Capstone Portfolio

CSCI468 Compilers - Spring 2024

Hoang Dang

Tester: Jake Ripley

Program

The source code is linked here:
https://github.com/sunchip39/csci-468-spring2024-private/tree/main/capstone/portfolio

Teamwork

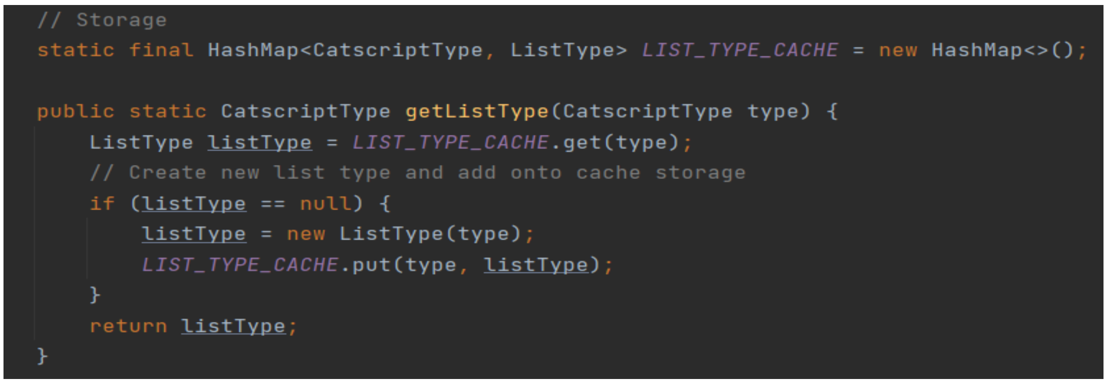
The project was divided into five phases: Tokenization, Parsing, Evaluation, Bytecode Generation, and lastly Partner Testing. Each team member was responsible for developing their own solution for implementing the Catscript language. Additionally, they created and ran tests on each other's solutions. This approach ensured that the codebases created by each team member could withstand the comprehensive testing.

Design Pattern

Memoization was used in the design of the Catscript system. This pattern aimed to minimize the redundancy of operations within the language's architecture. Memoization advocates for the retention of function or process outcomes. For instance, if a function is frequently invoked, memorization suggests storing the input and corresponding result so that no redundant processing is executed.

The application of Memoization is exemplified in the 'getListType()' method within the 'CatscriptType' class, Through this implementation, just a single 'ListType' is generated for 'CatscriptType'; reducing memory overhead and enhancing runtime efficiency.

The implementation is located at:
https://github.com/sunchip39/csci-468-spring2024-private/blob/main/src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java



```
// Storage
static final HashMap<CatscriptType, ListType> LIST_TYPE_CACHE = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    ListType listType = LIST_TYPE_CACHE.get(type);
    // Create new list type and add onto cache storage
    if (listType == null) {
        listType = new ListType(type);
        LIST_TYPE_CACHE.put(type, listType);
    }
    return listType;
}
```

Technical Writing

Catscript Documentation

Catscript is a straightforward scripting language with static typing, compiling directly to JVM bytecode. Below are the core features and syntax of Catscript.

Comments

To annotate Catscript code, use double forward slashes //.

```
// example comment
```

Variables

Declare variable statements using the keyword var.

```
var x = "Hello World!"
```

Types

Explicit typing is used, denoted by a colon :followed by the type. Type is inferred unless declared.

```
var y : int = 10
```

Supported types include:

- string
- int
- null
- object
- list
- bool

Print Statement

Output to the console using the keyword print.

```
print(10) // outputs the number 10.
```

Unary Expressions

Use the minus symbol -to negate variables:

```
var int = 5
print(-int) // outputs -5
```

Use the notkeyword to negate booleans:

```
var boo = true
print(not boo) // outputs false
```

Equality Operands

Equality can be checked by using the ==and !=operators.

```
5 == 1 // evaluates to false
5 != 1 // evaluates to true
```

Comparison Operands

The following comparison operations are supported:

```
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to


Examples -
5 > 0 // evalutes to true
5 < 0 // evaluates to false
5 >= 0 // evaluates to true
5 <= 0 // evaluates to false
```

