CSCI468 Compilers

Spring 2024

Elyse Dalager | Mason Watamura

Section 1: Program

The source code zip file is included in this directory as source.zip.

Section 2: Teamwork

Team member 1, myself, worked on 100% of the source code implementation of the recursive descent parser for the CatScript scripting language. Team member 2, Mason Watamura, was responsible for 100% of the documentation for the CatScript language and three tests. The documentation can be found in Section 4: Technical writing. The tests are included in the test/java/edu/montana/csci/csci468/demo/PartnerTests.java directory.

Section 3: Design pattern

The Memoization Pattern was used to memoize the getListType() function in main/java/edu/montana/csci/csci468/parser/CatscriptType.java. This pattern was used to optimize the execution of the parser by storing known list types in a hash map called cache, so we don't store elements and their respective data types more than necessary. The execution of the parser is faster with this technique and frees up memory to sufficiently implement the Memoization Pattern.

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

Section 4: Technical writing

CatScript Guide

This document should be used to create a guide for CatScript, to satisfy capstone requirement 4.

Introduction

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

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

Features

The CatScript programming language is a statically typed programming language, and has a small type system as follows. The CatScript programming language must begin with a program statement, which can either be a statement or a function declaration.

- 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

Statements

Statements are a group of expressions or other statements that either will or will not complete an action.

For Statement

For loops are a control flow statement that allows you to iterate through a sequence of elements, like lists. They will continue to execute through the block of code while there are still elements to iterate through.

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

If Statement

If statements are control flow statements that allow you to conditionally execute certain blocks of code. It allows the program to make decisions and execute different branches of code based on what conditions have or have not been met.

```
var x = 10
```

```
if(x == 10){
   print("true")
}
```

```
if(x == 10){
   print("true")
}
else {
   print("false")
}
```

```
if(x == 10){
   print("true")
}
else if(x < 10){
   print("less than")
}
```

```
if(x == 10){
   print("true")
}
else if(x < 10){
   print("less than")
}
else {
   print("greater than")
}
```

Print Statement

Print statements allow you to display certain output on a screen. You may encase any single value expression within a print statement.

```
print(1)
print("Hello World!")
print(true)
```

Variable Statement

Variable statements are declarations also used to create a storage location in memory that holds data. CatScript does not require variables to be declared with types, but you may choose to do so anyway.

```
var x = 1
var y = null
var bool : x = true
var string : z = "Hello World!"
```

Function Call Statement

Function call statements invoke the execution of a function. They may or may not return a value.

```
functionCall(var1, var2)
```

Assignment Statement

Assignment statements are used to assign a value to a variable. This allows you to store and manipulate data within the program.

```
x = 10
```

Function Declaration

Function declarations define new functions. You can define its input parameters, if any, and the body of the function will be executed when run.

```
function funcDeclaration (int x, int y) {
   if (x > y) {
      return(true)
   }
   return(false)
}
```

Function Body Statement

Function body statements define the body of the function. This is the block of code that gets executed when the function is called. For example, this is the function body of the function declaration above:

```
if (x > y) {
    return(true)
}
return(false)
```

Parameter List

Parameter lists are part of function declarations that list the parameters that a function accepts, along with their types. A parameter list may be as long or as short as you like. For example, this is the parameter list of the function declaration above:

```
(int x, int y)
```

Parameters

Parameters consist of an identifier and a type expression to define the inputs a function takes. You may name the identifier whatever you want, but the type expression needs to be one in the CatScript type system. For example, this is the first parameter of the funcDeclaration() function above:

```
x : int
```

Return Statement

Return statements are used to return a value within a function. This will terminate the execution of the function. This is the return statement of the funcDeclaration() function above:

```
return(false)
```

Expressions

Expressions are combinations of values, variables, operators, and function calls that are evaluated to produce a single result.

Equality Expression

Equality expressions evaluate whether two values are equal or not. You may use equality operators to produce a boolean return value.

- Equal: ==
- Not Equal: !=

```
10 == 10
10 != 11
```

Comparison Expression

Comparison expressions are used to compare two values. You may use comparison operators to determine the relationship between operands and produce a boolean value.

- Less than (<)
- Greater than (>)
- Less than or equal (<=)
- Greater than or equal (>=)

```
10 < 15
15 > 10
15 <= 15
15 >= 15
```

Additive Expression

Additive expressions involve the addition or subtraction between operands. You may add or subtract values from one another and get a value in return depending on the operation. The Addition operand can also work for string concatenation.

```
1 + 1
1 - 1
"Hello" + "World!"
```

Factor Expression

Factor expressions involve the multiplication or division between operands. You may multiply or divide values and get a value in return depending on the operation.

```
2 * 3
```

```
4 / 2
```

Unary Expression

Unary expressions involve only one operand or value and an operator. These operate on a single operand like "not" or "-".

```
-1
not true
```

