Compilers CSCI-468, Spring 2024: Simon Collins and Matthew Rolls

Section 1: Program

The source files for the program are included in a .zip file named “source.zip”.

Section 2: Teamwork

This Capstone project was worked on primarily independently. Approximately 95 percent of the work was done by team member 1 to implement the parser. The remaining 5 percent is where project team member 2 contributed documentation and 3 additional program tests in the “PartnerTest.java” file. The documentation contributed by team member 2 provides an overview of the features and functionality of the Catscript language. The additional testing added tests for type checking and expressions for the program. Specifically the first test verified that a previously declared string variable can not be assigned to an explicitly assigned int variable. The second test verifies that a function of return type string cannot return the value provided by a function of return type int. The third test verifies that an additive and factor expression can have their resulting values compared while still retaining their instance of respective expression type.

Section 3: Design pattern

A design pattern implemented in the project is the Memoization design pattern. In this implementation the purpose of memoization is to remember the results of an input to the function. In this case the input is a CatscriptType named type inputted to the function “getListType”. The function is located at line 36 of the “CatscriptType.java” file. This implementation of memoization ensures that there will only be one ListType of an individual Catscript type. To do this the code uses a static hashmap named cache to cache list types. When the function is called it will take the input Catscript type and see if there is a corresponding key in the hashmap cache. If there is a matched key it will return that entry in the cache. If there is no match a new entry in the cache will be created using the input as the key and a new ListType created with the input type as the value and then the new entry will be returned. Without the use of this design pattern a new instance of ListType with the input Catscript type would be created regardless if one already existed. Included below is an image of the described code.

```java
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = new ListType(type);
    if (cache.containsKey(type)){
        return cache.get(type);
    }else {
        cache.put(type, listType);
        return cache.get(type);
    }
}
```

Section 4: Technical writing

Catscript Guide

Introduction

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

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

Features

For loops

For loops allow for iteration over lists. The variable defined in the for argument will be updated with the next item in the list each iteration. Furthermore, a body can be defined inside the brackets of the for loop made up of statements which will be executed each iteration. The following code snippet will iterate through each value in the list `[1, 2, 3]` and print that value to the console.

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

Output:

```
1
2
3
```

If statements

If statements take an expression that evaluates to either `true` or `false`.

If `true` the statements contained within the body of the `if` statement will be executed.

Optionally, an `else` statement can be included immediately following the body of the `if` statement, containing code enclosed in brackets that will be executed in the event the expression evaluates to `false`.

The snippet of code below demonstrates this behavior. If the value of `x` is greater than 1, the then code `print("x is greater than 1")` will be executed, whereas if the value of `x` is less than 1, then the code `print("x is less than 1")` will be executed instead.

```
if (x > 1) {
    print("x is greater than 1")
} else {
    print("x is less than 1")
}
```

Print statements

`print()` statements offer the ability to output a string included in the parentheses of the print statement to the console.

For example:

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

Var statements

Var statements allow for the definition of variables. Var statements also support type inference, but alternatively, an explicit type can be given in the definition. Furthermore, a variable that has already been defined via a var statement can be set to a new value of the same type.

```
var x = 1
x = 2
var y : string = "Hello, World!"
```

Assignment Statements

An assignment statement consists of an identifier, or the name of a variable, followed by the assignment operator and a value. If the variable has previously been defined in a var statement, then the new value being assigned to the variable must be of the same type of the variable. If the variable has not already been defined in a var statement, it must be defined before it can be assigned a value.

For example:

```
var x = 1
print(x)
x = 2
print(x)
```

Output:

```
1
2
```

Function Definitions

CatScript also allows for function definitions. Function definitions must include at least a function name. Optionally, function definitions can include a parameter list. Each parameter must have a unique symbol and must be comma separated from other parameters. Parameters may also include explicit types. Functions may contain return statements and explicit return types may be included with those return statements to limit type conflicts. The body of the function definition can contain code that is executed each time the function is called.

A function with no parameters and no function body:

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

A function that takes three parameters and prints each one:

```
function y(a, b, c) {
    print(a)
    print(b)
    print(c)
}
```

A function that takes two parameters of type `int` and returns an `int`:

```
function z(a : int, b : int) : int {
    return a + b
}
```

Function Calls

To call a function in CatScript, all that is required is the function name, and a list of arguments corresponding to the function definition.

