Capstone Documentation

Austin Hoffman, Isaac Yauk

~ CSCI 468: Compilers ~

Section 1: Program

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

Section 2: Teamwork

This project was conducted through Test Driven Development. For an in depth review of this process refer to Section 7.

Test code I implemented for Team member 2. These can be found in their project in `csci468.partner.PartnerTests` and in their *partners* documentation - Technical Writing.

Team member 2 implemented 3 tests to my codebase. These can be found in `csci468.partner.PartnerTests` and in *partners* documentation - Technical Writing.

Section 3: Design Pattern

I used the Memoization Pattern for this project. It can be found in `csci468.parser.CatscriptType.getListType()`.

This method checks whether a type has been computed. If the type has not been computed the parser will evaluate the type through a regular chain of function calls. If the type has been computed it will instead use the cached type. This prevents the parser from making redundant type calculations for expensive function calls. The tradeoff here is primarily one of runtime versus memory. The Memoization Pattern is used to reduce the overall runtime of the parser in exchange for memory usage. This pattern is common practice in recursive decent parsing. Costs incurred in recursion often multiply due to its iterative nature. By removing redundant type checks this parser saves significant runtime.

Section 4: Technical Writing/Catscript Guide

Partner Tests

```
/**
 * Tests that the built-in length() function can be used directly
 * inside a comparison expression within an if condition.
 * Requires: length(), if/else, string variables, print()
 */
@Test
void stringLengthInCondition() {
    assertEquals("long\n", executeProgram("""
        var s = "hello"
        if (length(s) > 3) {
            print("long")
        } else {
            print("short")
        }
        """));
}

/**
 * Tests that a variable declared outside a for loop can be
 * modified inside the loop body and retain its value afterward.
 * Requires: for loop, list literal, assignment, print()
 */
@Test
void forLoopAccumulator() {
    assertEquals("15\n", executeProgram("""
        var sum = 0
        for (x in [1, 2, 3, 4, 5]) {
            sum = sum + x
        }
        print(sum)
        """));
}

/**
 * Tests that a function correctly returns different values
 * depending on which branch of an if/else is taken.
 * Requires: function definition, typed parameter, if/else,
 * return statement, function call, print()
 */
@Test
void functionReturnsFromIfElse() {
    assertEquals("pass\n", executeProgram("""
        function check(n : int) : string {
            if (n >= 60) {
                return "pass"
            } else {
                return "fail"
            }
        }
        print(check(75))
        """));
}
```

CatScript Language Reference

Overview

CatScript is a small statically typed programming language.
It supports:

- Variables and assignments
- Control flow (`if`, `while`, `for`)
- Functions with parameters and return values
- Lists and indexing
- Primitive and generic types

Program Structure

A CatScript program consists of either:

1. A single expression, or
2. A sequence of statements and function declarations.

Example

```
var x: int = 10
print(x)
```

Grammar

Program

```
catscript_program = expression | { program_statement };
```

A program may evaluate a single expression or execute multiple statements.

Statements

```
program_statement = statement |
                    function_declaration;
```

Statements define executable logic.

Supported statements:

- Variable declarations
-

- Assignments
- Print statements
- Conditionals
- Loops
- Return statements
- Function calls

Variable Declaration

Declares and initializes a variable.

Syntax

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

Examples

```
var x = 5
var name: string = "cat"
var numbers: list<int> = [1, 2, 3]
```

Notes

- Type annotations are optional if inferable.
- Variables must be initialized.

Assignment

Assigns a new value to an existing variable.

Syntax

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

Examples

```
x = 20
numbers[0] = 42
```

Print Statement

Outputs an expression.

Syntax

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

Example

```
print("Hello World")
print(x + 5)
```

Conditional Statements

If Statement

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

Examples

Basic If

```
if (x > 5) {
    print("large")
}
```

If / Else

```
if (x > 5) {
    print("large")
} else {
    print("small")
}
```

Else If

```
if (x > 10) {
    print("big")
} else if (x > 5) {
    print("medium")
} else {
    print("small")
}
```

Loops

While Loop

Executes while condition is true.

