CSCI 468 - Compilers
Capstone Portfolio
Lucas Bega

Section 1: Source Code

A zip file of the source code to this project can be found at https://github.com/thelitttleman/csci-468-spring2025-private/blob/main/capstone/portfolio

Section 2: Teamwork

This computer science capstone was all completed by myself excluding the following elements of this section. My work time estimate is 30-35 hours. My partner, teammate 2, spent roughly three hours creating the CatScript Documentation and extra tests.

CatScript Documentation - Written by Teammate 2

CatScript is a lightweight statically typed scripting language. Its programs compile to Java bytecode and support common features you would expect to be found in modern languages: variables, control flow, functions, etc.

CatScript Lexical Elements:

- *Identifiers*: [A-Za-z][A-Za-z0-9]*
- *Integers*: any sequences of digits, such as 42, 0, -7 (negative is unary minus)
- *Strings*: double-quoted string, such as "cat"; escape \ supported.
- *Keywords*: else, false, function, for, if, in, not, null, print, return, true, var.
- *Syntax*: +, -, *, /, <, <=, >, >=, =, ==, !=
- *Punctuation*: (,), {, }, [,], **,**, **.**, **:**,

Types:

- *int:* 32-bit integer
- *bool: b*oolean value (true/false)
- *string:* java-style string
- *list<x>:* list of values with the type ‘x’
- *object:* any type of value
- *null:* the null type

Type syntax:

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

Expressions:

Expressions compute values and are left-associative except for unary expressions. This document displays the types of expressions in a top-down order of precedence.

Primary expressions can have:

- *Identifier*: variable or function name
- *Literals*: STRING, INTEGER, boolean (true/false), null
- *List literal*: '[', expression, { '**,**', expression } ']'; creates a list: [1,2,3] or is empty [].
- *Function call*: IDENTIFIER, '(', argument_list , ')'; e.g. name(arg1, arg2, ...) returns a value

Primary expression syntax:

primary = IDENTIFIER | INTEGER | STRING | 'true' | 'false' | 'null' | list_literal | function_call | '(' expression ')';

Unary expressions can have:

- *not*: negates a boolean value
- -: negates an integer value

Unary expression syntax:

unary = 'not' unary | '-' unary | primary;

Binary Operators are, from highest to lowest precedence:

1. *, / (factor expression)
2. +, - (additive expression)
3. <, <=, >, >= (comparative expression)
4. ==, != (equality expression)

Additionally:

- '+' in (2) will concatenate strings if either left-hand side or right-hand side expression is a string
- Comparisons in (3) return a boolean value
- Equality operators in (4) can be used with any type.

Binary operator expressions syntax:

factor = unary { ('*' | '/') unary };

additive = factor { ('+' | '-') factor };

comparison = additive { ('<' | '<=' | '>' | '>=') additive };

equality = comparison { ('==' | '!=') comparison };

Statements:

- *for:* iterates over some list
- *if: evaluates the expression (boolean) and branches depending on value*
- *print: evaluates the expression, converts it to a string, writes the string*
- *variable:* declares some identifier and initializes it to its expression value
- *assignment:* updates existing variable to specified expression
- *return:* exits current function, either returning the expression value or 'nothing' (void)
- *function call:* calls a function

Statement syntax example:

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

if_statement = 'if', '(', expression, ')', '{', { statement }, '}'

['else', (if_statement | '{', { statement }, '}')];

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

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

assignment_statement = IDENTIFIER, '=', expression;

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

function_call_statement = function_call;

Functions:

- *Parameters*: zero or more, can be typed.
- *Return type*: specify after **:**, default is void.

