

Montana State University

Compilers: CSCI 468

Spring 2023

Miller, Benjamin

Senecal, Audrey

Section 1: Program

Attach the source listing of the program that you wrote for your capstone course (CSCI 468 or CSCI 483). Include the specifications for the program.

<https://github.com/benfmiller/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Catscript grammar:

```
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, '>']
```

CatScript is statically typed, with a small type system as follows:

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list<x> - a list of value with the type 'x'
- null - the null type
- object - any type of value

Section 2: Teamwork

Describe how your team worked on this capstone project. List each team member's primary contributions and estimate the percentage of time that was spent by each team member on the project. Identify team members generically as team member 1, team member 2, etc.

For the purposes of this document, I will identify myself as team member 1 and my partner, Audrey Senecal, is designated as team member 2. We worked together to design, implement, test, and document the Catscript compiler. Team member 1's role mostly focused on implementation while team member 2 acted as a test engineer and worked on the documentation for the specifications. The tests were a large driver for the project, so team member 1's estimated time contribution is 65% while team member 2's time contribution is estimated to be 35%.

Section 3: Design pattern

Identify one design pattern that was used in your capstone project and describe exactly where in the code it is located. Highlight the design pattern in yellow. Explain why you used the pattern and didn't just code directly.

A concrete design pattern that was used is the **memoization** pattern. This is located at *src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java:37*. This was used to greatly speed up the Catscript type system as type objects did not need to be created every time a type is referred to in a Catscript program. The Catscript type system is relatively small and simple, and each specific type object is treated immutably and

interchangeably, so it did not make sense to create separate, small objects for each type interaction. Catscript's type system is also static, so these objects can be created easily at compile time. This should also help with CPU caching as all available types should fit easily in the CPU cache. This feature was not tested for performance comparisons, so no concrete numbers are available, but the benefits are apparent.

I would argue that most design patterns are just ways to use interfaces to swap functionality more easily or decouple the program. The Catscript compiler makes judicious use of interfaces or abstract classes to easily swap objects and deal with them in a consistent manner. This is done at all layers of the compiler. All Tokens can be handled similarly by the tokenizer, and the same is true with the parser, the executor, and the JVM compiler.

Section 4: Technical writing

Include the technical document that accompanied your capstone project.

Included started on the next page.

Catscript Guide

Introduction

CatScript is a simple scripting language that features a small assortment of control flow tools and other characteristics, such as static typing, implicit assignment, and error reporting.

Features

Print Statements

Print accepts an expression as a parameter, the value of which is then printed to an output stream.

Syntax

```
print(expression)
```

Parameters
expression : CatScript expression An arithmetic, relational or primary expression

Example

```
print(10 + 12)
```

```
> 22
```

```
print(10 > 12)
```

```
> false
```

Types

CatScript is statically typed, and features a small type system described below.

int	Represents numeric values as 32-bit integers
string	Represents a sequence of characters
bool	Represents the boolean values True or False
list<type>	Represents a read-only collection of values with some type, in an ordered structure
object	Represents a value of any type
null	Represents the Null type

Syntax

```
var a : int = 5
var b : string = "Hello World"
var c : bool = false
var d : list <int> = [1, 2, 3]
var e : object = "Object"
```

Creating Variables

The *var* statement is used to assign a value to a user-defined identifier.

Syntax

```
var var_name : var_type = expression
```

Parameters	<p>var_name : CatScript identifier A user-defined identifier representing the name of the variable</p> <p>*var_type : CatScript type The explicit type of the variable. If this is not supplied, the type is implicitly assigned</p> <p>expression : CatScript expression An arithmetic, relational or primary expression, the value of which is assigned to the variable</p>
------------	--

Example

```
var x : int = 1 + 2
print(x)

> 3
```

Supported Operators

CatScript supports several different relational, arithmetic and logical expressions, whose operators/syntaxes are described below.

Syntax

Relational Operators

```
x == y    :    supported types: int, bool, object, list, string, null
x != y    :    supported types: int, bool, object, list, string, null
x > y     :    supported types: int
x < y     :    supported types: int
x >= y    :    supported types: int
x <= y    :    supported types: int
```

Arithmetic Operators

```
x + y     :    supported types: int, string
x - y     :    supported types: int
x * y     :    supported types: int
x / y     :    supported types: int
```

Logical/Unary Operators

```
not x     :    supported types: bool
-x        :    supported types: int
```

Control Flow Tools

For Statements

For iterates over the members of a given list in order, using the user-defined identifier as an iterator variable and executing the statements within the body each time.

