

Computer Science Portfolio

CSCI 468 - Compilers

Spring 2023

Jaden Schultz

Megan Weide

Section 1: Program

Please see the directory called “source.zip” in the same directory this document is in.

Section 2: Teamwork

Team member one worked on the code aspect of this project. They wrote the code for the compiler and ensured the tests passed. Team member one put in 75% -80 % of the time in the project.

Team member two worked on the documentation for this project. They wrote documentation that gave examples of the various features of the Catscript language. Team member two put in 20% -25 % of the time in the project.

Section 3: Design Pattern

```
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;  
}
```

One design pattern I used in this project was memoization. I used memoization in the CatscriptType class in the getListType method (lines 37-44). I used a pattern instead of just coding directly because the constructor call for a list with a certain type is expensive. By using memoization, I increased the speed and performance of the compiler, therefore optimizing it a bit.

Section 4: Technical Writing

Section 4: Technical Writing

Introduction

Catscript is a simple, statically-typed programming language, which means that the types of any variables written in a Catscript program are known at runtime. This documentation serves as a guide to the Catscript language, and as such, it will go over the details of the language's type system, variable declaration, statements, and more.

Type System

Compared to some other languages such as Java, Catscript has a relatively simple type system.

`int` - a 32 bit integer

`string` - a java-style string

`bool` - a boolean value

`list` - a list of value with the type 'x'

`null` - the null type

`object` - any type of value

Reserved Words

Catscript has a small list of *reserved words*, which cannot be used as the names of variables or functions in programs, as those variable and function names serve other purposes in the Catscript language. A list of reserved words in Catscript is included below:

`else`

`false`

`function`

`for`

`in`

`if`

`not`

`null`

`print`

`return`

`true`

`var`

Comments

Comments in Catscript can be declared with a double forward slash (`//`).

```
30 // This comment has no impact on the functionality of the program.  
31 // However, using comments correctly can help make a program easier to follow.  
32
```

Declaring Variables

In Catscript, if you wanted to declare a variable `x` of the value `20`, you would type out `var x = 20`. “`x`” and “`20`” can be substituted for any name and value you’d like, but it’s important to note that variables can’t be named with any of the previously mentioned reserved words, nor can a variable and function with the same name coexist in a program (ex: you can’t have a variable named `bird` and a function named `bird` in the same Catscript program).

Variables in Catscript can also be declared with a type. An example of this is provided below:

```
var eggsInOmelet : int = 3
```

If Statement

An if statement checks if a given condition in the code has been satisfied. If it is, the instructions inside the brackets are executed. If the condition isn't satisfied, the program either executes the instructions written in an else statement, or it continues going through different conditions specified by else if statements.

Example of an if statement in Catscript. Note that unlike many other languages such as Java, Catscript doesn't require the use of semicolons in its grammar:

```
var dollars = 100
if (dollars <= 100) {
    print("You are broke.")
}
```

Example of an if-else statement:

```
var dollars = 20
if (dollars > 10) {
    print("It seems like you can afford lunch today.")
}
else {
    print("Maybe you should eat at home instead.")
}
```

Example of an if-else statement using else if:

```
var eggs = 4
if (eggs > 2) {
    print("You could make a hearty omelet with those eggs.")
}
else if (eggs == 2) {
    print("Seems like you'll still have room for a side.")
}
```



```
else {  
    print("Are you sure that's enough to eat?")  
}
```

For Statements

Like in many other programming languages, a for statement in Catscript iterates through the instructions written within the brackets as long as the conditions of the for loop are being met. However, for loops in Catscript are a bit more limited in their functionality, as they require an object of type `list` in order to work. This limitation can be seen in the `validate()` function in the `ForStatement` expression class of the Catscript compiler:

```
if (type instanceof CatscriptType.ListType) {  
    symbolTable.registerSymbol(variableName, getComponentType());  
} else {  
    addError(ErrorType.INCOMPATIBLE_TYPES, getStart());  
    symbolTable.registerSymbol(variableName, CatscriptType.OBJECT);  
}
```

Despite this, for statements in Catscript are still very useful:

```
var instances = [1, 2, 3, 4, 5]  
for(y in instances) {  
    print("A cryptid has been spotted!")  
    if(y == 5) {  
        print("You've seen enough today. Please go home and  
        get some rest.")  
    }  
}
```

While lists can be declared separately from the for loop in Catscript, they can also be declared as a part of the for statement itself.

