Capstone Documentation

Lucy Lewis, Yves Vernooy

~ CSCI 468: Compilers ~

Section 1: Program

Please see the zipped source code & specification in source.zip

Section 2: Teamwork

To manage the capstone project effectively, our team divided the work into two primary roles: implementation of the CatScript compiler and validation through systematic testing and technical documentation.

Team Member 1, serving as the implementor, was responsible for approximately 90% of the overall project work. This included implementing the full CatScript compiler pipeline, beginning with the tokenizer. The tokenizer performed lexical analysis by scanning raw source code and converting it into a structured stream of tokens, with each token representing a fundamental syntactic element of the language such as a keyword, identifier, literal, or operator. The recursive descent parser then processed this token stream to construct a structured abstract syntax tree (AST), organizing tokens in accordance with CatScript's grammatical rules. Next, semantic analysis was performed to analyze program structure beyond syntax and ensure it adhered to CatScript's semantic rules before further processing. In particular, this stage analyzed the abstract syntax tree to enforce type correctness by ensuring that all operations were applied to compatible types. It verified that expressions such as arithmetic and comparison operations used valid operand types, and that values assigned to variables matched their declared or expected types. Following semantic analysis, the system supported two forms of execution: evaluation and bytecode generation. Evaluation directly processed the abstract syntax tree to execute program behavior by interpreting its structure and producing runtime results. This process involved traversing the tree and resolving expressions into concrete values, executing statements in order, and handling control flow constructs such as conditionals and loops to determine execution paths. In contrast, bytecode generation converted the validated program into a lower-level executable representation, where the abstract syntax structure was translated into a sequence of bytecode instructions designed for runtime execution in a virtual machine. Together, these components formed a complete compiler pipeline capable of processing CatScript programs from source code to execution.

Team Member 2 was responsible for approximately 10% of the overall project work, serving as the tester and documentor for the CatScript compiler. Their primary role focused on validating the correctness and completeness of the implementation developed by Team Member 1. This included developing additional test cases in csci468.partner.PartnerTests, including listMutationWorksProperly(), deepNestingAndLoopMutationWorksProperly(), and multiFunctionRecursionAndCompositionWorksProperly(), to further validate compiler behavior beyond basic functionality. These tests were designed to evaluate more complex program scenarios, including nested structures, iterative mutations, and interactions between multiple function calls, ensuring the compiler handled advanced and non-trivial cases correctly. Team Member 2 was also responsible for producing the technical documentation for the CatScript language. This included clearly explaining the language's core features, syntax, and overall structure, as well as describing how programs are formed using expressions, statements, and control flow constructs. As a result, the documentation serves as a clear reference for understanding and using the CatScript language.

Ultimately, collaboration on the project was conducted asynchronously to accommodate differing schedules and allow both team members to work independently. Team Member 1 primarily focused on implementing the CatScript compiler, while Team Member 2 worked in parallel on testing and documentation based on the expected behavior of the language. Progress was communicated through periodic updates, with Team Member 1 sharing completed functionality and Team Member 2 providing test results and documentation updates as needed. Both team members actively reviewed each other's contributions during development. Team Member 1 evaluated the test cases and documentation for accuracy and completeness, while Team Member 2 reviewed the implementation for correctness, readability, and overall code quality.Revisions were made iteratively until all components met the agreed standard of quality. This ongoing review cycle ensured that all components were continuously refined until they satisfied the agreed standards of quality, clarity, and completeness.

Section 3: Design Pattern

```
public CatscriptType getListType() {
    if (listType == null) {
        listType = new ListType(this);
        return listType;
    }
    return listType;
}
```

The memoization design pattern is implemented in csci468.parser.CatscriptType.getListType() to ensure that list types are created only once and reused throughout the compiler. At its core, each CatscriptType maintains a listType field that acts as a cache for the corresponding ListType instance. On the first call to getListType(), the method checks whether listType is null. If it is, a single ListType object is created using new ListType(this) and stored in the field. From that point onward, all subsequent calls return the already stored instance rather than constructing a new one. This ensures consistent reuse of type objects across the compiler, preventing redundant allocations and maintaining a one-to-one mapping between a base type and its list type representation.