Primary Expression

Primary expressions refer to the simplest form of an expression. It represents a single value or operand without any additional operators or nested expressions.

```
12, false, "Hello", foo(), [1, 2]
```

List Literal

List literals specify a collection of elements when defining or initializing a list. The lists may only store items of the same data type.

```
[1, 2, 3]
[true, false, false]
```

Function Call Expression

Function calls are expressions that invoke the execution of a function. This is where you may pass in predefined arguments in order to manipulate data, print something, or get a return value.

```
x = 10
y = 9
funcDeclaration(x, y)
```

Argument List

Argument lists are part of function calls that list the arguments that a function accepts. An argument list size may only be the amount of parameters defined in the function. For example, this is the argument list passed into the funcDeclaration() function in the Function Call Expression example above:

```
x, y
```

Type Expression

Type Expressions are used to specify or represent data types. They allow you to define the data type of variables, expressions, or values.

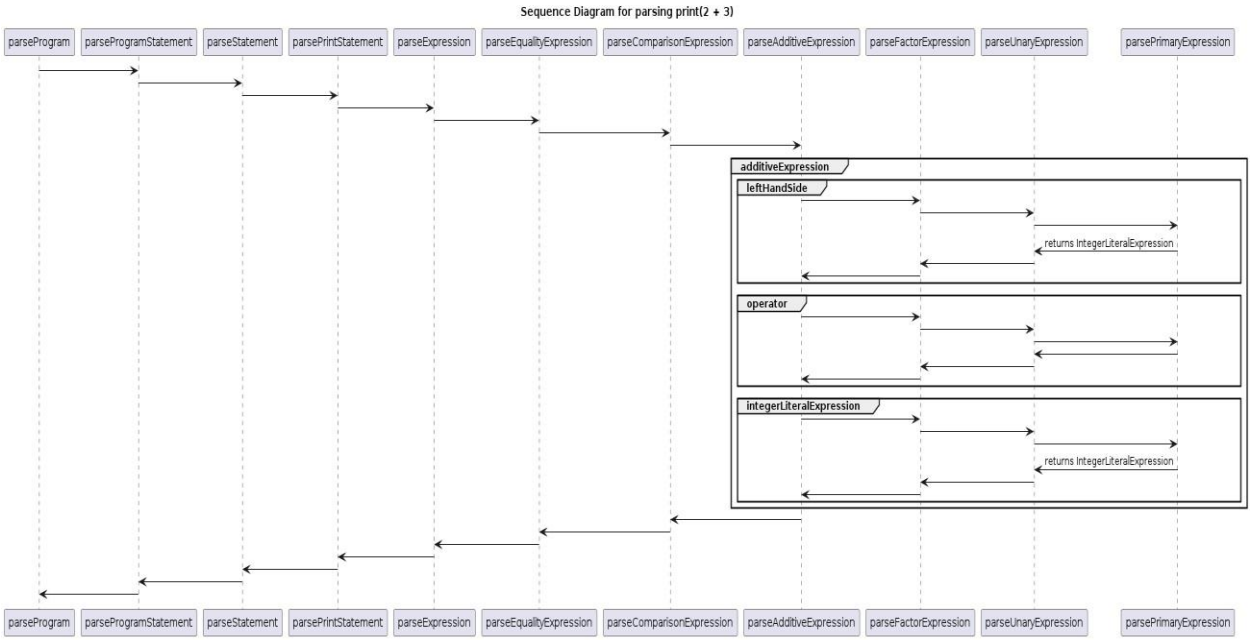
```
int
object
string
object
list
```

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



Section 6: Design trade-offs

An alternative approach to creating a parser would be a parser generator. Parser generators are programs that take a grammar and grammar rules to code generate a language. Typically, it takes a lexical grammar containing Regular Expressions and a language grammar using Extended Backus-Naur Form (EBNF) as its two inputs. One may use tools like lex (a lexer generator) and yacc (yet another compiler compiler) to aid in this concept of method generation. A common parser generator in use today is known as Another Tool for Language Generation (ANTLR), which is very popular in the Java community.

In contrast, a recursive descent parser is a simple and obvious approach to creating a parser. It begins by taking an EBNF grammar that outlines the rules and regular expressions of the language. Developers are responsible for creating a method for each rule within the grammar, then calling each other method defined on the right-hand side of the production and matching strings as needed. The GNU Compiler Collection (GCC), C#, and Python compilers all use this approach to parsing!

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project. TDD is a software development methodology that requires tests to be written before the code is implemented. Its goal is to ensure that the code is running as expected and meets any specified requirements. TDD aided in building this recursive descent parser by creating an efficient and accurate way to test the code; the test base was immediately available to test any progress in the parser's implementation. It also establishes expectations defined by each test to ensure the code is working properly and behaves as expected. For these reasons, I enjoyed using Test Driven Development to implement a recursive descent parser.