Syntax

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

Example

```
while (x < 10) {
    x = x + 1
}
```

For Loop

Iterates over a list.

Syntax

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

Example

```
for (item in numbers) {
    print(item)
}
```

Functions

Functions define reusable blocks of code.

Declaration

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

Example

```
function add(a: int, b: int): int {
    return a + b
}
```

Parameters

```
parameter_list = [ parameter, {',' parameter } ];
parameter = IDENTIFIER [':', type_expression];
```

Example

```
function greet(name: string) {
    print(name)
}
```

Return Statement

Returns from a function.

Syntax

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

Examples

```
return
return 42
return name
```

Function Calls

Invokes a function.

Syntax

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

Examples

```
print(add(1, 2))
greet("Milo")
```

Expressions

Expressions evaluate to values.

Expression precedence (highest to lowest):

1. Primary expressions
2. Indexing
3. Unary operators
4. Multiplication / division / modulo
5. Addition / subtraction
6. Comparisons
7. Equality

Equality

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

Example

```
x == 10
name != "cat"
```

Comparison

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

Example

```
x > 5
score <= 100
```

Arithmetic

Addition/Subtraction

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

Multiplication/Division/Modulo

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

Examples

```
1 + 2
10 - 3
4 * 5
20 / 4
10 % 3
```

Unary Operators

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

Examples

```
not true
-x
```

Lists

Lists store ordered collections.

Literal Syntax

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

Examples

```
[]
[1, 2, 3]
["a", "b", "c"]
```

Indexing

Access list elements by position.

Syntax

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

Example

```
numbers[0]
matrix[1][2]
```

Types

CatScript is statically typed.

Primitive Types

Type	Description
`int`	32-bit signed integer
`string`	UTF-16 Java-style string
`bool`	Boolean (`true` or `false`)
`null`	Null value

Generic Types

List

`list<T>`

Stores values of type `T`.

Examples:

```
list<int>
list<string>
list<list<int>>
```

Object Type

```
object
```

Represents any value.

Useful for generic storage or mixed types.

Literals

Supported literals:

Integer

```
42
-7
```

String

```
"hello"
"catscript"
```

Boolean

```
true
false
```

Null

```
null
```

Comments

CatScript does **not define comment syntax** in its grammar.

Example Program

```
function factorial(n: int): int {
    if (n <= 1) {
        return 1
    }
```

```
    return n * factorial(n - 1)
}

var result = factorial(5)
print(result)
```

Output:

```
120
```

Language Rules

Static Typing

All expressions have a compile-time type.

Examples:

```
var x: int = 5
var y: string = "hello"
```

Invalid:

```
var x: int = "hello"
```

Type Compatibility

Rules:

- `null` may be assigned where allowed by implementation
- `object` accepts all types
- `list<T>` requires elements of type `T`

Reserved Keywords

```
var
function
return
if
else
while
for
in
print
true
false
null
not
int
string
bool
```

