Catscript

Jamison Cleveland, Zachary Carmean

Gianforte School of Computing, Montana State University

CSCI 468: Compilers

Mr. Carson Gross

Spring 2024

Section 1. Program

Attached as a zip folder.

Section 2: Teamwork

Jamison Cleveland will be referred to as Team Member 1 and Zachary Carmean will be referred to as Team Member 2. Team Member 1 provided the implementation for the parser and parts of the lexer and interpreter, satisfying the requisite tests. Team Member 2 provided 3 additional tests, as well as writing the technical documentation for the program. The estimated percentage of time taken for the project was 90% for Team Member 1 and 10% for Team Member 2.

Section 3: Design Pattern

One design pattern used in this project was memoization. Memoization is a technique where a result of a computation is stored so that when the same input is given, the result can be given without recomputing the value, which makes the program more efficient. This pattern was used in the method `getListType` in the class `CatscriptType`. The method stores the list type of the given component type into a hash table if not already present, then returns the resulting list type. This prevents multiple instances of list types of the same components from being constructed.

```
private static final Map<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

The CatScript Programming Language

Introduction

CatScript is a simple high level scripting language utilizing various expressions and statements to provide code that looks like the sample below:

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

CatScript uses an algorithm known as "recursive decent" to parse all the tokens into a parse tree allowing us to execute and compile our expressions and statements creating the basis for the CatScript language. The CatScript languages contains the following data types: int, string, bool, object, null, and void; it is also capable of handling lists of all types and parsing them correctly.

Features

Expressions

CatScript contains a number of expressions which gives it the ability to do basic mathematical computation, dispute the difference between the different data types within expressions, handle parenthesized expressions, etc...

Additive Expression

The Additive Expression handles the addition and subtraction computation within CatScript as well as any additive string concatenation. It does this by getting the tokens and their data type then sorts them into a parse tree creating a left hand side and a right hand side for the tree. It does this by taking each side of the parse tree and pushing it on a stack then pushing the operator onto the stack and popping it, computing it, then pushing the solution back onto the stack.

```
1 + 1   addition            or      1 + 2 + 3...
2 - 1   subtraction         or      2 - 1 + 7...
```

The grammar for the Additive Expression just requires a factor_expression followed by the PLUS or MINUS sign followed by another factor expression:

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

Boolean Literal Expression

The Boolean Literal Expression handles all boolean expressions by taking the boolean value and setting its CatScript type to 'BOOLEAN'. These expressions will also validate this value using a symbol table then return the boolean value during execution.

```
true
false
```

Comparison Expression

The Comparison Expression deals with operators such as (<, >, <=, >=) and works in a similar way to the Additive Expression in that it will parse each of the tokens and get there token types then put them into a parse tree with the expression becoming the root and what's being compared becomes the right and left hand side of the parse tree. It will then push everything onto a stack with the expression at the top of the stack, pop everything, evaluate the expression, then return a boolean (true or false).

```
1 < 2   true
2 < 1   false
3 >= 4  true
4 <= 3  false
3 >= 3  true
```

The Comparison Expressions grammar only requires the operators between two additive expressions:

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

Equality Expression

The Equality Expression handles cases when a double equals (==) and not equals (!=) expression is present working in a similar way to the Comparison Expression, but only using the double equals and not equals as the root of the parse tree after tokenization. the left and right values of the parse tree are then pushed onto a stack with the root on top, popped, evaluated then the Equality Expression returns a boolean (true or false)

```
1 == 1  true
1 == 2  false
1 != 2  true
```

The Equality Expressions grammar only requires the operators between two comparison expressions:

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

Factor Expression

The Factor Expression handles the multiplication and division computation within CatScript; it does this by getting the tokens and their data type then sorts them into a parse tree creating a left hand side and a right hand side for the tree. It does this by taking each side of the parse tree and pushing it on a stack then pushing the operator onto the stack and popping it, computing it, then pushing the solution back onto the stack.

```
3 * 5   multiplication      or      3 * 5 / 2...
4 / 8   division            or      4 / 2 * 3...
```

The Factor Expressions grammar only requires the operators between two unary expressions:

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

Function Call Expression

The Function Call Expression works with the functions within the CatScript scripting language. This introduces basic function functionality to it allowing it to take in parameters with or without specific types such as int and bool. This Expression takes in all arguments and validates them within a symbol table then gets the type for all parameters. During execution all parameters will be placed into a list and be invoked on runtime.

```
function foo(){}
function foo(a, b, c){}
```

The Function Call Expression grammar requires an Identifier and an argument list:

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

Identifier Expression

The Identifier Expression contains all the types for each variable, list, or function assuming it does not fall within the standard identifiers such as int and string etc... This expression only acts as an IDENTIFIER within the CatScript scripting language

Integer Literal Expression

The Integer Literal Expression is how CatScripts assigns the types for integers and allows CatScript to use "toString()" methods on integer values. This Expression will always return an integer with the type IntegerLiteralExpression after it has received it as a token.

