Montana State University
Gianforte School of Computing

CSCI 468 Compilers

Jada Bryant
Walker Ward
May 2021

Table of Contents

Section 1: Program

The source code can be found at this location: csci-468-spring2021-private/capstone/src.zip

Section 2: Teamwork

This project was divided such that each team member would have dedicated tasks and sections of the project to focus on. Team Member 1 was responsible for management of the GitHub repository and was the primary contributor to the code base. Their programming focused heavily on completing the tokenizer, parser, evaluation, and bytecode checkpoints to the GitHub repository prior to each one's deadline. This team member also resolved any merge conflicts when they arose, and for ensuring that the program passed all tests. Team Member 2 was responsible for writing the technical documentation, such as this report, and for writing program tests to augment our test driven development paradigm. All communication between team Member 1 and team Member 2 was facilitated via discord and email. Due to the coronavirus pandemic in-person contact and meetings were deemed an unnecessary safety hazard.

Estimated Contributions

Total estimated hours: 120

Team Member 1:

Contributions: Code management and primary programmer.
Estimated Work Hours: 100 hours, ~83% contribution in time.

Team Member 2:

Contributions: Documentation and tests.
Estimated Work Hours: 20 hours, ~17% contribution in time.

Section 3: Design Pattern

Memoization

In programming memoization is a technique applied to expensive recurring function calls in order to optimize it. At a high level, this approach stores the result of this function call so, that the next time this function is called, it will not have to repeat the same computation. It is essentially caching a set of inputs and outputs.

To accomplish this functionality the function must store its parameter values and return value in a cache any time it is called with new arguments. The cache is typically implemented as a HashMap or another type of data structure. Anytime this function is called it compares its parameters to those in the cache and, if they are logically equivalent the function returns the originally computed value from the cache that is associated with those arguments.

The benefits of this technique are that for extremely time-consuming computations it saves time that would otherwise be spent on the computation. It could also be argued that this approach could save system resources such as ram and CPU time. However, this system is not without drawbacks. For one, this technique must take time to check whether this call has already been stored in the cache or not. Secondly, this pattern is generally only viable for functions that are called often, can only take a narrow range of input and do not rely on random numbers. Furthermore, this approach is not thread safe. Finally, this technique would not be viable in a real production system, but it is sufficient for this project.

Implementation: src.zip\...\parser\CatscriptType.java

Section 4: Technical Report

Introduction

The program is a compiler for Catscript written entirely in Java. Catscript is a statically typed language that supports standard if statements, for loops, and function definitions as well as List Literals and local variable type interface, two features that Java does not have.

File Structure

The code for this project is in `/src/main/java/edu/montana/csci/csci468`. In this folder is `tokenizer`, `parser`, and `bytecode` which are the main folders for this project. `tokenizer` contains the code for the tokenizer in `CatScriptTokenizer.java` `parser` contains the parser `CatScriptParser` as well as the folders `statements` and `expressions` which have various files for all the statements and expressions that CatScript supports.
The tests for this project are in `/src/test/java/edu/montana/csci/csci468`. In this folder are the folders `tokenizer`, `parser`, `eval`, and `bytecode` which contain all multiple files that run tests to check the program is running as expected.

Typesystem

Catscript is a statically typed language with a fairly simple type system. - `int` - integers
- `string` - Strings
- `bool` - a Boolean value
- `object` - any object
- `null` - the type of null value
- `void` - the type no type

Catscript has one complex type `list`. A list can be declared with given type with `list<T>`.
- `list<int>` - list of integers
- `list<object>` - list of objects
- `list<list<int>>` - list of lists of integers Catscript elements have a `verify` method that verifies the code. It registers all function types in the symbol table, calls `validate`, collects parse errors from children, and throws if any errors occurred.

Assignability

For simple types, assignability is fairly simple: Nothing is assignable from `void`, everything is assignable from `null`. For all other types, check the assignability of the backing java classes. Lists have different assignability. Catscript lists are covariant and immutable: Null is assignable to list. Otherwise check if it is list type, if so then check if the component types are assignable.

Tokenizing

The tokenizer simplifies the grammar and optimizes the parser by turning strings into tokens. The goal of the tokenizer is to understand the syntax of the language rather than the meaning behind each token. The tokenizer is responsible for identifying tokens and maintaining the line the token is on.
Below is the method in the tokenizer that consumes the whitespace between each token. You can see how the tokenizer keeps track of where it is in the line by adding to the offset each time whitespace is used. When a new line token is present in the input document, the tokenizer resets the `lineOffset` variable and adds to `line` to keep track of which line it is on.

```
private void consumeWhitespace() {
    while (!tokenizationEnd()) {
        char c = peek();
        if (c == ' ' || c == '\r' || c == '\t') {
            lineOffset++;
            postion++;
            continue;
        } else if (c == '\n') {
            lineOffset = 0;
            line++;
            postion++;
            continue;
        }
        break;
    }
}
```