For example, to call the following function, a string and an int must be passed as arguments respectively:

```
function foo(a : string, b : int) {
    print(a + " costs $" + b + ".")
}

foo("bar", 7)
```

Output:

```
bar costs $7.
```

Expressions

Equality Expressions

Equality expressions evaluate to `true` or `false` according to the expressions given on either side of the operator as well as the operator itself. This operator can either be the `==` equality operator that evaluates to `true` if the left and right hand sides are equal or the `!=` inequality operator that evaluates to `true` if the left and right hand sides are different.

The statement:

```
print(1 == 1)
```

Outputs:

```
true
```

The statement:

```
print(1 != 1)
```

Outputs:

```
false
```

Comparison Expressions

Comparison expression evaluate to `true` or `false` according to the expressions given on either side of the operator as well as the operator itself.

The `>` operator will evaluate to `true` if the value on the left-hand-side is greater than the value on the right-hand-side.

The `>=` operator will evaluate to `true` if the value on the left-hand-side is greater than or equal to the value on the right-hand-side.

The `<` operator will evaluate to `true` if the value on the left-hand-side is less than the value on the right-hand-side.

The `<=` operator will evaluate to `true` if the value on the left-hand-side is less than or equal to the value on the right-hand-side.

For example:

```
1 > 1
```

evaluates to `false`, whereas

```
1 >= 1
```

evaluates to `true`.

Additive Expressions

Additive expressions containing the `+` or `-` operators will add or subtract the integer values on the left and right-hand-sides of the operator. If one of the values on either side of the operator is not of type `int` but instead evaluates to type `string` then the right-hand-side will be concatenated to the left-hand-side if the `+` operator is used.

For example:

```
1 + 1
```

evaluates to:

```
2
```

And:

```
1 + true
```

evaluates to:

```
"1true"
```

Factor Expressions

Factor expressions containing the `*` or `/` operators will multiply or divide the integer values on the left and right-hand-sides of the operator. CatScript does not support floating point numbers, therefore, any factor expressions that do not evaluate to an integer will be rounded down to the nearest integer.

For example:

```
1 * 2
```

evaluates to:

```
2
```

And:

```
1 / 2
```

evaluates to:

```
0
```

Unary Expressions

CatScript supports two unary operators: `not` and `-`. The `not` operator when prefixing a boolean literal expression will flip the value of that boolean. The `-` operator when prefixing an integer literal expression will change the sign of that integer.

For example:

```
not true
```

evaluates to:

```
false
```

And:

```
-(1 + 1)
```

evaluates to:

```
-2
```

Primary Expressions

Primary expressions consist of identifiers, function calls, and all literal expressions, such as `true`, `"Hello, World!"`, and `null`.

For example the refrence to the variable `x` in the print statement below is a primary expression.

```
var x = 1
print(x)
```

The function call on the second line of code below is also a primary expression:

```
function foo() {}
foo()
```

Lists

Lists in CatScript are typed and immutable. For loops can be used to iterate through a list.

For example, the following list is a list of objects containing a string, an int, and another list of component-type int.

```
var x : list<object> = ["one", 2, [3, 4]]
```

Types

CatScript supports the following types:

`int` : an integer number. i.e. `1` .

