

Compilers: CSCI 468
Spring 2023
Griffin Austin
James Lucas

Section 1: Program

See [source.zip](#) in this directory for all of the source code for this project. The grammar of Catscript is as follows:

```
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

Team member 1 was the primary engineer. Team member 2 was the secondary and quality assurance engineer. Team member 1 was responsible for all source code other than the three tests in [edu.montana.csci.csci468.demo.Scratch](#). Team member 1 also created and populated this document.

Team member 2 was responsible for section 4 of this document as well as the three tests in [Scratch](#).

An estimation of the percentage of time spent by each member was as follows: 95% team member 1 and 5% team member 2.

Section 3: Design pattern

In [CatscriptType](#) the design pattern memoization was used. This block of code can be found at [edu.montana.csci.csci468.parser.CatscriptType](#) lines 36-44.

```

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 listType
}

```

The reason this pattern was used is because list types are often used several times each during compilation. If this design pattern was not used, a new instance of each type would be created and destroyed for each list object in the source code. With this design pattern, unnecessary object creation is avoided and performance is therefore increased. If a list type already exists in the map, its instance is used rather than a new object being created. Since fewer objects are created, performance is increased and memory usage is reduced. Overall, memoization was used to cache the results of the function call for increased performance on subsequent calls.

Section 4: Technical writing

Catscript Guide

This section serves as a high level guide to interacting with Catscript. Examples are provided where appropriate.

Introduction

Catscript is a simple scripting language. Here is an example:

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

which would output

```
foo
```

Features

For loops

For loops in Catscript function similar to those in other programming languages. A sequence of elements can be iterated over as follows:

```
var numbers: list<int> = [1, 2, 3];  
for (num in numbers) {  
    print(num)  
}
```

with output

```
1  
2  
3
```

If statements

The if statement is a conditional statement which allows for execution of different code blocks based on the specified condition. If the condition evaluates to true, the code within the first set of curly braces executes. If it is false, then the optional else block is executed.

```
if (7 * 2 == 0) {  
    print('true')  
} else {  
    print('false')  
}
```

which would output

```
false
```

Print statements

This is a type of statement that allows a program to print to the console during execution. An expression is passed as an argument and whatever it evaluates to is printed.

```
var x: int = 8  
print(x)
```

would output

```
8
```

Variable statements

This is how new variables are declared and initialized. This statement can optionally include a type expression that specifies the data type of the variable. If no type is specified, the variable is assigned to type `object` by default. The evaluated expression on the right of the equal sign is the variable's value.

```
var x: int = 15
```

In this example, a new variable `x` of type `int` is declared with value `15`. To declare a list, the following syntax is used

```
var y: list<int> = [1, 2, 3]
```

Function declaration

This is used to define a function. It consists of a function name, parameter list, optional return type, and body similar to other programming languages. Here is an example of a function declaration in Catscript:

```
function add(a: int, b: int): int { // add is function name, a and b are parameters  
  return a + b // body  
}
```

Return statements

This is used only in the body of function declarations. It allows a function to return a value to where it is being called. If an expression follows a return statement, whatever value the expression evaluates to is returned. Otherwise, the function will return null.

```
function example(x: int) {  
  if (x > 0) {  
    return 1 // 1 is returned  
  } else {  
    return // null is returned  
  }  
}
```

```
}  
}
```

Function calls

This allows for calling functions with optional arguments as follows:

```
add(1, 2)
```

would call function `add` with arguments `1` and `2`.

Assignment statements

This assigns a value to a variable that is already initialized.

```
var x: int = 10  
x = 15
```

`x` is assigned a value of `15`.

Function declaration

Define a new function with name, parameters with optional types, return type, and body.

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

Equality expressions

Compares two expressions for equality (`==`) or inequality (`!=`).

```
1 == 2 // false  
'foo' != 'bar' // true
```

Comparison expressions

Compares two expressions using `>`, `>=`, `<`, and `<=`.

```
1 > 2 // false  
5 >= 1 // true  
2 < 3 // true  
4 <= 4 // true
```

Additive expressions

Perform addition or subtraction on two expressions.

```
3 + 1 // 4
5 - 2 // 3
```

Factor expressions

Perform multiplication or division on two expressions.

```
3 * 2 // 6
10 / 2 // 5
```

Unary expressions

Perform operations (`not`, `-`) on a single expression.