Syntax

```
for (identifier in list) {  
    body  
}
```

Parameters	identifier : CatScript identifier An identifier representing the name of the iterator variable
	list : CatScript list A list containing one or more values to be iterated over
	body : CatScript statement(s) 0 or more statements to be executed over each iteration of the loop

Example

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

If Statements

The *If* statement allows the conditional execution of statements, executing a given block of code only if a supplied expression resolves to true.

Syntax

```

if (condition) {
    if_body
}
else {
    else_body
}

```

Parameters

condition : CatScript bool
An expression which resolves to a boolean value, or a bool literal

if_body : CatScript statement(s)
0 or more statements to be executed only if the expression supplied to the if statement resolves to true

*else_body : CatScript statement(s)
0 or more statements to be executed only if the expression supplied to the if statement resolves to false and an associated else block exists

Example

```

if (false) {
    print(1)
}
else {
    print(2)
}

> 2

```

Defining Functions

The *function* statement is used to create a new subroutine that may be called via its user-defined identifier.

Syntax

```

function func_name(arg : arg_type, ...) : return_type {
    body
}

```

```
    return return_value
}
```

Parameters	<p>func_name : CatScript identifier An identifier representing the function name</p> <p>*arg : CatScript identifier 0 or more identifiers representing the names of the function arguments</p> <p>*arg_type : CatScript type The explicit type assigned to a given argument</p> <p>*return_type : CatScript type The type of the value that the function will return. If this is not supplied, the function's return type is set to void</p> <p>body : CatScript statement(s) 0 or more statements to be executed when the function is called</p> <p>return_value : CatScript expression An arithmetic, relational or primary expression whose value is returned to the function call only when an associated return statement is present within the function body, which is required when a return type is defined in the function header</p>
------------	--

Example

```
function foo(a, b) { print(a + b) }

function bar(a : int, b : int) : int { return a + b }
```

Calling Functions

In CatScript, a function may be called via its user-defined identifier. If the called function requires arguments, they must be supplied. If the called function returns a value, it may be assigned to a new variable or used within other statements.

Syntax

```
func_name(args)
```

Parameters

`func_name` : CatScript identifier
An identifier representing the name of the function to be called

