Capstone Portfolio

CSCI 468 - Compilers
Montana State University
Spring 2024

Zachary Carmean & Jamison Cleveland

Section 1 Program

The source.zip file has been submitted alongside this PDF in the capstone/portfolio folder.

Section 2 Teamwork

This project is a collaboration between Jamison Cleveland and Zach Carmean working to create a CatScript parser for the CatScript scripting language utilizing the recursive descent algorithm to create the underlying functionality within the scripting language. Zach worked on making sure Expressions, Statements, Evaluation, and Bytecode within CatScript functioned properly and compiled in order to create CatScripts functionality. Starting with the tokenization of the lexemes to create tokens for the parse tree which was then used to create the expressions that were used within each of the statements. Jamison then contributed 3 tests for the CatScript scripting language that would test the functionality within certain aspects of the language. Jamison also wrote the technical documentation for the CatScript parser including information about all expressions and Statements within the CatScript scripting language.

The technical documentation written by Jamison also includes the grammar involved in the CatScript language showing how each of the Expressions and Statements work at a lower level. In terms of overall team collaboration with the Compiler being made by Zach and tests and documentation of the CatScript language made through Jamison the work for this compiler is split evenly between the two. Individually each partner worked on their own separate compiler and did the documentation for it throughout the duration of the project.

Section 3 Design Pattern

One design pattern within the CatScript scripting language that I am going to be outlining is the memoization design pattern which will allow us to ensure that certain methods will not execute more than once or multiple instances of a method aren't created when given the same instance of a type such as having multiple lists of integers. We used this design pattern for the CatScript type system in CatscriptType.java to assist in reducing the amount of computations happening within CatScript by cashing previously used types or results within the CatScrip system, so we would be able to just pull these results and types from memory rather than just creating a new instance of them saving the CatScript type system from having to perform unnecessary computation. This design pattern makes the CatScript type system much faster in terms of overall performance in compiling. Below is an example of the memoization algorithm used within the CatScript scripting languages type system:

```
public static HashMap<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;
}
```

Section 4 Technical Writing

Introduction

Catscript is a simple, imperative scripting language with a C/Java style syntax. It contains expressions, standard control flow statements, and recursive functions. It also has a small type system for primitives and includes a built-in list type.

Here is an example:

```
function describe(x: int): string {
    var abs = x
    var sign = "positive"
    var size = "small"

    if (x < 0) {
        sign = "negative"
        abs = -x
    }
    if (abs > 100) {
        size = "large"
    }
    return "a " + size + ", " + sign + " integer"
}
print(describe(42)) // --> "a small, positive integer"
print(describe(-200)) // --> "a large, negative integer"
```

Features

Each of the following syntactic features will be described, as well as their productions within the Catscript grammar. The productions are given in EBNF syntax. The recursive descent parser is based on the grammar specification given.

Equality Expression

The equality expression returns a boolean that determines whether the two objects are equal. Equality expressions use `==` and `!=` as operators.

```
0 != 1
```

Equality expression production:

```
equality_expression = comparison_expression { ("!=" | "==")
comparison_expression };
```

Comparison Expression

The comparison expression compares two values using operators such as `<`, `>`, `<=`, and `>=`, returning a boolean result.

```
42 < 100
```

Comparison expression production:

```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" )
additive_expression };
```

Additive Expression

The additive expression performs addition or subtraction between two values using the `+` and `-` operators.

```
2 + 2
```

Additive expression production:

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Factor Expression

The factor expression performs multiplication or division between two values using the `*` and `/` operators.

```
3 * 6
```

Factor expression production:

```
factor_expression = unary_expression { ("/" | "*" ) unary_expression };
```

Unary Expression

The unary expression applies a unary operator, such as negation (`-`) or logical NOT (`not`) to a single operand.

```
-42
```

```
not true
```

Unary expression production:

```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```

Identifier Expression

The identifier expression represents a variable or constant identifier in the program.

```
foo + 2
```

Literal Expression

The literal expression represents a literal value, such as a number, string, boolean, or a null pointer.

```
[80, null, "string", true]
```

Productions for list literals:

```
list_literal = '[', expression,  { ',', expression } ']';
```

Function Call Expression

The function call expression invokes a function with the specified arguments.

```
2 + fib(22)
```

Function Call Expression production:

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

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

Print Statement

The print statement outputs a value or a string to the console. If the object is a string, it prints it as it is, otherwise, it will be the string representation of the value.

```
print("hello") // --> hello
print(null)    // --> null
print(4)       // --> 4
```

Print Statement production:

```
print_statement = 'print', '(', expression, ')'
```

If Statements

The if statement allows conditional execution of code blocks based on a condition. They can be chained like the following example:

```
if (x < 10) {
    print("small")
} else if (x < 100) {
    print("medium")
} else {
    print("big")
}
```

The `else` clause of an if statement is optional and can be excluded, like in this example:

```
if (a == b) {
    print("is equal")
}
```

It also creates a new local scope.

`if` statement production:

```
'if', '(', expression, ')', '{', { statement }, '}' [ 'else', ( if_statement |
'{', { statement }, '}' ) ];
```

For Statement

The for statement allows iteration over a list of values. The sequence given must be a list. Unlike many other languages, there is no `while` statement, nor is there extra control flow primitives like `break` or `continue` in `for` statements. Like the `if` statement, the `for` statement creates its own local scope.