The `scanSyntax` method is the method that is responsible for parsing the syntax for Catscript. It checks each syntax token, if that token matches it adds the token to the list of tokens that will later be sent to the parser. Below is a snippet of this method.

```
private void scanSyntax() {
    int start = postion;
    if(matchAndConsume('+')) {
        tokenList.addToken(PLUS, "+", start, postion, line, lineOffset);
    } else if(matchAndConsume('-')) {
        tokenList.addToken(MINUS, "-", start, postion, line, lineOffset);
...
    } else if (matchAndConsume(']')) {
        tokenList.addToken(RIGHT_BRACKET, "]", start, postion, line, lineOffset);
    } else if (matchAndConsume(':')) {
        tokenList.addToken(COLON, ":", start, postion, line, lineOffset);
...
```

Parsing

The parser takes the list of tokens from the tokenizer and verifies that the string is part of Catscript's grammar. If any syntax errors are found, it throws an error with the location of the syntax error provided by the tokenizer. Because recursive descent was used to build the parser, the structure of the parser looks similar to the grammar that Catscript recognizes.

Evaluating

Literals

A literal is a literal value encoded into the programming language, evaluating these values is fairly simple. The values are returned when evaluated. Below is the evaluate method for a integer literal. Strings, booleans, null, and list literals are all evaluated this way.

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
    return integerVal;
}
```

Parentheses

Parenthesis are evaluated by returning the value the enclosed expression evaluates to.

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
    return expression.evaluate(runtime);
}
```

Unary Expressions

Unary expressions are expressions with a single argument expression and are evaluated by evaluating the right hand side and applying the operator to that value. Below is the evaluate method for unary expressions.

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
    Object rhsValue = getRightHandSide().evaluate(runtime);
    if (this.isMinus()) {
        return -1 * (Integer) rhsValue;
    } else if (this.isNot()) {
        if (rhsValue.equals(true)) {
            return false;
        } else {
            return true;
        }
    } else {
        return false;
    }
}
```

Binary Expressions

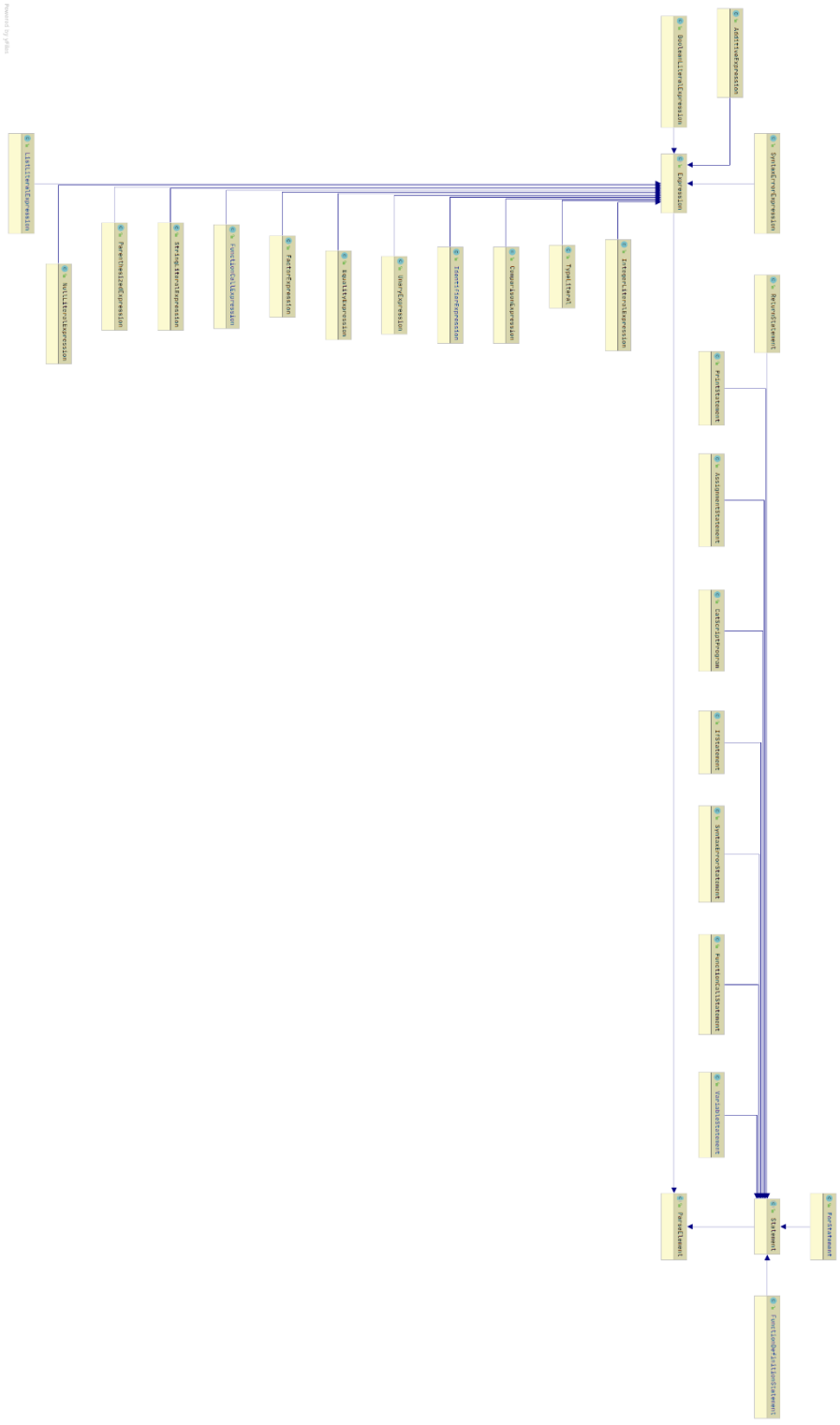
Binary expressions are expressions that have to arguments, a left hand and a right hand side. Binary expressions are evaluated by checking the type then concatenating or adding the two values together and returning the resulting value. Below is the evaluate method in `AdditiveExpression.java` for evaluating a binary expression.