- *Body*: is a sequence of statements. Must cover all branches with a return if not of void type

Function syntax example:

> **function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' + [':' + type_expression], '{', { statement }, '}';**
>
> **parameter_list = [parameter, {',' parameter }];**

Additional Testing - Written by Teammate 2

Teammate 2 created the following three tests to ensure additional functionality. After implementing them in the codebase, I learned that the program passed the tests without any needed modifications.

```
@Test
public void parseDoubleParenthesizedExpressionWorks() {
   ParenthesizedExpression outer = parseExpression("((1))", false);
   assertTrue(outer instanceof ParenthesizedExpression);

   // first () is another parenthesized
   Expression middle = outer.getExpression();
   assertTrue(middle instanceof ParenthesizedExpression);

   // nested inside is the integer literal 1
   Expression innerMost = ((ParenthesizedExpression) middle).getExpression();
   assertTrue(innerMost instanceof IntegerLiteralExpression);
   assertEquals(1, ((IntegerLiteralExpression) innerMost).getValue());
}

@Test
public void parseNestedListLiteralExpressionWorks() {
   ListLiteralExpression expr = parseExpression("[[1],[2]]");
   assertEquals(2, expr.getValues().size());
   assertTrue(expr.getValues().get(0) instanceof ListLiteralExpression);
   assertTrue(expr.getValues().get(1) instanceof ListLiteralExpression);
}

@Test
public void parseMultiStringConcatenationExpressionWorks() {
   AdditiveExpression expr = parseExpression("\"foo\" + \"bar\" + \"again\"",
false);
   assertTrue(expr.isAdd());
   assertTrue(expr.getLeftHandSide() instanceof AdditiveExpression);
   assertTrue(expr.getRightHandSide() instanceof StringLiteralExpression);
}
```

Section 3: Design Pattern

Although there were most likely multiple design patterns used in the overall structure of classes in the CatScript codebase, one design pattern that was used entirely inside of a class was the memoization pattern. This pattern is designed to speed up the execution of potentially high-cost functions by caching already-computed results, and storing them in such a way that retrieving those results is faster than executing the function again. In the implementation, the memoization pattern was used to cache ListType objects in an effort to conserve both space and running time. Without the pattern, a new ListType object would be created every time a new list was created in code. The implementation uses a hashmap as the storing data structure, which has a constant lookup time. In addition, if two lists of the same component type are created, then only one ListType object will be used for both. Below is the code that implements the memoization pattern. This code segment can be found in

src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java

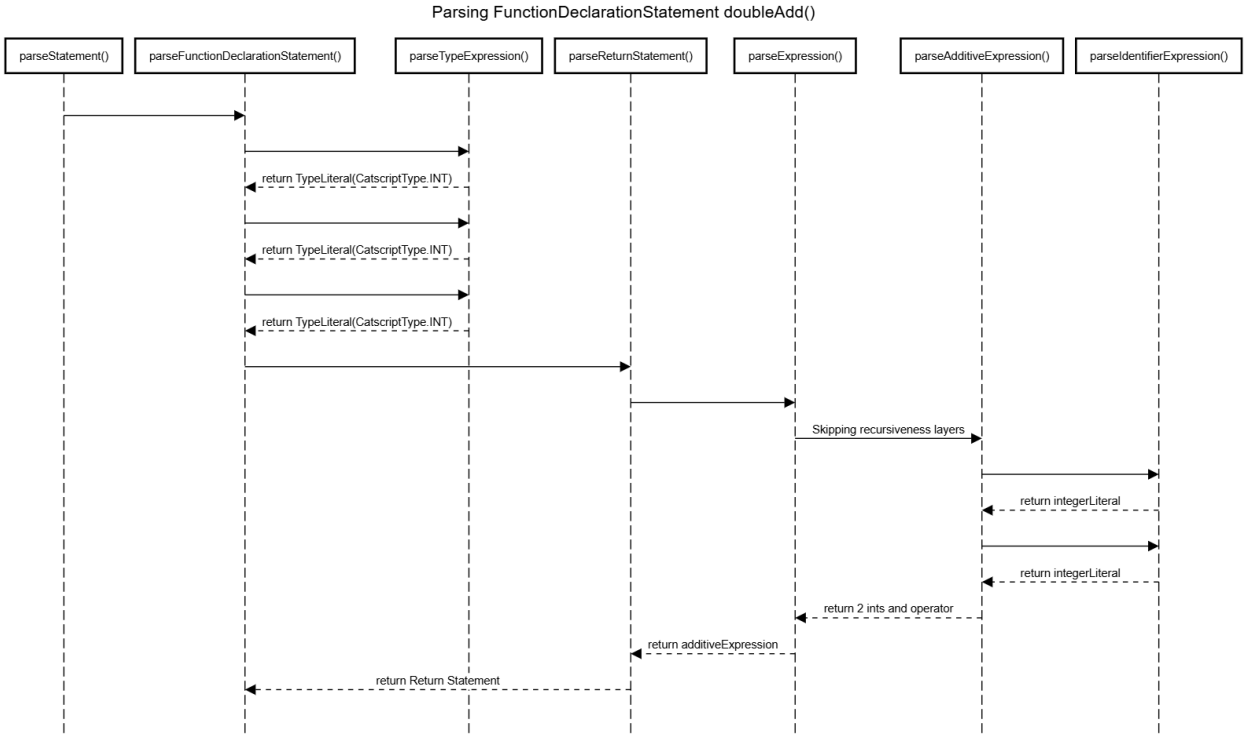
```
static Map<CatscriptType, CatscriptType> MEMOIZATION_CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType existingType = MEMOIZATION_CACHE.get(type);
    if(existingType != null) {
        return existingType;
    }
    else {
        ListType listType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

Section 5: UML Diagram

If we consider the following code segment in CatScript.