```
print("countdown!")
for (x in [5, 4, 3, 2, 1]) {
    print(x)
}
```

`for` statement production:

```
'for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}';
```

Variable Statement

The variable statement declares a new variable with an initial value. Unlike other languages, variables cannot be declared uninitialized, and an initial value must be provided. An optional type can also be provided, which must match the inferred type of the initial expression.

```
var a = 2 + 2
var b: int = a * a
print(b + a)
```

Variable statement production:

```
'var', IDENTIFIER, [':', type_expression, ] '=', expression;
```

Assignment Statement

The assignment statement assigns a value to a variable. The variable must be within scope prior to the assignment. The type of the expression on the right hand side must also be assignable to the type of the variable.

```
var a = 0
if (x < 100) {
    a = 10
}
print(a)
```

Assignment statement production:

```
IDENTIFIER, '=', expression;
```

Function Definition Statement

The function definition statement defines a new function with parameters and a body. Its parameters and return value can be annotated with a type. It can also have return statements within the body that can cause an early return. It also has its own local scope.

```
function greet(name: string) {
    print("hello, " + string)
}
```

Function Definition statement production:

```
parameter_list = [ parameter, {',' parameter } ];
parameter = IDENTIFIER [ , ':', type_expression ];
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' + [ ':'
+ type_expression ], '{',  { statement },  '}';
```

Return Statement

The return statement exits a function and optionally returns a value. Must be within a function declaration. The return value must match the type annotation on the function declaration.

```
function measure(x: int): string {
    if (x < 10) {
        return "small"
    } else {
        return "big"
    }
}
```

Return Statement production:

```
return_statement = 'return' [, expression];
```

Function Call Statement

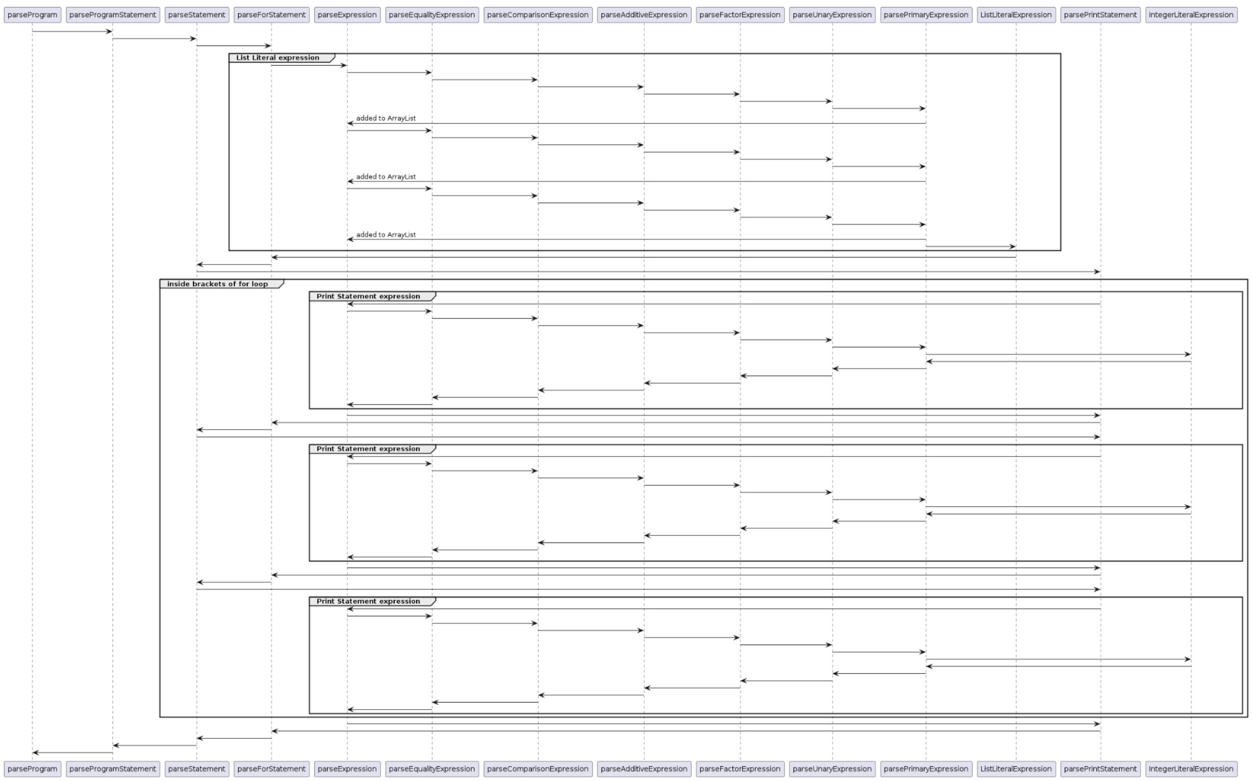
The function call statement invokes a function with the specified arguments. Any return value from the function call is discarded.

