

Capstone Portfolio

CSCI 468: Compilers

Montana State University

Spring 2023

Brady Ash and Turner Burchard

Section 1: Program Specifications

Grammar:

```
catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

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

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;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

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

function_body_statement = statement |
                        return_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_expression };

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

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

unary_expression = ( "not" | "-" ) unary_expression |
primary_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 , '>']

```

Section 2: Teamwork

Team members: Turner Burchard, Brady Ash

Brady consistently kept up with the predefined checkpoints in his code throughout the whole semester. He wrote 100% of his code on his own, and passed all of the required tests for Catscript working. Brady did 80% of the documenting and testing for Turner's codebase, and was very helpful along the way. In short, we each did the coding for our own projects, and did most of the documentation and testing for each other.

Section 3: Design Pattern

In our program, we used the Memoization pattern. It is located in CatscriptType.java. We used this pattern as it saves our program computation. It is a form of caching, meaning that if we have already created the type we are requesting, we can grab it from where we stored it in memory instead of creating a new one each time. While this is very useful, it would break down if we were to implement multi-threading - a definite design trade-off. The following is the implementation of the Memoization pattern:

```
public static HashMap<CatscriptType, CatscriptType> LIST_TYPES = new
HashMap<>();

public static CatscriptType getListType(CatscriptType type) {

    CatscriptType listType = LIST_TYPES.get(type);
    if (listType == null) {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}
```

Section 4: Technical Writing

Introduction

Catscript is a language based on a simplified version of C formatting, with static typing. A simple example of a function looks like this:

```
function foo(x:string) {
    print(x)
}

var x = "Hello World"
// This is a comment
foo(x)
```

Result:

Hello World

CatScript Language Specification

CatScript is a simple, statically typed programming language that allows you to write programs with ease. It provides basic programming constructs such as loops, conditions, functions, and data types.

Comments

A comment can be created in Catscript with a “//”

Anything on a line in Catscript following a “//” will not be tokenized, parsed, nor compiled by the compiler, it will be completely ignored.

Types

These are the main types available in Catscript:

- Int
- String
- List
- Bool
- Object
- Null
- void

Lists can also be recursively defined with other types, to create for example a List of Objects

Program Structure

The highest level structure of Catscript programs is a “statement.” These statements are split into two categories, function declaration statements and program statements. They are split in order to maintain the context/scopes of functions. The program statement category has the following subcategories of statements:

- for statement
- if statement
- print statement
- variable declaration statement
- assignment statement
- function call statement

Control Structures

For Statement

The for statement allows you to loop over an iterable expression. The syntax of the for statement is as follows:

```
for (identifier in expression) {  
    statements  
}
```

The identifier must be a valid identifier and will be assigned each value in the expression. The statements inside the block will be executed for each value.

If Statement

The if statement will evaluate an expression and only execute the following statements if the expression evaluates to true, allowing conditional execution as follows:

```
if (expression) {  
    statements  
}
```

The if statement can also be followed by an else block, which will trigger execution when the if expression evaluates to false, which looks like this:

```
if (expression) {  
    statements  
} else {  
    statements  
}
```

There is currently no support for elif statements, which are common in other languages. However, if statements can be nested to create the same functionality.

Statements

print Statement

The print statement allows you to output a value to the console. The syntax of the print statement is as follows:

```
print(expression)
```

The expression can be any valid expression that evaluates to a value.

Variable Declaration Statement

The variable declaration statement allows you to create a new variable and assign a value to it. The syntax of the variable declaration statement is as follows:

```
var identifier [: type_expression] = expression
```

The identifier must be a valid identifier, and the `type_expression` (optional) must be one of the following data types: `int`, `string`, `bool`, `object`, or `list`. The expression must be an expression that evaluates to a value.

Assignment Statement

The assignment statement allows you to assign a new value to an existing variable. The syntax of the assignment statement is as follows:

```
identifier = expression
```

The identifier must be an existing variable, and the expression must be an expression that evaluates to a value.

Function Call Statement

The function call statement allows you to call a function. The syntax of the function call statement is as follows:

```
function_call([argument_list])
```

The `function_call` must be a valid function call expression. Each of the arguments must match the expected type of the function, and there must be the correct number of arguments (possibly zero).

