tests. For example, if someone makes a change in code to satisfy a unit test, that change potential can cause another unit test to fail. If one runs the all unit tests after the change, one is able to immediately recognize that further action is required.

This model also hindered our team, because it also caused a lot of bugs to persist after all tests had been completed. Test driven development only tests for the tests specified, if the person writing the tests does not cover edge cases or does not write diverse scenarios then it is very likely that a bug will get past the unit tests. It is impossible for a unit test suite to cover every single possible test case; therefore the tests cannot catch every single bug. There are many possible inputs in a compiler and our test suite contained roughly one hundred tests. This is not nearly enough to catch all the bugs in the compiler. Also, many of the tests covered atomic functionality of the compiler. When writing unit tests for the integration of multiple functionalities is where many of the bugs appeared to occur.