CSCI 468 Compilers

Capstone

Mason Watamura : Elyse Dalager

Teamwork

I worked with Elyse Dalager during the project. I implemented my end of the compiler, taking care of tokenization, parsing, evaluating, and compiling the code into byte code. Towards the end, Elyse wrote the technical writing section in this writeup and provided more tests to test the parser. These tests were tests that maybe were not thought about when using the base tests of testing the different portions (tokenizer, parser, etc.) while completing the compiler.

Tests given to test other features:

```
8
    MasonWatamura
9   public class PartnerTests extends CatscriptTestBase {
10
        MasonWatamura
11      @Test
12      public void parseEqualityAdditiveExpression() {
13          EqualityExpression expr = parseExpression( source: "(1 + 1) == (0 + 2)");
14          assertTrue(expr.getLeftHandSide() instanceof ParenthesizedExpression);
15          assertTrue(expr.getRightHandSide() instanceof ParenthesizedExpression);
16          assertTrue(expr.isEqual());
17      }
18
        MasonWatamura
19      @Test
20      public void parseListLiteralAdditiveExpression(){
21          ListLiteralExpression expr = parseExpression( source: "[[0, 1], [2]]");
22          assertEquals( expected: 2, expr.getValues().size());
23          ListLiteralExpression innerList = (ListLiteralExpression) expr.getValues().get(0);
24          assertEquals( expected: 2, innerList.getValues().size());
25      }
26
        MasonWatamura
27      @Test
28      public void parseUnaryExpression() {
29          UnaryExpression expr = parseExpression( source: "not not not true");
30          assertTrue(expr.isNot());
31          assertTrue(expr.getRightHandSide() instanceof UnaryExpression);
32      }
33  }
```

Design Pattern

The design pattern that we used was the memoization pattern. Memoization is a type of caching that is used in dynamic programming and helps to improve the performance of the code, or in my case, the compiler. This is used in the CatscriptType file when we memoize the getListType function call. We used the pattern instead of coding directly because we wanted to avoid creating multiple instances of ListType for the same type. Creating multiple instances of ListType for a type is 'wasteful' in the main sense that you should only need one instance of ListType for a type.

This is the code that was implemented to memoize the getListType function

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

A CatScript program is statically typed and begins with a program statement, which contains either a statement or function declaration. Its type system is outlined below.

```
- 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 the individual commands or instructions that make up the program.

For loops

For loops are control flow statements that allow you to iterate through a sequence of elements, like lists. They repeatedly execute a block of code until there are no more elements to iterate through.

```
var list = [0, 1, 2]

for( i in lst ) {
    print(i)
}
```

If statements

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("Hello, world!")
```

Variable statement

Variable statements are declarations 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 intExample = 1
var bool : boolExample = true
```

Function call statement

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

```
foo()
```

May return:

```
Hello, world!
```

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 = 0
```

Function declaration

Function declaration statements are used to define new functions. This is where you can also define its input parameters, if necessary, and the body of code to be executed when it's called.

```
function bar(x : int, y : int) {
    var answer = x + y
    return(answer)
}
```

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. This is the body of the bar() function in the function declaration example:

```
var answer = x + y
    return(answer)
```

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. These are the parameters of the bar() function in the function declaration example:

```
x : int, y : int
```

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. This is the first parameter of the bar() function in the function declaration example:

```
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 bar() function in the function declaration example:

```
return(answer)
```

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 in order to produce a boolean return value. They may be used in if statements.

- Equal (==)
- Not equal (!=)

```
if(x == 10){}
```

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. They may be used in if statements.

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

```
if(x >= 10){}
```

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.

```
1 + 1
```

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 * 2
```

Unary expression

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

```
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.

```
false
```

List literal

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

```
[1, 2, 3]
```

Function call

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.