Print Statements

Print statements in Catscript are rather straightforward, as they only require `print()` to be called. The statement will output whatever is written inside of it, as long as the types match. In the case of strings, double quotes ("") are required within the print statement, or else the program will throw an error.

```
print("This is a print statement. Nothing to see here")
```

Declaring Functions

In Catscript, the `function` keyword is used to declare functions. A function in Catscript consists of the aforementioned keyword followed by the function's name and list of parameters. A return type may be used, but it isn't required. After that, the function body is contained within brackets. Inside of the function, variables and statements such as the ones above may be used.

```
function iWonderWhatsForDinner(x) {  
    print(x)  
}
```

Calling Functions

Outside of a function, it may be called elsewhere in the program by passing a corresponding parameter or multiple parameters through.

```
iWonderWhatsForDinner(1)
```

Function Parameters

In Catscript, the parameters of a function call may be declared with either their names, or with their names along with their respective types.

```
function iWonderWhatsForDinner(x) {  
    //function body  
}  
  
function doIKnow(x: bool) {  
    //function body  
}
```

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 } ']';

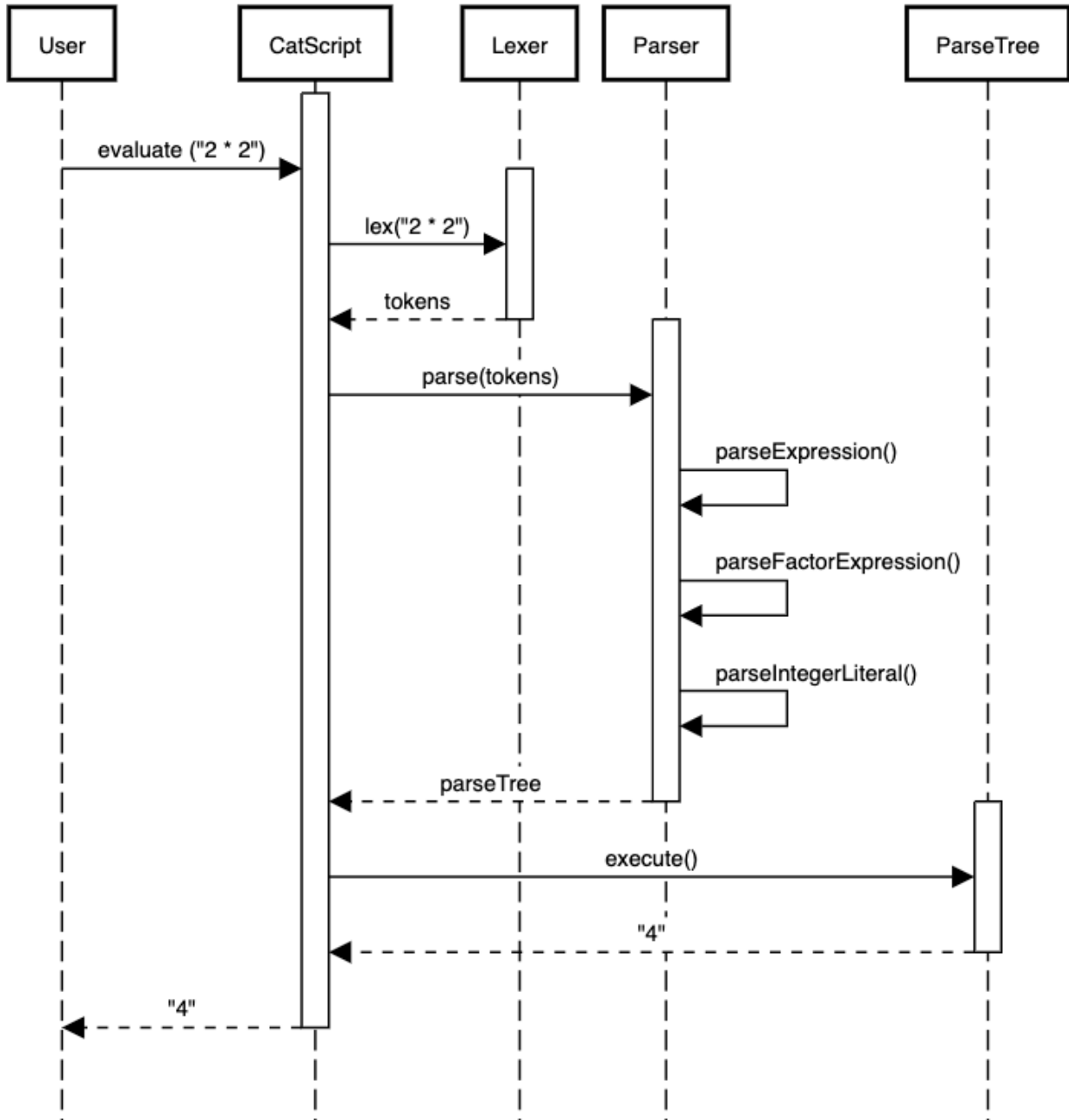
function_call = IDENTIFIER, '(', argument_list, ')'

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

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

Section 5: UML

Catscript Multiplication Sequence Diagram



Section 6: Design Trade-offs

One design trade-off was creating a parser by hand, rather than by using a parser generator. We decided it would be more intuitive to use a recursive descent algorithm as it would teach us exactly how grammars work in more in-depth than using a tool like Lex or Yak to generate a grammar for us.

Section 7: Software Development Life Cycle Model

For our software development life cycle model, we did test-driven development. We were given a test suite that documented how the language should work and fixed the code so that it passed those tests. This allowed us to have a bit of freedom over how we implemented the code itself. As long as we got the tests passing, that meant that the programming language was functioning as expected. That is all that mattered to us. We want the language to work and function properly, and to implement it in whatever way we see best fit.