CSCI 468 Compilers Capstone Portfolio
Spring 2025

Stephen Brook

Section 1: Program

The source code for this project is in the same directory as this portfolio, zipped and labeled as source.zip

---

Section 2: Teamwork

Team Member 1:
This team member served as the primary developer for their CatScript compiler. They were responsible for implementing all core components following test-driven development principles. Their work began with the creation of a tokenizer to handle lexical analysis, converting raw input into tokens. They then implemented a recursive descent parser to construct parse trees for expressions and statements. Finally, they developed the compilation phase, which translated the parse trees into Java Virtual Machine (JVM) bytecode, enabling CatScript execution. IN addition to their own compiler, they wrote test cases for Team Member 2's compiler and produced language documentation for inclusion in Team Member 2's portfolio. Estimated time contribution: ~40 hours (80% of total project time).

Team Member 2:
This team member focused primarily on validating compiler functionality by developing a comprehensive suite of tests for Team Member 3's implementation. Their test cases addressed a wide range of scenarios including normal, boundary, invalid, and edge cases, ensuring effective error handling and clear error reporting. They also authored CatScript documentation for Team Member 3's portfolio. Additionally, they developed their own CatScript compiler, for which Team Member 1 provided tests and documentation. Estimated time contribution: ~5 hours (10% of total project time).

Team Member 3:
This team member contributed by designing and implementing test cases to validate Team Member 1's compiler. Their tests covered a range of inputs, including invalid, boundary, and erroneous cases, to verify correctness and robustness. They also wrote CatScript documentation for Team Member 1's portfolio. Like team member 2, they developed their own CatScript compiler, supported by tests and documentation provided by a teammate. Estimated time contribution: ~5 hours (10% of total project time).

Section 3: Design Pattern

The compiler utilizes the memoization design pattern, specifically, implemented in Catscript.java located inside the parser directory. This design pattern optimizes performance by minimizing redundant computations through caching.

In CatscriptType.java, memoization is applied in the getListType(type) method. Before creating a new list type, the method first checks the HashMap cache. If a cached result exists, it is returned immediately, avoiding unnecessary computation. Otherwise, a new list type is created, stored in the cache, and used for subsequent calls, ensuring that any unnecessary computation will be avoided in the future.

```
static HashMap<CatscriptType, CatscriptType> MEMOIZATION_CACHE =
     new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
     CatscriptType existingType = MEMOIZATION_CACHE.get(type);
     if(existingType != null){
          return existingType;
     }
     else{
          ListType listType = new ListType(type);
          MEMOIZATION_CACHE.put(type, listType);
          return listType;
     }
}
```

By reducing redundant object creation and computation, this design pattern enhances efficiency, lowers memory usage, and improves overall performance, especially for repeated function calls with identical inputs.

Section 4: Technical Writing

The technical writing for this section was created by Team Member 3, and offers comprehensive documentation of the CatScript programming language.

CatScript Language Documentation

Introduction

Catscript is a simple scripting language which utilizes static typing, compiles to JVM bytecode, and uses syntax best described as a hybrid between JavaScript and Python.
Here is an example:

```
var x = 7
print(x)
Output:
     7
```

Grammar

CatScript grammar defines the structure of the language, and defines rules for statements, function definitions, function calls, expressions, etc. The grammar reflects recursive descent parsing, which was implemented in this project. Consequently, most statements or expressions call other statements or expressions in their uses.

Features:
CatScript contains the following literals: int, string, boolean, list, and null.

Statement:

```
Regular expression:
     statement = for_statement |
     if_statement |
     print_statement |
     variable_statement |
     assignment_statement |
     return_statement |
     funtion_call_statement;
```

Throughout CatScript, statement is used to refer to the list above. As such, whenever ‘statement’ is mentioned in the technical specification of another statement or expression, it can refer to any one item listed above.