Ultimately, without memoization, every call to getListType() would execute new ListType(this), creating a completely new ListType instance each time. This means that repeated calls would produce multiple duplicate objects that all represent the same logical type. Instead of reusing a single shared instance, the compiler would accumulate many identical but distinct objects in memory, leading to redundancy and inefficiency. Over time, especially during semantic analysis and type checking where types are requested frequently, this can result in a large number of unnecessary allocations. These duplicate objects increase memory usage and add overhead for object creation and garbage collection.

More importantly, this duplication introduces correctness issues. Because each instance is a different object, simple reference comparisons (e.g., ==) would fail even when two types are logically identical. This complicates type checking, since the compiler can no longer rely on object identity to determine whether two types are the same. As a result, the compiler would need to perform deeper structural comparisons each time types are checked, such as comparing component types recursively, which increases both complexity and runtime overhead. It also makes the implementation more error-prone, since forgetting to use structural comparison in even one place could lead to incorrect type mismatches or inconsistent behavior during semantic analysis.

Section 4: Technical Writing/Catscript Guide

CatScript Overview

The CatScript language supports the following core features:

- Variable declarations and assignments
- Expressions
  - Comparison and equality
  - Arithmetic (addition, subtraction, multiplication, division, and modulus)
  - Unary (negation and logical NOT)
- List literals, indexing, and indexed assignment
- For loops (iteration over elements in a collection)
- While Loops
- Conditional statements (if/else)
- Print Statements
- Function declarations and function calls
- Return statements