```
-5 // negative 5
not true // false
```

Types

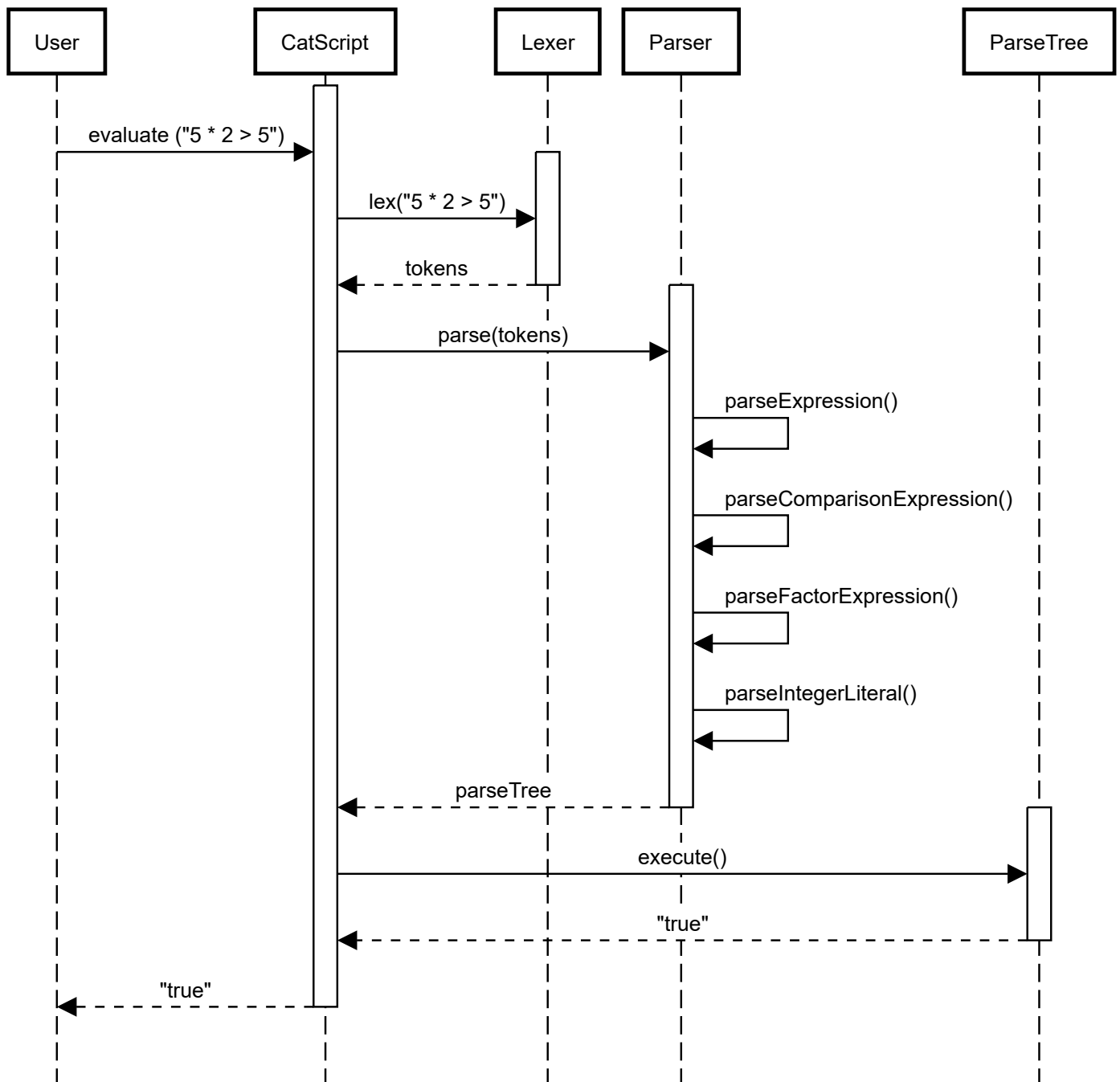
Catscript is statically typed with the following types:

- `int`: 32 bit integer
- `string`: Java-style string
- `bool`: boolean value
- `list`: list of values with the type `x`
- `null`: null type
- `object`: any type of value

Section 5: UML

The following diagram illustrates the process, at a high level, when a user inputs an expression into Catscript.

Catscript Multiplication and Comparison Sequence Diagram



Section 6: Design trade-offs

We created by hand a recursive descent parser instead of using a parser generator such as `lex` and `yacc`. The foremost reason for this was because this method was more intuitive and provided much more educational value than using a parser generator. While using parser generators would have greatly sped up development, it would have hindered the understanding we obtained by making one by hand.

Second, creating a parser gives us more fine-tuned control than using a generator. We are able to implement custom error handling and messages. The downside in this regard is the lack of robustness as a result of creating and debugging our own parser. While parser generators, especially those such as `lex` and `yacc` have much more extensive testing suites than our parser, so we can be more confident in the efficacy of parser generators. However, we were able to curtail this by utilizing test-driven development.

A large advantage of writing our own recursive descent parser is it reduces the dependencies in our program. There are no external dependencies to the parser we have written; this would not be the case if we used a parser generator.

Section 7: Software development life cycle model

We heavily utilized test driven development (TDD) as our SDLC model. Prior to implementing any code, we were provided an extensive test suite that outlined the expected functionality of each part of the compiler. By filling in the tokenizer, parser, evaluator, and bytecode generator, these tests were slowly solved.

This model forced us to consider what our code should do before actually writing it. It also allowed us to find bugs much earlier than if we used other methods. Any time a new piece of functionality was implemented, testing was immediately performed and relevant changes were able to be made.

TDD also gave us the confidence that our code worked as intended and likely had few to no bugs. On the other hand, there were a few instances of all tests passing, but the code still did not behave as expected. In this case, it became very difficult to determine where the code was faulty and thus how to remedy it. Overall, TDD proved to be a large help despite these challenges.

Alternatively, we could have used the waterfall model. This model requires prior stages of development to be completed before continuing to later stages. Usually this consists of requirement gathering, design, implementation, testing, and maintenance. Although this project could be neatly broken up into stages (tokenizer, parser, evaluator, and bytecode generator), there is no convincing reason this needs to be a linear process in our case. If we were to choose this model over TDD we likely would have experienced slower development and more difficult changes during later stages of the project as it becomes impossible to modify earlier stages.