Return Statement

The return statement is used to exit a function and return a value. When the return statement is executed, it immediately terminates the function and returns the specified value to the caller. The syntax of a return statement is as follows:

```
return [expression];
```

Parameters

`expression` (optional): The value to be returned from the function. If omitted, the function returns `null`.

Examples

Return a value from a function:

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

Return null from a function:

```
function doSomething() {  
    // statements  
    return  
}
```

Notes

- The usage of the return statement is limited to within a function.
- Functions have the capability to return only a singular value. However, you can bundle multiple values by enclosing them in an object, an array, or a similar data structure.
- In case a function lacks a return statement, it will return null by default.
- It is possible to use the return statement with any data type, such as objects, arrays, and functions.
- In the event that the data type of the expression given to the return statement does not match the expected type (e.g., if a function anticipates an integer but the expression is a string), a runtime error will arise.

Functions

Function Declaration

The function declaration statement creates a new function. The syntax of the function declaration is as follows:

```
function identifier (parameters) [: type_expression] {  
    statements  
}
```

The identifier must be a valid identifier, and the parameters are a comma-separated list of parameter declarations. The `type_expression` (optional) must be one of the following data types: `int`, `string`, `bool`, `object`, or `list`. Inside the function are an unlimited number of statements which are executed when the function is called.

Parameter Declaration

A parameter declaration allows you to declare a parameter for a function. The syntax of a parameter declaration is as follows:

```
identifier [: type_expression]
```

Expressions

There are also various expressions in Catscript. The following types of expressions are available:

- Equality expression
- Comparative expression
- Additive expression
- Unary expression
- Primary expression

These are the most basic building blocks of Catscript, and all are evaluated to primary expressions at compile time. A primary expression is one of the following:

- Identifier
- String
- Integer
- True
- False
- Null
- List literal
- Function call

Code Tests written by Turner Burchard:

```
package edu.montana.csci.csci468;

import org.junit.jupiter.api.Test;

import edu.montana.csci.csci468.CatscriptTestBase;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class PartnerTests extends CatscriptTestBase {
```

```

@Test
void test1() {
    assertEquals("[4, 6, 8]\n", executeProgram("function foo(x : int) { return [x*2, x*3, x*4] }" +
        "var a = foo(2)" +
        "print(a)"));
}

@Test
void test2() {
    assertEquals("9\n", executeProgram("for( x in [1, 2, 3] ) {\n" +
        "  var y = x * 3\n" +
        "  if(y > 7) {\n" +
        "    print(y)\n" +
        "  }\n" +
        "}\n"));
}

@Test
void test3() {
    assertEquals("64\n32\n16\n8\n4\n2\n1\n0\n", compile(
        "function foo(x : int) {\n" +
        "  print(x) +\n" +
        "  if(x > 0) {\n" +
        "    foo(x / 2)\n" +
        "  }\n" +
        "  }\n" +
        "foo(64)"
    ));
}
}

```

Section 5: UML

The first UML diagram describes what happens when a simple comparative expression is evaluated. The expression is turned into tokens by the lexer, each token is parsed by the required parsing functions. Once it is all parsed, the expression is executed, evaluating it to a boolean. This is then returned back up the recursive parser, which results in the user getting back the evaluated value. The second UML diagram describes the overall structure of the compiler, with all of the types of statement and expressions.