CatScript Grammar

```
catscript_program = expression | { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            while_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            return_statement |
            function_call_statement;

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

while_statement = 'while', '(', expression ')',
                '{', { statement }, '}';

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

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

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

function_call_statement = function_call;

assignment_statement = IDENTIFIER [ '[' expression ']' ] '=', expression;

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

parameter_list = [ parameter, {',' parameter } ];
```

```
parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

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

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expressio

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

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

unary_expression = ( "not" | "-" ) unary_expression | index_expression;

index_expression = primary_expression { '[' expression ']' }

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

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

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

argument_list = [ expression , { ',' , expression } ]

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

The grammar for CatScript is shown above in EBNF form. This grammar should be used as a reference for understanding the correct syntax and structure of CatScript programs.

CatScript Type System

CatScript is statically typed, with the following types:

- `int` - a 32-bit integer value
- `string` - a Java-style string
- `bool` - a boolean value (true/false)
- `list<T>` - a list containing elements of type T
- `null` - the null type
- `object` - a generic type that can store any value

In CatScript, types can be explicitly declared for variables, function parameters, and function return values. Type annotations are optional for variables and parameters, meaning they may be included to enable type checking but are not required. Function return types may also be specified to enforce the expected type of a function's output, but this is optional.

Type checking occurs at compile time during semantic analysis, where type annotations are validated for correctness.

CatScript Scoping Rules

CatScript uses block scoping, where variables are only accessible within the {} block in which they are declared. Each new block introduces a new scope, including control flow blocks (if, for, while) and function bodies. Variables declared in an inner scope are not accessible outside of that scope.

Example Program

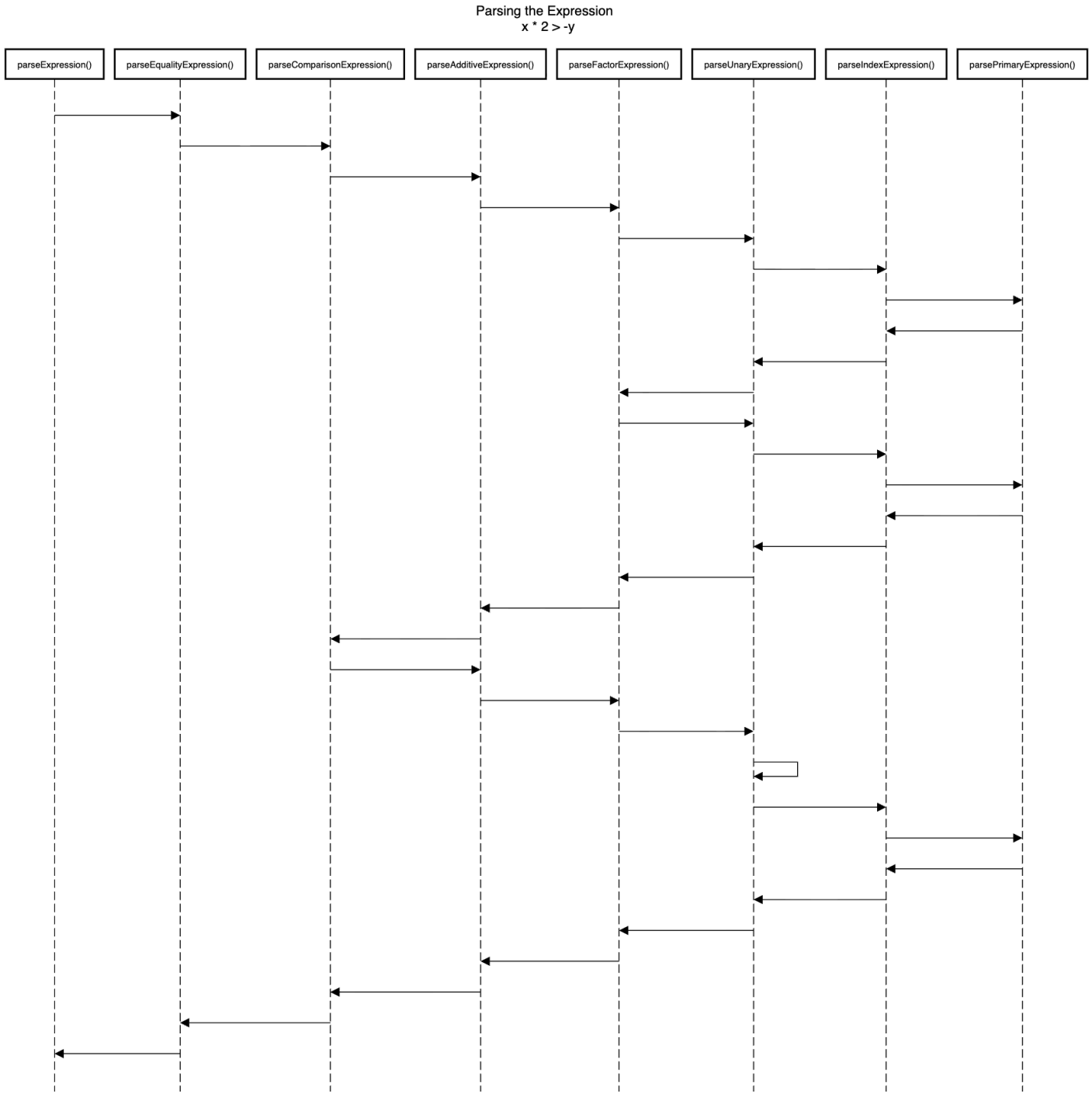
The following is a complete CatScript program demonstrating basic language functionality, including variables, functions, control flow, and expressions.

```
function add(a: int, b: int): int {
    return a + b
}
var x: int = 10
var y: int = 5
var sum: int = add(x, y)
print(sum)

var numbers: list<int> = [1, 2, 3, 4, 5]
var total: int = 0
for (i in numbers) {
    if (i % 2 == 0) {
        total = total + i
    } else {
        total = total + add(i, 1)
    }
}
print(total)