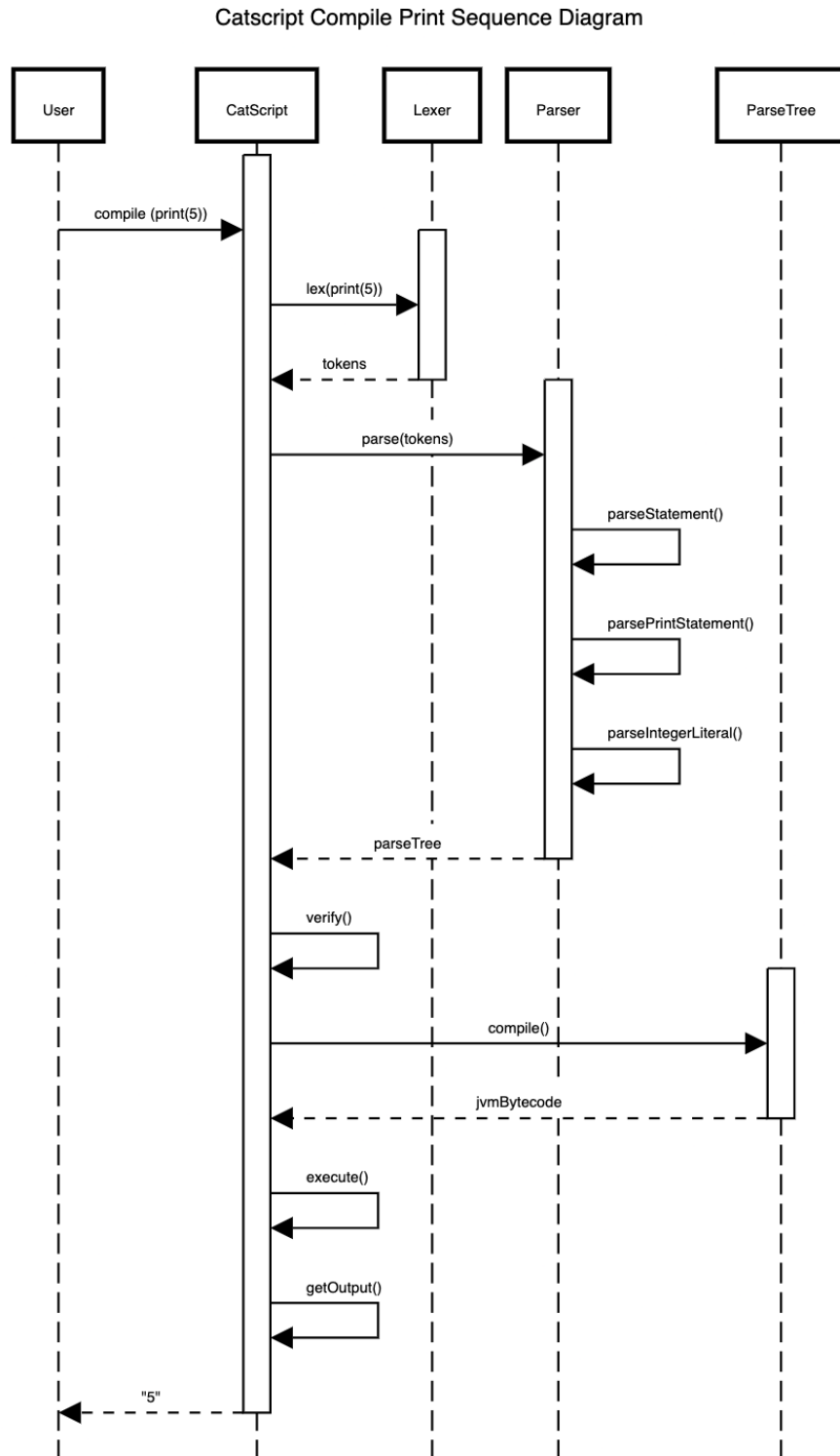
`*args` : CatScript expression(s)
0 or more values to be supplied to the function as arguments

Example

```
function foo(a : int, b : int) { print(a + b) }  
  
foo(2, 5)  
  
> 7
```

Section 5: UML

Attach the UML design diagrams for your capstone project that were created **before** you began coding your project.



Section 6: Design trade-offs

Describe a design trade-off decision (e.g. execution time vs. space requirements or compile time) in your capstone project and justify the design decisions that you made.

We decided to implement the parser by hand via recursive descent rather than through the use of a parser generator. Parser generators are very powerful, but complex, tools. While using a parser generator would have been instructive, we felt that we would gain a better understanding and intuition about Catscript and compilers in general if we implemented it with recursive descent.

Another design trade-off was how to handle separate elements within the parser, tokenizer, compiler, and evaluator. We could have used the visitor pattern to separate the functional aspects of each object. We decided against this and to instead use interfaces along with per-class implementations to allow us to handle elements similarly and keep the functionality of each class with the class itself. This made it much easier to debug the compiler and implement it while still maintaining a great deal of flexibility.

We also considered implementing the compiler in different languages, but we decided that Java fit all of our desired characteristics. Compiling to JVM bytecode would probably also be more difficult in non-JVM languages. The class was using Java as well, so it was not practical to use a different language.

Section 7: Software development life cycle model

Describe the model that you used to develop your capstone project. How did this model help and/or hinder your team?

We used test-driven development to implement the Catscript compiler. This made implementation much easier because the specifications of the compiler were laid out before we began implementing it, so we knew a given feature worked as long as the tests passed. The test suite was also very thorough and built upon the program in stages, so we were able to similarly build the compiler in stages. We were able to design and implement edge cases easily as well because we just had to confirm the edge cases passed with a quick test.

One downside to the test-driven development is that we had to have a clearer specification for each step of the compiler than if we didn't have tests to pass. The design phase could have been much more difficult and long, but our professor helped us greatly to speed up this process.