Expression:
Regular expression:

```
     expression = equality_expression
```

Throughout CatScript, expression is mentioned heavily in the technical specifications of both statements and expressions. Note that that expression is really an equality_expression, which is really a comparison_expression, etc. This pattern continues as follows: expression, equality_expression, comparison_expression, additive_expression, factor_expression, unary_expression, primary_expression. As such, whenever expression is mentioned in the technical specification of another expression or statement, it can parse to any of those expressions listed above.

Print Statement:

Regular expression:

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

The print_statement is very simple: it prints the expression inside the parentheses.

Consider the following example:

```
print("This is an example of the print statement.")
OUTPUT:
    This is an example of the print statement.
```

Equality Expression:

Regular expression:

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

The equality_expression simply evaluates whether two comparrison_expressions are equal or not equal.

Consider the following examples:

```
print(10 == 5)
print(10 != 5)
OUTPUT:
    false
    true
```

Comparison Expression:

Regular expression:

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

A comparison_expression acts exactly as the regular expression suggests, comparing two additive_expressions using boolean operators.

Consider the following example:

```
print(1>3)
OUTPUT:
    False
```

Additive Expression:

Regular expression:

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

An additive_expression acts exactly as the regular expression suggests, adding or subtracting two factor expressions. However, if one of the factor_expressions is a string, the + operator will perform string concatenation instead.

Consider the following example:

```
print(1+2)
print(“Hello “ + “world”)
OUTPUT:
    3
    Hello world
```

Factor Expression:

Regular expression:

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

The factor_expression covers both multiplicative, and divisive operations.

Consider the following examples:

```
print(4*6)
print(6/3)
OUTPUT:
    24
    2
```

Unary Expression:

Regular expression:

```
unary_expression = ( “not” | “-” ) unary_expression |
primary_expression;
```

The unary_expression covers both the negative signs for negative integers, and the boolean negation operator.

Consider the following examples:

```
print(-4)
print(not true)
OUTPUT:
    -4
    false
```

Primary Expression:

Regular expression:

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" |
    "false" | "null" |list_literal | function_call | "(",
    expression, ")"
```

The primary_expression covers everything from basic types like strings, integers, and booleans, alongside more complicated functions such as evaluating list_literals, function_calls, and parenthesized expressions.

Consider the following example:

"This is an example of a primary_expression in the form of a string"

Parenthesized Expression:

Regular expression:

```
"(", expression, ")"
```

A parenthesized expression is used to group sub expressions allowing groups to have higher precedence than others and be evaluated first

Consider the following example:

```
print(4*(3-2))
OUTPUT:
    4
```

Note that the parenthesis force 3-2 to be evaluated first, then that quantity is multiplied by 4. Without parenthesis, 4*3 would be evaluated first, then 2 is subtracted from that quantity.

List Literal Expression:

Regular expression:

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

The list_literal expression represents a collection of elements. List can store multiple values of any type but it is important to note that all elements within a list_literal must have the same type.

Consider the following example:

```
Var names = ["Bob", "Alice", "Dave"]
```

Type Expression:

Regular expression:

```
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list'
    [, '<', type_expression, '>' ]
```

CatScript includes the primitive types: int, string, bool, object, and list. The type_expression is used in several statements and expressions usually to allow the user to dictate a specific type. For example, a user can declare a variable, and the type of that variable explicitly, or the compiler can implicitly derive the type of that variable at runtime.

Variable Statement:

Regular expression:

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

The variable_statement is used to create variables that can be used to reference the value that variable is assigned. In catscript, variables can have an explicitly defined type, or the type can be derived at compile time.

Consider the following examples:

```
var sentence : String = "This variable is explicitly defined as a
    String!"
var unknown = "This variable is implicitly derived as a String!"
```

Both variables are strings, but the type of sentence is determined by the user, whereas the type of unknown is interpreted at compile time based on the expression on the right hand side of the equals sign.

It is important to note that two variables can be assigned the same IDENTIFIER within the same scope. There are errors to give warnings for incorrect IDENTIFIERS.

Assignment Statement:

Regular expression:

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

The assignment_statement is used to assign a new value to a pre-existing IDENTIFIER. When using an assignment_statement the expression on the right hand side of the equals sign must be a compatible type to the IDENTIFIER on the left hand side.

Consider the following example:

