Montana State University Computer Science Department

“Catscript” Capstone Portfolio

Compilers 468

Spring 2024

Thompson, Trey

Flinn, Kaitlin

Section 1: Program. The source for my compiler is available in the same directory as this document.

Section 2: Teamwork.

In this section, Team Member 1 was in charge of coding and Team Member 2 was in charge of documentation and unit tests. Member 2's documentation can be found in section 4, technical documents.

Team Member 2's Unit Test:

```
//Kait's Tests
new *
@Test
public void MixedListTypes(){
    VariableStatement expr = parseStatement( source: "var x = [1, true, \"hello\"]");
    assertNotNull(expr);
    assertTrue(expr.getExpression() instanceof ListLiteralExpression);
}
new *
@Test
public void ObjectIsAssignable(){
    assertEquals( expected: "1\n", executeProgram( src: "var x : object = 1\n" +
            "print(x)"));
    assertEquals( expected: "hello\n", executeProgram( src: "var x : object = \"hello\" \n" +
            "print(x)"));
    assertEquals( expected: "true\n", executeProgram( src: "var x : object = true \n" +
            "print(x)"));
}

new *
@Test
public void pemdasCheck(){
    assertEquals( expected: 20, evaluateExpression( src: "1 * 4 + 8 * 2"));
    assertEquals( expected: 8, evaluateExpression( src: "2 + 2 * 3" ));
    assertEquals( expected: 9, evaluateExpression( src: "5 + 2 * 6 / 3" ));
    assertEquals( expected: 6, evaluateExpression( src: "30 - 7 * 3 + 1 - 8 / 2" ));
}
```

Section 3: Design pattern.

In our implementation, we've integrated the memoization pattern to optimize the efficiency of retrieving a ListType for a given CatscriptType during variable initialization, like in the statement "var x : int = [1, 2, 3]". This pattern efficiently stores previously computed ListType objects in a cache, represented by a HashMap. In addition, multiple requests for the same CatscriptType can be swiftly retrieved in the cached result instead of redundantly computing it. This strategic use of memoization optimizes performance by minimizing function calls, resulting in faster execution times and an improved algorithm.

```
3 usages
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
3 usages  Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
    if(cache.containsKey(type)) {
        return cache.get(type);
    } else {
        ListType listType = new ListType(type);
        cache.put(type, listType);
        return listType;
    }
}
```

Section 4: Technical writing

Team Member 2 Documentation:

Catscript Guide

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

Introduction

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

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

Features

Assignment Statement

```
x = 10
```

The assignment assigns the integer value 10 to the variable x.

For Loops

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

This loop will execute three times, each time printing the value x holds at that time. It will first print 1, then 2, then 3 before terminating

Function Definition Statement

```
function example_function(int i) {}
```

The function, example_function, is declared with the keyword function and takes in an integer.

Function Call Statement

```
example_function(2)
```

Now the function, example_function, is being called, passing in the integer 2.

If Statement

```
if(x > 10){ print(x) } else { print( 10 ) }
```

These two statements work together to form a conditional sequence. If the integer x is evaluated larger than 10, the print(x) statement will execute. Otherwise, if x is smaller than 10, print(10) will execute.

Print Statement

```
print("hello")
```

The print statement will print the word "hello".

Return Statement

```
function x() {return}
```

Here, the return statements ends the execution of the x( ) function.

Variable Statement

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

X is being declared as type int, with a value of 10 assigned.

Additive Expression

```
1 + 2
3 - 1
```

The additive expression evaluates both sides of the
operator, then determines if the expression is addition or subtraction, then evaluates.

Boolean Literal Expression

```
bool = true
```

The boolean literal expression holds a value of true or false.

Comparison Expression

3 < 5

The comparison expression evaluates both the left and right hand sides of the operator, then determines if the sign is greater, less, greater than or equal to, or less than or equal to. The result is a boolean value of true or false.

Equality Expression

1 == 2 (false)
1 != 2 (true)

The equality expression evaluates both sides of the == operator to determine if they are equal. The result will be a boolean value of true or false.

Factor Expression

2 * 2
10 / 5

The factor expression evaluates both sides of the operator, then determines if the expression is multiplication or division, then evaluates.

Function Call Expression

example_function(1)

The function call expression calls a function that already been declared.

Identifier Expressoin

name

The identifier here is the word 'name', which is an identifier since there are no registered types, expressions or statements that are called 'name'.