For Statement

Using the keyword in, a for-loop can be used to iterate over list items.

```
var list =
[1,2,3,4,5] for (i
in list) {
   print(i) // prints all items in the list.
}
```

If-Else Statements

Using the keywords if, else, and if-elseallow for conditional logic.

```
if (x = 10) {
   print("x is equal to 10")
} else if (x < 10) {
   print ("x is less than 10)
} else {
   print ("x is greater than 10)
}
```

Math Operands

The following basic math operations are supported:

```
+ Addition
- Subtraction
* Multiplication
/ Division


Examples -
print(10 + 2) // outputs 12
print(10 - 2) // outputs 8
print(10 * 2) // outputs 20
print(10 / 2) // outputs 5
```

Function Calls

Use the keyword functionfollowed by the identifier name to define a function.

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

Return Statements

Use the keyword returnto exit a function and if desired return a value to the caller.

```
x = 1
y = 2
function add(x, y) {
   return x + y // evaluates to 3
}
```

Design Trade-Offs

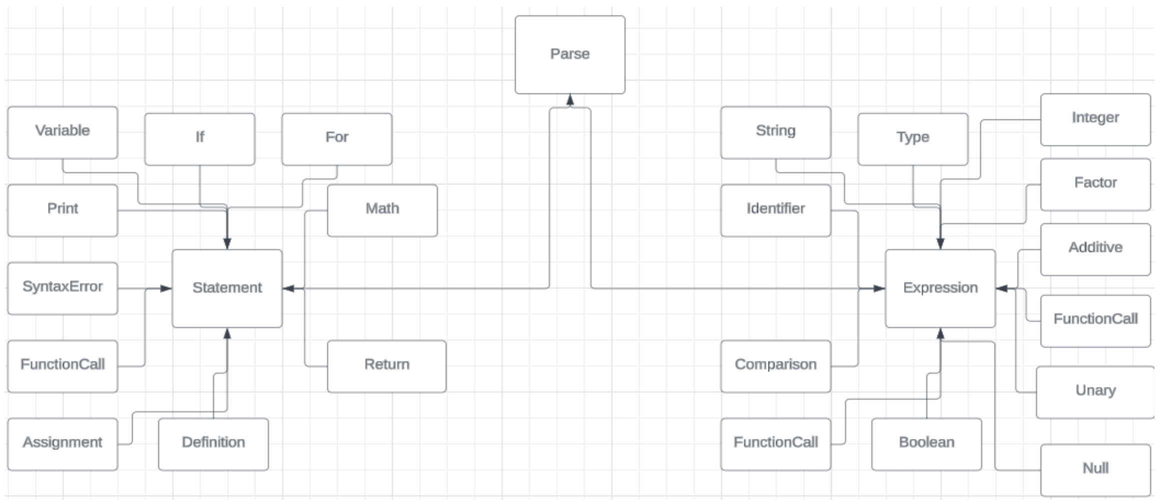
A significant decision in our project was the approach to separation of concerns. Traditionally, SWE principles advocate for clear separation of different functionalities. However, we chose to deviate from this to prioritize simplicity and clarity. Specifically, we tightly coupled evaluation and compilation procedures to parse tree nodes, bypassing the usual separation of concerns. This decision enhanced organization and aided in faster development. Additionally, instead of using a parser generator, we chose to implement our own parser using recursive descent, allowing us to create a parser that mirrors the language's grammar.

Software Development Life Cycle Model

The Catscript project was developed using the test-driven development methodology, organizing our software development life cycle around incremental testing milestones. With each phase – from Tokenization to Parsing, Evaluation, and Bytecode – we utilized a series of tests to validate milestone completion. This approach ensured integrity of the language at every step, as successful test executions indicated progress and readiness for the next phase.

UML

No UML was needed or used for this project because the overall design was pre-determined by the professor so that we could focus on parsing. Here is a basic class diagram showing the functionality of parsing expressions and statements.


Parse
Variable
If
For
Print
Math
SyntaxError
Statement
FunctionCall
Return
Assignment
Definition
String
Type
Integer
Identifier
Factor
Additive
Expression
FunctionCall
Comparison
Unary
FunctionCall
Boolean
Null