```java
@Override
public Object evaluate(CatscriptRuntime runtime) {
    if (leftHandSide.getType().equals(CatscriptType.STRING) || rightHandSide.getType().equals(CatscriptType.STRING)) {
        Object leftHandValue = leftHandSide.evaluate(runtime);
        Object rightHandValue = rightHandSide.evaluate(runtime);
        if (leftHandValue == null) {
            leftHandValue = "null";
        }
        if (rightHandValue == null) {
            rightHandValue = "null";
        }
        return leftHandValue.toString() + rightHandValue.toString();
    } else {
        Integer leftHandValue = (Integer) leftHandSide.evaluate(runtime);
        Integer rightHandValue = (Integer) rightHandSide.evaluate(runtime);
        if (isAdd()) {
            return leftHandValue + rightHandValue;
        } else {
            return leftHandValue - rightHandValue;
        }
    }
}
```

Compiling

The compile methods for each statement and expressions are written bytecode. We write what we need to do in Catscript into our Java IDE and use the bytecode that is generated to write the compile method for the expressions and statements. To compile an expression, we start by compiling the right hand side and adding it onto the stack. Depending on the expression, the left hand side will also be compiled and added to the stack. Both the right hand and left hand side will compile to a set of instructions and leave the value on the stack. The instruction is then run with that value. Opcodes are also used to compile expressions and statements. Different opcodes have different meanings as well.

Section 5: UML



UML Diagram of Parse Element Hierarchy

Section 6: Design Trade-Offs

Recursive Decent vs. Parser Generators

Recursive Decent

Recursive Decent is a top-down parsing algorithm, widely used in industry to parse tokenized source code. This method of parsing allows the programmer to develop a much better understanding and intuition about grammars and parsing because of its simplicity. The primary idea of this technique that for each production in the grammar there exists a method for it that is named after it. In that method, call the methods defined on the right-hand side of the production, matching as needed. This idea of providing every production its own method allows the recursive nature of parsers to be explained more clearly and understood better by those learning them for the first time.

Parser Generators

Parser Generators are made of two chunks: A lexical grammar as a REGEX (Regular Expression) and a long grammar as a EBNF (Extended Braccus Naur Form). The lexer is generated by mixing code generation with the actual grammar. The parser looks very similar to the lexer and produces an AST (Abstract Syntax Tree). The parser actually generates a recursive decent parser it's just much more abstract and harder to read. Theoretically, Parser Generators require less code and infrastructure to get going. However, more time is typically spent learning about their idiosyncrasies that the compiler and parser itself. Furthermore, Parser Generators are not used widely in industry, are much more complicated for our purposes and much harder to read than a Recursive Decent parser.

So, after careful consideration and paying close attention to out complexity budget and schedule it makes much more sense to implement a recursive decent parser. To reiterate, they are simpler, easier to understand, more widely used and gives us a better understanding of parsers and compilers.

Section 7: Software Development Life Cycle Model

Test-Driven Development (TDD)

Test-Driven Development is a style of programming that focusses on coding, testing and design at the same time. With this in mind, developers design the bare minimum to make tests pass, no more, no less. This has been attributed with much cleaner, clearer and simpler code that is often less prone to bugs. This allows the programmers to focus on only what is important, ignoring, for the time being, the more advanced, complicated, or unrelated features.

Personally, TDD has been an eye-opener. Being able to run tests against code you're writing is an extremely helpful feature. Before being introduced to this method of programming I would write my own makeshift tests and run tests by hand, over and over and over again to test features, work out bugs and verify outputs. Now, all I must do is write a single test, or twenty, without even having to even start the code. Then, I can work on each test at a time, slowly but surely, getting each one to pass until the entire project is complete. Not only did this style of programming help me ensure my code was functioning properly but it also gave me a sort of roadmap, like chapters in a book. Each of them allowing me to get closer and closer to then end while leading me in the right direction.