```
Var sentence = "This is a cool sentence!"
sentence = "This is the second cool sentence!"
print(sentence)
OUTPUT:
    This is the second cool sentence!
```

This example requires the sentence variable is type String or else an error will be thrown.

If Statement:

Regular expression:

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

The if_statement allows for branching logic by executing different paths based on conditional expressions. It consists of: a condition that determines execution, a body that executes if the condition is true, an optional else if branch for additional conditions, and an optional else branch

for a default fallback. The best way to showcase the functionality of the if_statement is via examples. It is important to note that when using an if_statement, the user can choose to use either an else if, an else, both, or neither. Additionally, the user can have many different else if statements, allowing for several branches.
Consider the following examples:

```
if (true) { print (“if”) }
else if (true) { print(“else if”) }
else { print(“else”) }
OUTPUT:
     If
```

In this example, the initial if_statement evaluates to true, so the body of the if_statement is evaluated and executed, and the program does not evaluate the else if, or else statements.

```
if (false) { print (“if”) }
else if (true) { print(“else if”) }
else { print(“else”) }
OUTPUT:
     else if
```

In this example, the initial if_statement evaluates to false, so the program moves to the else if statement which evaluates to true. So the body of the else if statement is evaluated and executed, and the program does not evaluate the else statement.

```
if (false) { print (“if”) }
else if (false) { print(“else if”) }
else{ print(“else”) }
OUTPUT:
     else
```

In this example, the initial if_statement evaluates to false, so the program moves to the else if statement which also evaluates to false. Then the program moves on to the else statement which always executes in the case that the if_statement evaluates to false, and the optional else if statement evaluates to false.

For Statement:
Regular expression:

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

The for_statement is used to iterate over some expression, and execute statements inside the loop. The components of the for_statement are: IDENTIFIER, expression, and the body which is composed of statement(s). The IDENTIFIER is a variable which will update while the

for_statement iterates. The expression marks the boundary the for_statement must operate on, i.e. the for_statement will only iterate the size of the expression. Finally, the body is composed of some number of statements, and can even be empty.
Consider the following example:

```
for (s in [“Bob”, “Alice”, “Dave”]) {
    print(s)
}
OUTPUT:
    Bob
    Alice
    Dave
```

In this example, the for_statement iterates over the list_literal [“Bob”, “Alice”, “Dave”], and for each item in the list_literal, the body of the for_statement is executed which is just a simple print_statement.

Return Statement:
Regular expression:

```
return_statement = ‘return’ [, expression];
```

The return statement either returns an optionally specified expression, or simply exits the execution of a function. I.e. using return without an expression, just prevents the rest of a function from executing, using return with an expression prevents the rest of a function from executing, and returns that expression.
Consider the following example:

```
return “return statement”
```

Function Declaration Statement:
Regular expression:

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

The function_declaration statement is by far the most complicated statement. Note that there are lots of optional parts to a function_declaration, and lots of required parts which can be populated, or left empty. The best way to model function_declaration is by using several examples.
Consider the following examples:

```
function fun1() { }
```

This function_declaration is entirely valid. The function has an IDENTIFIER, parameter_list, and a body of statements. However, in this example, the parameter_list, and body of statements are all empty meaning there isn’t a lot to this function.

```
function fun2 (a) { print(a) }
```

This function_delclaration is very simple. Similar to the previous example, the function has an IDENTIFIER, parameter_list, and a body of statements, however this time, the parameter_list and body of statements are not empty. That means that when called, this function expects a single variable, in this case it's called ‘a’, and when called, the function will execute its body of statements. In this case, it will print the input variable, ‘a’.

```
function fun3 (a : int) { print(a) }
```

This function_declaration operated almost exactly as the previous example, however, the input variable is explicitly declared to be of type int. This means that when the function is called, the user must supply an integer argument or the program will throw an error.

```
Function fun3 : int (a : int) { return a }
```

Again, this function_declaration is very similar to the previous example, however, now the function is required to return an integer. This means that the body of statements must contain a reachable return_statement, where the expression of that return_statement must be of the specified type, in this case int. If there is not a guaranteed reachable return_statement of type int, the program will throw an error.