Integer Literal Expression

x = 5

An integer literal holds an integer value.

List Literal Expression

list<int> = [1, 2, 3]

A list literal holds a series of numbers. They can be of any literal type, with any number of variables.

Null Literal Expression

x = null

The null literal expression will hold the value of null, not dissimilar to representing nothing.

Parenthesized Expression

2 * (1 + 2)

Following arithmetic rules, the addition will be evaluated before the multiplication since it is in parentheses.

String Literal Expression

name = "trey"

The string literal holds a string, or word, value. In this case, the value of trey is held in the name variable.

Syntax Error Expression

"foo(1, 2",
UNTERMINATED_ARG_LIST

Errors represent mistakes in the code. In this example, there is no closing parentheses to the parameter list. This is a problem as the parser doesn't know when the end of the parameter list has been reached.

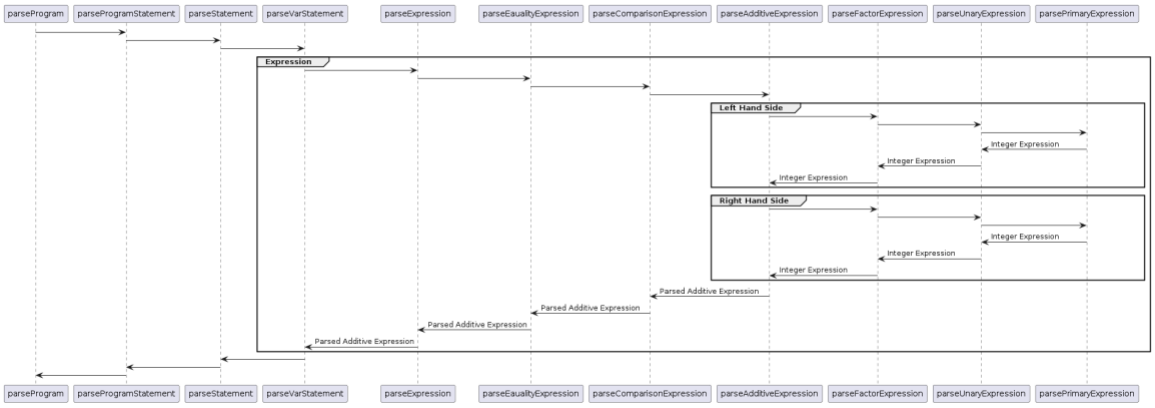
Unary Expression

true != false (true)
if((not) 5)

The unary expression negates the equality expression. The exclamation point before the equal represents that terms
must not be equal to be true.

Section 5: UML. No UML was needed or used in this project because the overall design was pre-determined by the professor so we could focus on parsing.

Below is a Sequence Diagram for the parsing of the Additive Expression, “var x = 2 + 2” in the Catscript language.



The diagram demonstrates the recursive descent of parsing. As you can see, it starts in a wide scope and starts the program. Then we begin to parse statements within the program. Our statement is variable so once this is recognized, the parsing of the expression begins. Once the language recognizes that it is an additive expression, it forks into 2 different expressions—one for the left-hand side and one for the right-hand side. As the program returns an integer expression, the recursive ascent begins to complete the parsing of the program.

Section 6: Design trade-offs.

With the starting code provided at the outset of the project, our team embarked on enhancing the existing recursive descent algorithm. Recursive descent was a good fit for our project due to the structure of our grammar. The grammar of Catscript builds on itself. Recursive descent allows this grammar to transition seamlessly between statements and expressions. While recursive algorithms are more complex in terms of Big O notation, this approach allowed us to accurately reflect the language's grammar. All in all, building our first complier we were willing to sacrifice optimization to promote clarity and structure.

Section 7: Software development life cycle model.

For our capstone project, we decided to use the Test-Driven Development (TDD) approach, which isn't usually popular due to concerns about its flexibility. However, TDD worked really well for us. When it comes to parsing and evaluating, every detail counts. TDD helped us execute these processes perfectly, ensuring our language ran smoothly. Although others worry that TDD can lead to a lot of rewriting and sticking too strictly to a plan, we found it saved us time and helped us avoid headaches later on. This is true since we developed parsing before evaluating. If the parsing was not correct, then our team would have spent hours trying to fix a parsing error in the evaluation code. By focusing on writing tests before writing the actual code, we caught problems early and gained a deeper understanding of our project.