```
measure(30)
```

Function Call statement production:

```
argument_list = [ expression , { ',' , expression } ]
function_call = IDENTIFIER, '(', argument_list , ')'
```

Section 5 UML



This UML diagram showcases a sequence diagram for how the CatScript scripting language handles the given for loop, "for(x in [1, 2, 3]{ print(x) })". This gives good insight on how CatScript handles not only for loops, but also showcases CatScripts ability to deal with list literal expressions as well as print statements. This diagram also outlines both the expression parsing and the functionality inside the for loop showing the parsing of the print statement in CatScript when inside a for loop. This diagram starts at the parseProgram function then goes into the parseStatement functions before reaching the parseExpression functions for the expressions within the for loop.

Section 6 Design Trade-offs

The CatScript scripting language uses a recursive descent parser to parse the grammars within it rather than a parser generator. Recursive descent gives the CatScript language better readability in terms of knowing what is happening at any step throughout the parsing process. Given the grammar of the CatScript scripting language, a parser generator would make the parsing much more complicated than it needs to be since CatScripts grammar is not an extremely big or complicated one and can be done faster and with better understanding through a recursive descent algorithm. Because CatScript has such a small grammar, doing the parsing by hand using recursive descent is overall a better design choice that allows for a better understanding of the recursive nature of grammars.

A parser generator gives the developer no control over the parse elements and will often create very lengthy unreadable code which leads to confusion epically when attempting to debug. Parser generators also take away a lot of debugging functionality that is given to us through the recursive descent parser. Although recursive descent requires more writing than a parser generator, the readability in a recursive descent parser is crucial when developing high-quality code. Recursive descent ability to handle more complex grammars for better flexibility plays a major factor in why it is becoming a more popular method for parsing.

Section 7 Software Development Life Cycle Model

Our compiler was developed over the course of the semester using test driven development which is just a development method that utilizes the use of tests for our code helping guide our project in the right direction. Test driven development allowed us to become more comfortable using a debugger to understand exactly what the test wanted us to accomplish. It also was a way to give us more insight on how exactly a compiler works and what it may be like to work within a bigger code base. More specifically our CatScript language was developed using test-first driven development meaning we had gotten the tests first before writing any sort of code. This can be a bad thing if a developer doesn't know exactly how their program is going to operate which is why it is typically better to test driven development after some code has been written.

I prefer test driven development over other development methods because I believe it creates a better sense of what needs to be done among everyone working on a project getting rid of any sort of confusions before they develop between developers. It also can be a way of seeing just how far along you are in a project's production and can make dividing the tasks among others much easier. Test driven development also guarantees that under certain circumstances your code will always work which means any problems will be cases that are not covered by the tests. This brings up a problem that may come up with this style of development which is that you really only know how your code will operate under the tests you are given, making a developer blind to any potential problems that may arise as they write their code. Some developers may also just force tests to work through hard-coding which will present problems in the future of that software product. Overall I believe test driven development is a great development method to use when developing software by making things easy to understand and work with.

Partner Tests

```
@Test
    public void parseComplexExpression() {
        AdditiveExpression expr = parseExpression("not true + \"asdf\"");
        assertEquals(true, expr.isAdd());
        assertTrue(expr.getLeftHandSide() instanceof UnaryExpression);
        assertTrue(((UnaryExpression)
expr.getLeftHandSide()).getRightHandSide() instanceof
BooleanLiteralExpression);
        assertTrue(expr.getRightHandSide() instanceof StringLiteralExpression);
    }

    @Test
    public void parseFunctionDeclaration() {
        FunctionDefinitionStatement expr = parseStatement("function f(x: int) :
int {return x + 1}");
        assertNotNull(expr);
        assertEquals("f", expr.getName());
        assertEquals(1, expr.getParameterCount());
        assertEquals("x", expr.getParameterName(0));
        assertEquals(CatscriptType.INT, expr.getParameterType(0));
        ReturnStatement returnStmt = (ReturnStatement) expr.getBody().get(0);
        assertNotNull(returnStmt);
        assertTrue(returnStmt.getExpression() instanceof AdditiveExpression);
        assertTrue(((AdditiveExpression)
returnStmt.getExpression()).getLeftHandSide() instanceof IdentifierExpression);
        assertTrue(((AdditiveExpression)
returnStmt.getExpression()).getRightHandSide() instanceof
IntegerLiteralExpression);
    }

@Test
Void basicFunction() {
      assertEquals("75\n", executeProgram("function f(n : int) : int { return n
+ 42} print(f(33))"));
}
```