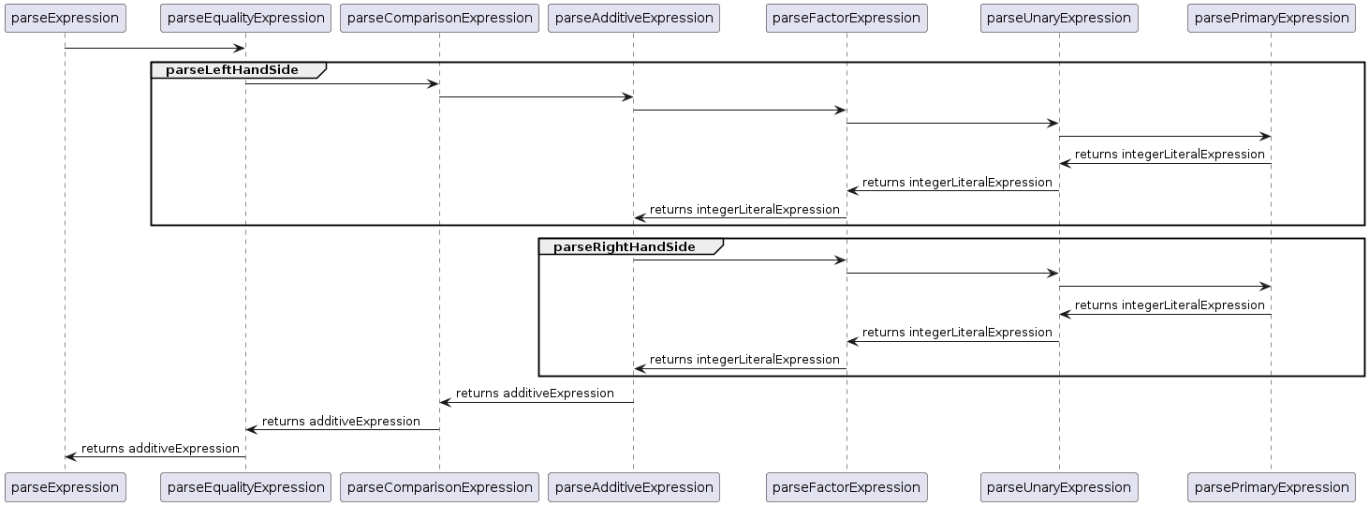
`string` : a string of characters including no characters enclosed in double quotes. i.e. `"Hello, World!"` .

`bool` : a boolean value of either `true` or `false` .

`list<type>` : a list where the components of the list are assignable to the type specified in declaration. i.e. `list<int>` .

`object` : a super type that is assignable from all other types.

Section 5: UML.



The UML sequence diagram above shows a sequence diagram of the parsing of the expression “1 + 1”. As the design of the code uses recursive descent when parseExpression is called when examining the current token, or left side, parseEqualityExpressions is called and so on until the bottom is reached at parsePrimaryExpression. Here the “1” in “1 + 1” is recognized and the current token is consumed and an integerLiteralExpression “1” is returned back up the chain until it hits parseAdditiveExpression where the current token of “+” is recognized and consumed. Inside of parseAdditiveExpression the right side is sent back down the chain until the bottom at parsePrimaryExpression where the “1” token is recognized consumed and an integerLiteralExpression is sent back up the chain to parseAdditiveExpression where a new additiveExpression is constructed with the “+” operator and two integerLiterals “1” and “1” representing the left and right hand sides. With no more tokens in the input to form an additional right hand side to the new returned expression the additiveExpression is returned all the way back up to parseExpression.

Section 6: Design trade-offs

This project implements a recursive descent parser. A possible alternative could be a parser generator, this design is actually quite common at the university level. Recursive descent however, is widely used in the computer science industry. Where recursive descent parsers are built by hand based on a given grammar language parser generators will take a language and generate a parser for the given language. This generation is great because it is quick and requires significantly less code and infrastructure to accomplish the task, but it has the drawback of less control and understanding of the inner workings of the parser. The parser generator also does a fine job of teaching what a parser accomplishes, however it does not teach the ‘how’ of how it accomplishes the parsing task. The parser generator’s popularity at the university level makes sense because at the university level you are generally working on short term projects that you will abandon after you have completed the course and because it requires less code it can be a more practical option given the available time. In the real world the drawback of time needed to code the more infrastructure heavy recursive descent parser is worthwhile because in the long term you have more control over it and it can be crafted to your needs. Additionally, recursive descent if done properly can be faster and can have better error handling.

Section 7: Software development life cycle model

This project used test driven development. This model uses tests to verify that the code is outputting the expected result. For this project specifically a large number of tests were used to verify that results of various inputs matched the expected results according to the rules of the Catscript grammar. Each individual test often handled a small rule, for example, a for statement would require a closing parenthesis and this specifically would be tested. The earlier tests for parts such as the tokenizer test at a micro level, but eventually later tests depend on earlier features functioning properly so the tests in a way approach a more macro level. This model helped significantly in the development of the project as it allowed the person writing the code to easily be able to quickly test newly implemented code. If the new code did not generate the expected result you could do a quick comparison between the result and the expected result to see where it went wrong. Test driven development especially for this project was a fantastic model to use because the project works on an established grammar, so the expected results are generally set in place from the beginning. Because of this chosen model and project, a clear system of milestones in the form of checkpoints were able to be set to work through the project. Without this model developing the code for the project would have been a disorganized mess. One small mistake early on could have massive repercussions later on, so catching mistakes early with testing was incredibly helpful.