```
1
2  function doubleAdd(x : int, y : int) : int {
3      return x + y
4  }
```

Then the corresponding sequence diagram looks as follows.



Catscript starts all program executions by either parsing a single expression, or a collection of statements. In this case it is a statement. Due to the explicitly typed variables in the function declaration statement, the parser requires three type literals to be returned to it. After, the body of the function parses. The only statement in the function is a return statement, but it has an additive expression connected to it. This is where the recursive descent process is clearest. All levels of the parser are not shown for brevity but the parser checks for equality and comparison expressions before figuring out that it is an additive expression. The additive expression resolves the two identifiers on either side, as well as the “+” operator. The expression gets sent back to the return statement which in turn gets sent back to the function declaration statement, ending the parsing.

Section 6: Design Trade-Offs

The largest design trade-off made throughout this project was the decision to write a recursive descent parser by hand instead of using a parser generator. This decision was made for multiple reasons. Typically, parser generators create a parser and tokenizer that is constrained by the wanted language specification. They are known to create much simpler and more optimized code than what humans tend to make on their own, but this comes with the cost of readability. Output from parser generators can be extremely unintuitive and without proper documentation for the language, general users would be left to figure out the semantics on their own. Input to these generators also

uses heavy syntax and can require a lot of time to learn, but the end user is not affected by it. Ultimately, recursive descent parsers require much more work to implement, but It was worth the effort. The codebase structure much more closely resembles the grammar that CatScript is based upon, making it easy to use. The structure of the final product is clean and follows typical object-oriented design principles. The actual recursive descent parser is all contained within one class, but each main function of the compiler is isolated from the others and there is not an overuse or underuse of any part of the codebase.

Section 7: Software Development Life-Cycle Model

While most of the software development was pre-planned as part of the compilers class, test driven development was used to ensure functionality of the codebase. Test driven development was probably the best choice, as the grammar that the CatScript language is based on has an already-defined structure. This meant that the tests could be created easily and be easily understandable to the implementers. The highest degree of abstraction needed for the tests were for those that tested faulty code. The codebase uses an error system that allows for each parse element to have errors attached to it if needed, as well as an ErrorType class that acts as a dictionary for the possible errors a user could make. This allowed for much easier debugging of code. Debugging was almost as important as the tests that were used. The IDE used while implementing the system allowed for step-by-step execution through all tests to find problems in code much quicker. The largest hindrance that test driven development brought was the overall lack of tests. Some checkpoints of the project had more in-depth testing than others, so sometimes a problem in a previous checkpoint would go undetected, and changes to code were necessary even after that portion was considered “done”. This then affected the schedule of the following checkpoints.