while (x > 0) {
    print(x)
    x = x - 1
}
```

Section 5: UML



The sequence diagram above illustrates how the parser handles the expression x * 2 > -y, showing the step-by-step flow of function calls as the expression is parsed. It begins at the highest expression level and works downward through the grammar hierarchy, reaching the comparison level where each side of the operator is evaluated in turn. The parser first fully resolves the left-hand side expression x * 2 by applying the multiplication rule, ensuring higher-precedence operations are handled before comparison. It then recognizes the > operator and proceeds to handle the right-hand side, -y, using a recursive call to the unary rule to apply negation. The diagram highlights how the parser enforces operator precedence through its layered structure, guaranteeing that multiplication and unary operations are resolved before the comparison. As each rule

returns its result, the components are progressively combined until the full comparison expression is built and represented as a complete abstract syntax tree.

Section 6: Design Trade-offs

A key design decision in the CatScript compiler was the use of a handwritten recursive descent parser instead of a parser generator. In this approach, the grammar of the language is represented directly in code, with each parsing function corresponding to a specific rule in the grammar. As a result, the parsing logic directly reflects the structure of the language's grammar.

This decision was partly motivated by readability concerns. Compared to parser generators, which typically produce large amounts of automatically generated code from an external grammar specification, the handwritten approach is significantly easier to read and navigate. Generated parsers often rely on dense, machine-produced logic that is not intended to be manually read or modified, which makes it difficult to trace how a specific language construct is processed or where a particular rule is applied. This can create a disconnect between the grammar definition and the actual runtime behavior of the parser, since understanding the system requires interpreting both the original grammar specification and the generated code. In contrast, a recursive descent parser keeps all parsing logic explicit within a small set of clearly named functions, making the structure of the language visible directly in the code. Each function directly represents a grammar rule, and calls between functions reflect how those rules are composed and applied during parsing. This means the execution flow closely follows the hierarchical structure of the language itself, rather than being hidden behind generated or abstracted logic. As a result, understanding how the parser handles a construct only requires reading the relevant functions in the codebase, without needing to interpret an external grammar specification or navigate through machine-generated parsing routines.

In addition, a handwritten recursive descent parser provides a more straightforward debugging experience because the entire parsing process is explicitly visible and easy to follow within the code. This allows a developer to step through execution line by line in a debugger and directly observe how each token is consumed and how the parse tree is constructed step by step. When an error occurs, such as incorrect parsing of nested expressions, missing tokens, or broken statement structure, the issue can be traced directly to the specific function responsible for that grammar rule. This tight correspondence between grammar rules and implementation makes it easier to isolate the exact point of failure and understand why the parser is misinterpreting the input. In contrast, parser generators often obscure the cause of errors within their generated output, where parsing behavior is distributed across many internal states and transitions rather than organized into clear, named functions. This makes debugging difficult because the runtime execution does not map cleanly to the original grammar rules, so a single language construct may be handled across multiple generated states with no obvious entry point. As a result, when an error occurs, it is harder to determine where in the parsing process the input first deviated from the expected structure, and developers are often forced to rely on indirect error messages or trace through complex generated logic rather than following a clear, readable control flow.

Furthermore, a handwritten recursive descent parser also allows for more precise error handling compared to parser generators, which often produce more generic syntax errors by default. In CatScript, errors can be detected at specific points in the parsing process, allowing the compiler to report exactly what was expected and where the issue occurred. For example, if a required token is missing or an expression is malformed, the parser can generate a targeted and meaningful message based on the current context. This leads to clearer and more helpful feedback compared to generic syntax errors, improving both debugging during development and usability for anyone writing CatScript programs.

However, this approach required significantly more effort than using a parser generator, since all parsing behavior and parse tree construction had to be implemented manually. Unlike parser generators, which can automatically build a parse tree (or abstract syntax tree) directly from a grammar specification, a recursive descent parser requires explicit code to create, connect, and manage nodes during parsing. This increases implementation complexity, as every grammar rule must also include logic for constructing the

corresponding tree structure rather than relying on automated generation. This also leads to a larger amount of boilerplate and repetitive code, since similar node-building patterns must be written repeatedly across different grammar rules. As the language grows, this makes the codebase longer and more time-consuming to maintain and extend. It also introduces more opportunities for mistakes or inconsistencies, such as missing nodes or incorrectly structured trees, particularly when handling more complex language features. In contrast, parser generators derive much of this structure automatically from the grammar definition, resulting in a more concise implementation with less manual overhead.