```
var x = 10
var y = 11
bar(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. These are the arguments of the bar() function in the function call example:

```
x, y
```

Type expression

Type expressions are used to specify or represent data types. They allow you to define the data type of variable, expression, or value.

```
int
```

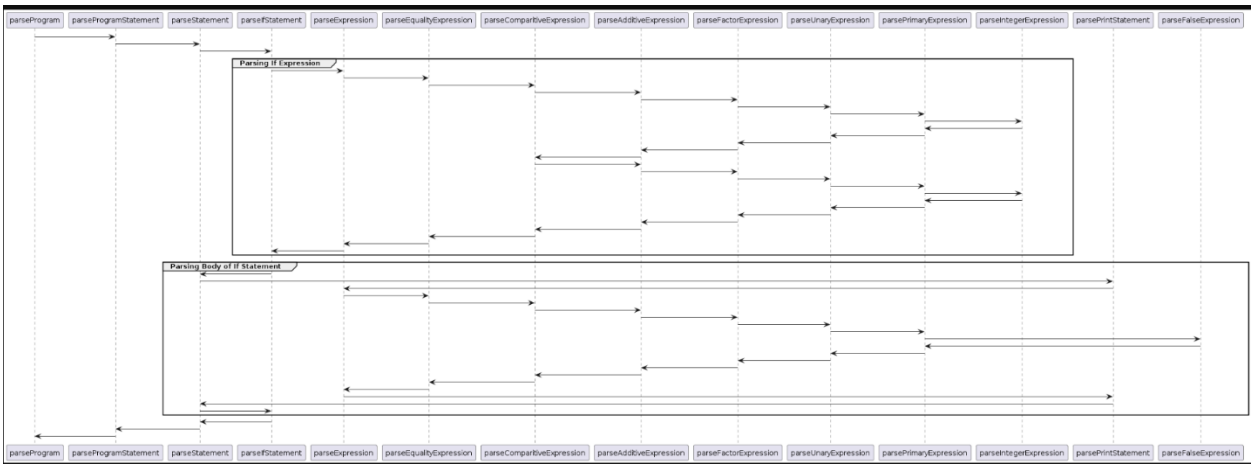
CatScript Grammar

```
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, '>']
```

UML Diagram

if (9 > 10) { print(false) }

Sequence Diagram for Parsing an If Statement



This sequence diagram shows the process and the functions used to parse the above given if statement. We start off by parsing the program which starts to go down the recursive descent tree. We look for the IF token that was tokenized and start trying to parse both the expression being checked by the if statement, and the body of the if statement. The expression can be seen as we work our way down the recursive descent methods to looking for the integer that we need to parse. After finding the integer, we start going back up looking for the comparative token to be parsed, only to go back down for the next integer to be parsed.

Once the expression is parsed, we need to parse the body of the statement. We start by parsing another statement, which will be the print statement. We then go down recursive descent again looking to parse the boolean primary expression. Once the expression is parsed we work our way back up, but need to return to the parse if statement function because the if statement method calls the parseStatement method. Once back to the if statement we can parse the closing brace and finish parsing the full statement.

Design Trade Offs

For this project, we could have gone one of two ways to write the compiler. We could have used parser generators, or we could have used recursive descent. We chose to use recursive descent for its simplicity and ease for the implementation. It also can generate parse trees for the input very easily. One of the tradeoffs of choosing recursive descent is that there are limitations if we wanted to use left recursive grammars. There would also be limitations to handling ambiguous grammar.

This compares to recursive descent for a couple of reasons. The main reason is that trying to tokenize expressions and statements would have been much more complicated. When picking up this project using recursive descent, we had to first grasp what was going on during the tokenization tests. We could easily use a debugger and pinpoint when and where different actions were happening and be able to read the code around it to know why it is happening in that way. Parser generators have a lot more code when trying to tokenize and when trying to debug to understand what that code is doing throughout that process, you can get lost very quickly. This leads to a slower process of getting the project done, especially under a time crunch.

Software Development Life Cycle Model

Throughout the development of this compiler, we implemented a test driven development process to test the different features that we were making. First, we tested that expressions and statements would tokenize. This consisted of making sure all the tokens could be recognized and whitespace could be ignored when it needed to be. Then we moved onto making sure the parser we were writing could parse those tokens. We parse the tokens in the way that we wanted the syntax to appear. This is also where recursive descent was really started to be implemented since we had to start off by trying to parse a statement and go down the descent from there. Next we tested that the evaluations of the difference expressions and statements were able to be evaluated after being both tokenized and parsed. Multiple different tests were made for different cases, making sure that errors were also caught if, say, an if statement did not have a closing brace. Finally, we had to

make sure the code that was inputted could be compiled into byte code. Multiple tests that were used to test the evaluation of input code could be reused but had to check that the code compiled still produced the same output as it did when ran through evaluation.