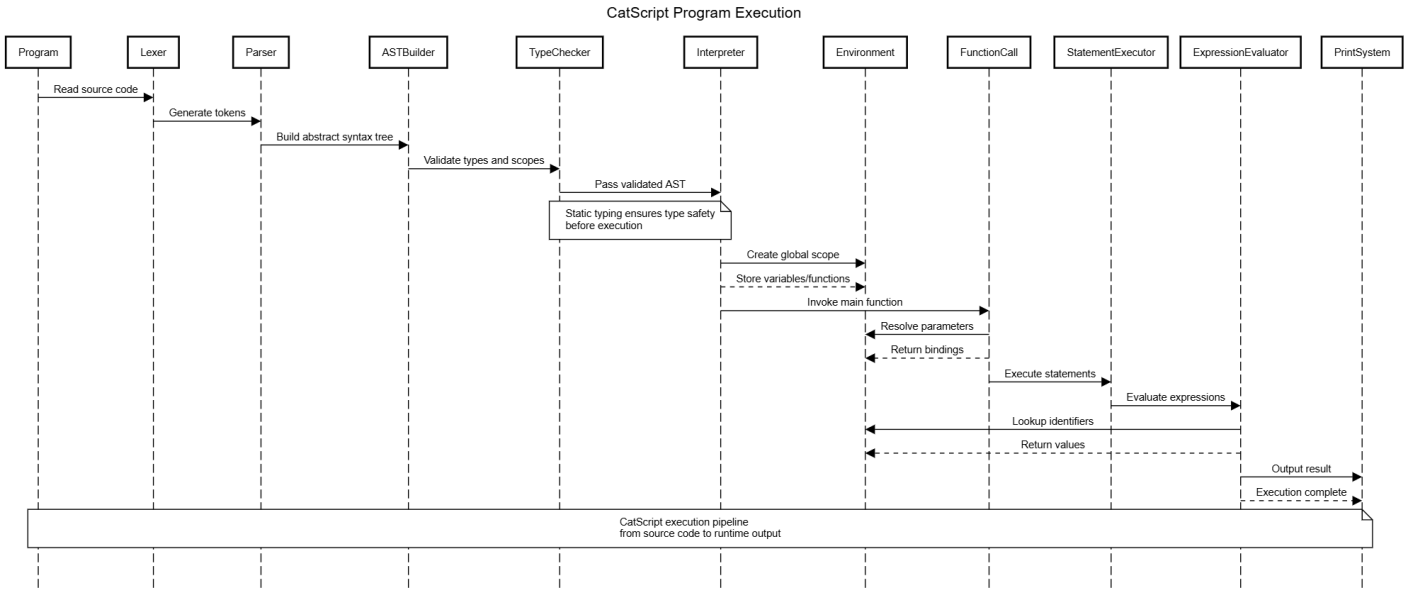
```
object
list
```

Summary

CatScript provides:

- Static typing
- Functions
- Variables
- Lists
- Loops
- Conditionals
- Expressions

Section 5: UML



Section 6: Design Trade-offs

A handwritten recursive-descent parser gives fine-grained control over parsing behavior. It can be tailored to specific applications, providing robust error messaging and type control. Edge cases are handled deterministically, grammar is predictable in a well-defined abstract syntax tree, and since code mirrors grammar, it is readable. It is often easier to understand the parsing pipeline of a handwritten recursive-descent parser than a parser generator because of these factors.

However, handwritten parsers come at a cost. As the grammar grows, parser code can become repetitive and error-prone. More development time is needed to implement a handwritten recursive-descent parser. Handling recursion correctly takes careful consideration. Errors can be subtle and hard to diagnose because parser logic and grammar are often interlinked. A parser generator automatically creates parser code from written grammar specifications. This reduces overhead significantly.

Catscript requires fine control to demonstrate learning from implementation. Automating this process would undermine this result. The better option became:manually implement Catscript using recursive descent.

Section 7: Software Development Life Cycle Model

Test-Driven Development (TDD) Coordination

My partner and I used Test-Driven Development as our primary workflow throughout the development of CatScript. We coordinated by reviewing the project requirements. Then we decided which behaviors to test for. We wrote tests that captured the expected behaviors of potential new features. These tests were accompanied by documentation. Documentation included any prerequisite logic needed to run the test and the test's architecture. After exchanging tests, each of us implemented the necessary code to pass them. We shared updates, including test modifications and missing dependencies. Since every feature started with tests, we could clearly define its expected end behavior. Here is a breakdown of some trade-offs we faced with Test-Driven Development:

Benefits:

- Test bases could be built incrementally.
- Having redundant developers led to more robust testing and debugging.

Detriments:

- Coordination was informally decided on. This often delayed development.
- Having redundant developers reduced development speed.