```
function complicatedFunction : int (a : int, b : int) {
      var result : int = (a + b)
      return result
}
```

This example is a slightly more complex function_declaration. It takes two arguments, both of type int. Inside the function body, a new variable named ‘result’ of type int is declared and assigned to the sum of the two int arguments. The function then returns the ‘result’ variable, ensuring that the return type requirement of the function_declaration is met.

Function Call Statement:

Regular expression:

```
function_call = IDENTIFIER, ‘(‘, argument_list , ‘)’
```

The function_call statement is used to invoke functions that have been previously declared using the function_declaration statement from above. Note if the function is not previously defined, or is outside the scope of the function_call, the program will throw an error.

Consider the following example:

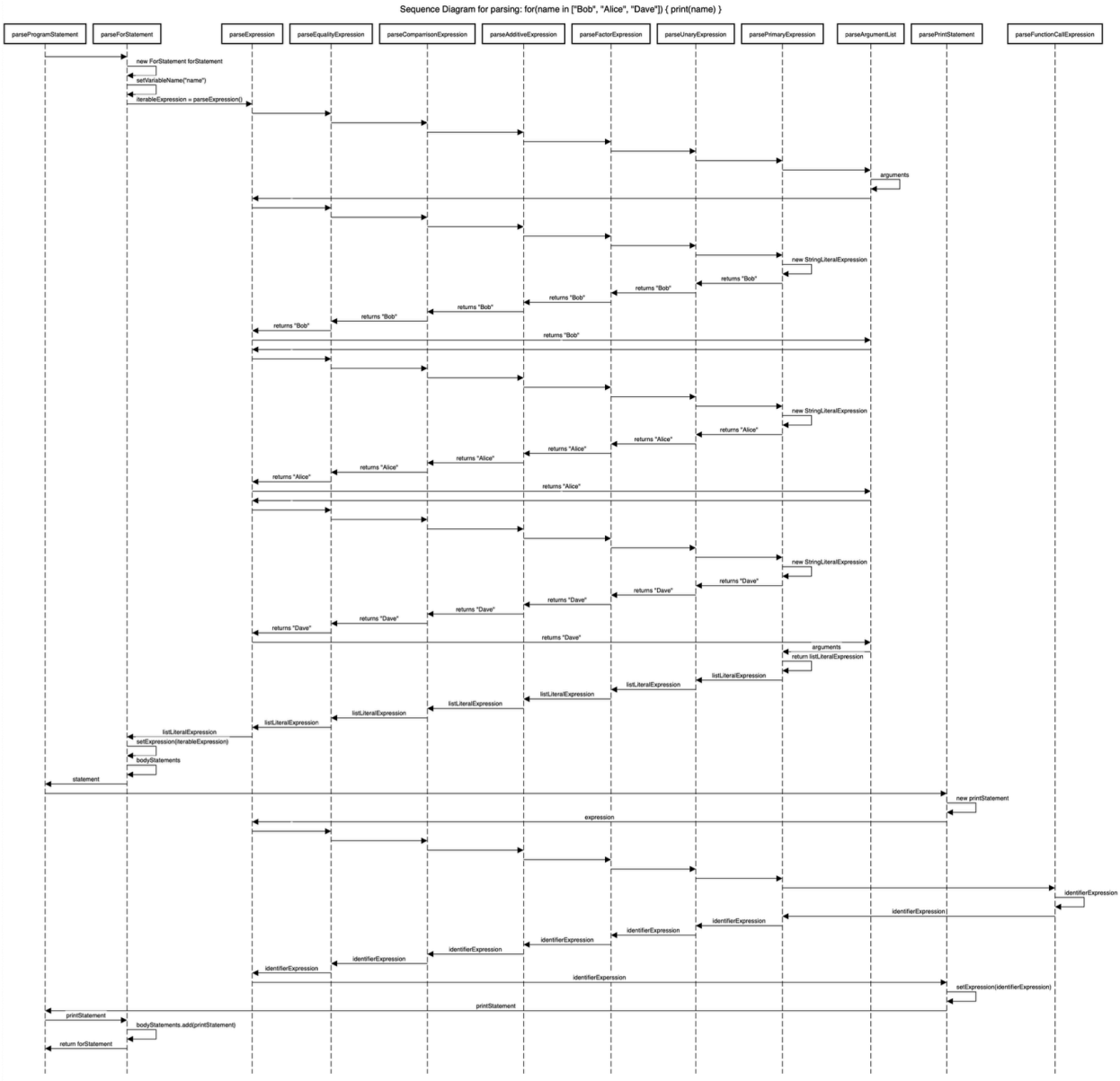
```
print(complicatedFunction(2, 3))
OUTPUT:
      5
```

Note this example uses ‘complicatedFunction’ declared in the final example of the function_declaration statement documentation above. The example also uses the function_call as the expression of a print_statement so the result of the function_call can be easily displayed.

Section 5: UML

This UML sequence diagram was created using the tools found at sequencediagram.org and showcases the functionality of the recursive descent parser. Specifically, the diagram represents the following CatScript:

```
for (name in [“Bob”, “Alice”, “Dave]) { print (name) }
```



The compiler starts the parsing process by calling parseProgramStatement, which then calls parseForStatement. From there, the compiler must parse the iterable expression of the forStatement, in this case a list of strings. This is accomplished by calling parseExpression,

which works its way down the parsing hierarchy to parsePrimaryExpression, where it is determined that the iterableExpression is a List. Then, the compiler calls parseExpression for each element in the list, where again they all work down to parsePrimary Expression. Finally, the List of strings is returned to the forStatement, where the body is then parsed. This is accomplished by calling parseProgramStatement, which then calls parsePrintStatement. The parsePrintStatement method then calls parseExpression, which is eventually determined to be an Identifier, in this case ‘name’. The printStatement is then returned to the forStatement, which in turn is returned to the initial parseProgramStatement call. This process shows the utility of recursive descent parsing, and how similar it is in its implementation to the overall grammar of CatScript.

Section 6: Design Trade-Offs

A major design choice in this project was implementing a recursive descent parser over a parser generator. A recursive descent parser provides a high degree of control and readability, which makes it an ideal choice for a custom language like CatScript. Since the grammar of CatScript is relatively simple in structure, implementing it manually allowed for straightforward mapping between the grammar rules and the corresponding parser methods. This transparency made debugging and customization significantly easier, especially when handling error messages or extending language features. Additionally, recursive descent parsing closely mirrors the structure of the grammar, which provides a better understanding of how parsing works at a fundamental level.

However, this approach also comes with trade-offs. Recursive descent parsers can be more error-prone to implement by hand, particularly for more complex grammars. They lack the automation and robustness provided by parser generators which can automatically handle grammar conflicts, generate parse trees, and manage more sophisticated language features out of the box. Parse generators can significantly speed up development time and reduce the risk of bugs in the parsing logic. In this project, the decision to use recursive descent was ultimately justified due to the simplicity of CatScript and the focus of the assignment, but for larger or more production-level languages, a parser generator might offer a more scalable and maintainable solution.

Section 7: Software Development Life Cycle Model

This project was developed using test driven development. Before the actual project was started, a comprehensive suite of tests was created. These tests covered a range of scenarios, including normal input testing, boundary cases, invalid inputs, and extreme conditions.

Test driven development offers several advantages such as streamlined debugging, well-defined checkpoints and goals, improved code maintainability and modularity, and a more structured development process. However, it also has drawbacks. Poorly written tests can lead to false confidence in code correctness, and excessive focus on refining tests may result in over engineering, diverting time away from the overall functionality.

Ultimately, I believe test driven development was the right choice for this project. It allowed me to experiment, learn, and iterate on my compiler implementation with structured guidance while maintaining independence. The tests provided a clear roadmap, ensured functional correctness, and made debugging more efficient, ultimately helping me develop a deeper understanding of compiler construction.