Section 7: Software Development Life Cycle Model

A Test Driven Development (TDD) approach was used throughout the full lifecycle of the CatScript compiler, from initial implementation to final validation. At each stage, development was guided by test cases that explicitly defined the expected and correct behavior, and implementation was written directly to satisfy those requirements.

Early in the project, professor-provided unit tests established explicit requirements for the CatScript compiler and directed implementation of key components, including tokenization, parsing, semantic analysis, evaluation, and bytecode generation. Each failing test revealed a specific feature or behavior that had not yet been implemented, clearly identifying functionality that was missing or incomplete in the compiler. By carefully reviewing each test and analyzing its assertions and expected input-output behavior, it was possible to determine the intended behavior of each language feature and what the compiler was required to produce. Once the test requirements were understood, the corresponding code was written specifically to satisfy the test. The implementation was refined iteratively by rerunning the test after each change. After each run, targeted adjustments were made to address any failures or mismatches in behavior until the test passed successfully. Only once the test passed did development continue to the next case, ensuring each feature was fully validated before moving forward. Using this approach, we incrementally built and validated each major part of the compiler, including tokenization, parsing, semantic analysis, evaluation, and bytecode generation.

Toward the end of the project, after all professor-provided tests were passing, additional tests written by Team Member 2 in csci468.partner.PartnerTests were used as a final validation step. These tests extended coverage by targeting edge cases, more complex inputs, and interactions between language features. Running the full test suite helped confirm that the compiler behaved correctly beyond the basic requirements and remained stable under more challenging conditions. It also helped identify any remaining gaps in implementation that were not exposed by earlier tests, ensuring more comprehensive correctness. This final stage of testing increased confidence that the compiler was robust, consistent, and able to handle a wider range of valid and invalid inputs as expected.

Ultimately, one major advantage of using Test Driven Development (TDD) in this project was that it provided clear, concrete definitions of expected behavior at every stage of the compiler. Instead of relying on informal assumptions about how a language feature should work, each unit test specified exact input-output behavior that the compiler was required to satisfy. This reduced ambiguity in implementation decisions by making correctness fully test-defined. When a test failed, it immediately pinpointed which stage of the compiler did not yet match the expected behavior and was not yet producing the correct output, making it clear what specific logic needed to be added or corrected. This created a consistent feedback loop where development decisions were always grounded in observable test results, ensuring that every feature was aligned with its specification as defined by the test suite.

In addition, TDD also significantly reduced error propagation across the compiler pipeline. Since tests were run continuously as features were implemented, issues were typically caught immediately at the level they were introduced, rather than surfacing later in unrelated parts of the system. For example, a mistake in parsing or semantic analysis could be detected before it affected evaluation or bytecode generation, preventing cascading failures that would be harder to debug. Instead of tracking down interconnected issues across the entire compiler, we could isolate problems to a single feature or stage. Overall, this led to a more stable implementation process and increased confidence that each part of the compiler was correct before building on top of it.

However, one limitation of using Test Driven Development (TDD) in this project was that it tightly constrained implementation decisions to the scope of the provided tests. Because progress was primarily

measured by passing specific test cases, development often focused on satisfying exact input-output requirements rather than designing more general or extensible solutions. This sometimes led to a reactive style of implementation, where code was shaped by what the tests happened to check rather than by a broader understanding of how the language could behave. In more complex areas of the compiler, this sometimes limited exploration of alternative designs or structural improvements, since the priority remained on conforming to the test suite instead of evaluating better long-term design choices.

Notably, another limitation of using Test Driven Development (TDD) in this project was that it restricted exploration of edge cases beyond those explicitly covered in the test suite. Since development was driven by passing predefined test cases, attention was primarily directed toward the specific inputs and scenarios already specified, which reduced the motivation to proactively explore additional unusual or borderline cases during implementation. As a result, unexpected inputs or behaviors that were not represented in the original suite could be unintentionally overlooked, especially in more complex interactions between language features. This made the implementation somewhat dependent on the quality and completeness of the test coverage, meaning that any gaps in the tests could allow subtle bugs, inconsistent behavior, or unhandled cases to persist without immediate detection. In practice, this also limited confidence in full correctness, since passing all tests did not necessarily guarantee that all possible inputs or combinations had been fully considered.