```
1, 2, 3, 4, ...
```

List Literal Expression

The List Literal Expression will take a list of an assigned CatScript type then create a list of expressions as a linked list. Using a symbol table to validate, it will then get the list type through the first item in the List Literal Expression, always returning a new list when executed.

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

Null Literal Expression

The Null Literal Expression handles all cases where the null value is involved. When any null value is put through the CatScript parser this Expression will always assign that value as null and return null.

```
null
```

String Literal Expression

The String literal Expression, much like the previous Literals, handles all string cases within the CatScript scripting language. It does this by taking a string value as a token and returning that token back as a string for the CatScript scripting language to use.

```
"this is a string"
or
var x = "this"
```

Unary Expression

The Unary Expressions within CatScript handles all cases with negative values such as "-1" or the "not true/false" value, and it uses a right leaning parse tree to deal with these values. It will type check each expression on the parse tree then return either the boolean value, integer value, or null value.

```
-1
not true
```

The Unary Expression grammar is simple in that it requires a "not" or "-" followed by a unary expression or a primary expression:

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

Statements

With statements we start getting into the standard functionality of CatScript exploring how variables are assigned to if statements and how for loops are implemented and what the CatScript syntax will look like for each of these.

Assignment Statement

The Assignment Statement within CatScript allows for variable assignment; this means expressions within the scripting language are able to be given variable names. This also allows for any CatScript user to reuse these variables later in their own code for that expression. Here is an example on how you would assign a variable statement within the CatScript scripting language:

```
var x = "foo"
```

This example sets "x" as the variable name using the keyword "var" and sets the string "foo" as the variable assigned to our recently made variable x. What this statement looks like in the grammar is as follows:

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

meaning CatScript must start with an Identifier followed by an equal sign followed by expression for the CatScript assignment statement to validate.

For Statement

The For Statement is CatScripts basic implementation for a for loop in which it will loop for a set amount of iterations using the syntax outline (for _ in _). Everything within the for loop will then be added as a part of a linked list acting as the body of the for loop. The CatScript for loop is able to check if there is a duplicate name stored within the symbol table and will throw an error if one is found. The for loop also handles type checking within the symbol table for each of the variables used by the for loop. The for loop will iterate through the loop for the set amount of time or until it has reached the end of any list being iterated on. The way a for loop is set up in CatScript is as follows:

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

The example above shows a for loop in CatScript iterating through a list of integers giving an example of how iteration within CatScript will be handled. More specifically looking into the grammar of the CatScript programming language we can see more of what is required for the For Statement in CatScript to validate.

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

Function Definition Statement

The Function definition Statement within CatScript is how functions are primarily set up using the start word "function" succeeded by the function name and any parameters that may go along with it. This is how CatScript is going to define its functions to perform specific tasks; an example of how this would look with a function without given parameters would be something like this:

```
function foo(){}

function foo(){print("Hello World")}
```

This shows how the CatScript Function Definition Statement would be without any given parameters to it. It would still be able to perform anything within the brackets, but looking at an example where we give it a few parameters:

```
function foo(a, b, c){}
```

This is how we could achieve this just by separating any parameters with commas. Another thing we are able to do with the Function Definition Statement is add specific types to each parameter such as:

```
function foo(a : int, b : string, c : bool){}
```

With this we would need each of the parameters defined above to match the data types associated with the arguments within this function. Within CatScript function parameters are not the only thing we are able to set a specific type to. We can set a return type to the function by adding what the return type should be after setting the arguments of the function Definition:

```
function foo() : int { return 0 }

function foo(a : int, b : int, c : int) : int { return 0 }
```

Once the function has been made within the CatScript language all that needs to be done to call it is state the Function Definition name within our program by saying "foo()". CatScripts Function Definition Statement has a lot of functionality making it very malleable within the CatScript scripting language with the grammar also being extensive as shown below:

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

If Statement

The If Statement within CatScript introduces the use of conditional branching for more functionality. This statement within CatScript is fairly simple to set up with the syntax outline for this statement being (if _ then _) and an else statement added onto that if needed. An example of this in the CatScript scripting language would look like this:

```
if(x > 0){ print(x) }
```

As stated earlier CatScript also contains an else statement to catch any other conditions not caught by the initial if statement. With that added to the if statement CatScript now has:

```
if(x > 0){ print(x) } else { print("no") }
```

Looking at the grammar given through the CatScript scripting language:

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

For an if statement to validate within this scripting language an expression between two parentheses is required followed by bracket, statement, bracket and an optional else statement that goes by the same format.

Print Statement

The CatScript Print Statement allows for this scripting language to print out contents when compiled. An example of how this would look in the CatScript language is:

```
print()
```

The keyword 'print' is all that is needed to be able to print something; CatScript takes the token within the print method then gives that token a type and assigns it to a value putting it into a symbol table. The print statement within CatScript is able to print anything such as integers, strings, booleans, etc... An example of it printing some of these data types would be as follows:

```
print(1)
print("hello")
print(true)
```

The grammar behind the print statement is also made simple as shown:

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

Showing that only the keyword 'print' is needed to be followed by an expression.

Return Statement

The Return Statement within CatScript will always return a value with the data type specified within the Function Definition Statement. The syntax for the Return Statement with CatScript is like many other languages:\

```
return x
```

In the case above x is acting as a holder that is replaced with whatever we need returned. The grammar for the return statement is the keyword 'return' followed by an expression:

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

Variable Statement

The Variable Statement in CatScript is how we are able to assign variables to values we need in CatScript using the keyword "var" followed by what we want our variable name to be then an equal sign and the value of the variable. Within the CatScript scripting language this would look like this:

```
var x = 1
or
var x = "six"
```

In CatScript multiple variable types such as strings, ints, etc... are able to be assigned as variables. Optionally a type expression can be added to the Variable Statement making the variable one only of that type:

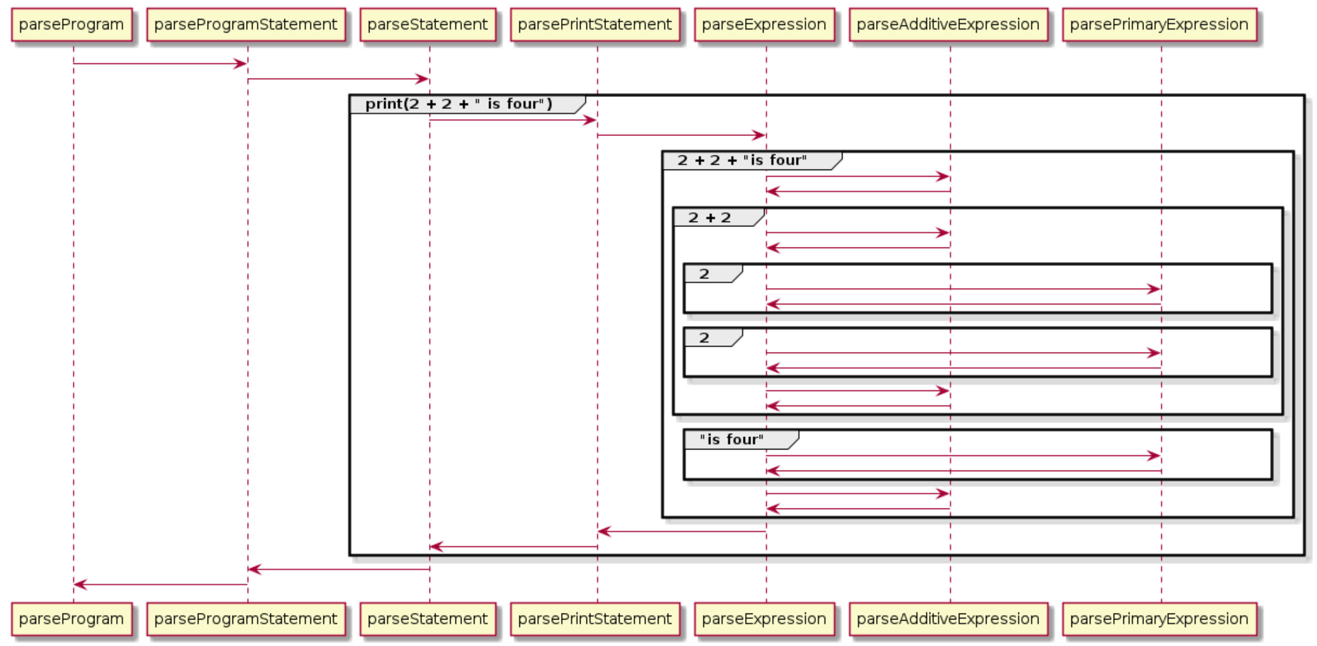
```
var x : int = 2
or
var x : string = "example"
```

This is proven through the CatScript grammar which shows that we start with the keyword 'var' then an identifier and an optional type expression followed by an equal sign and the expression:

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

Section 5: UML

The following sequence diagram depicts the parsing of the statement `` `print(2 + 2 + " is four")` ``



Section 6: Design trade-offs

One design trade-off we made was to use recursive descent parsing instead of a parser generator. Parser generators are programs that take a specification of a grammar and creates a parser for it, whereas recursive descent is implemented by calling recursive functions to handle the recursion within the grammar. Parser generators have the advantage that the code is written for you, but the code generated is generally difficult to read, and the grammar specification tends to not have built-in debugging. On the other hand, recursive descent parsers require you to handle the parsing yourself, which automatically makes it debuggable, but has the drawback that you have to handle implementation details such as preventing stack overflows and left-recursion. We ultimately chose recursive descent because the productivity gained from a debuggable parser outweighs the time saved from having the parser written for ourselves.

Section 7: Software Development Life Cycle Model

For this project, we used Test Driven Development (TDD). TDD prescribes creating tests for the design first, then writing the code until all tests pass, then refactoring. This approach helped us be productive as the tests would give us an early warning if there was an issue in the code, as well as giving convenient entry points for debugging particular parts of our program.