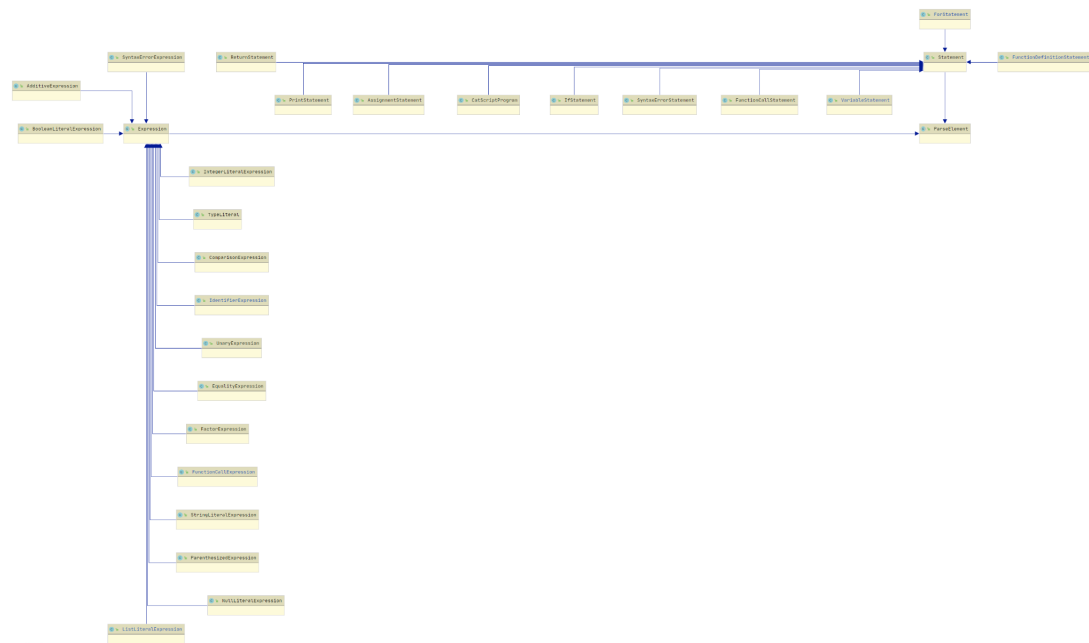
Catscript Comparison Sequence Diagram

```

sequenceDiagram
    participant User
    participant CatScript
    participant Lexer
    participant Parser
    participant ParseTree

    User->>CatScript: evaluate("2 > 1")
    activate CatScript
    CatScript->>Lexer: lex("2 > 1")
    activate Lexer
    Lexer-->>CatScript: tokens
    deactivate Lexer
    CatScript->>Parser: parse(tokens)
    activate Parser
    Parser->>Parser: parseExpression()
    Parser->>Parser: parseComparativeExpression()
    Parser->>Parser: parseIntegerLiteral()
    Parser-->>CatScript: parseTree
    deactivate Parser
    CatScript->>ParseTree: execute()
    activate ParseTree
    ParseTree-->>CatScript: true
    deactivate ParseTree
    CatScript-->>User: true
    deactivate CatScript
  
```

The diagram illustrates the sequence of operations for evaluating a comparison expression in Catscript. The process begins with the User sending an `evaluate("2 > 1")` message to CatScript. CatScript then sends a `lex("2 > 1")` message to the Lexer, which returns `tokens`. CatScript then sends a `parse(tokens)` message to the Parser. The Parser performs three recursive calls: `parseExpression()`, `parseComparativeExpression()`, and `parseIntegerLiteral()`, before returning a `parseTree` to CatScript. CatScript then sends an `execute()` message to the ParseTree, which returns `true`. Finally, CatScript returns `true` to the User.



Section 6: Design Tradeoffs

We made the decision to utilize a recursive descent parser over a parser generator. While both have their merits, we believed that the recursive descent parser would give us better results. Parser generators do their job by taking in a grammar, and generating code based on it. These are useful tools that usually require less code than our alternative, and as a result we have less to get right. However, since these are programs that write our code, we get further from the machine than we would like. The syntax can often be confusing and hard to read, and isn't a huge player in the Computer Science industry. For those reasons, we went with the recursive descent parser. This method offers us more customization based on our language, and lets us stay close to the machine in terms of how we code and interact with it. While we have plenty more to get right, the flexibility and recursive nature of the recursive descent parser are well worth the extra lines of code and extra hours. It also gets us up close and personal with how exactly parsing works, which is a great learning experience.

Section 7: Software Development Lifecycle

We used test driven development to develop our compilers. This is not a new system for me, I have used it both in my work and in previous classes. In general, I think it works extremely well for creating production software. This is especially true in complex projects like this, where there is a huge codebase with many moving parts. Changing one thing can unexpectedly cause rippling effects throughout the code, so tests are necessary to ensure everything is still working. However, it does tend to lead to a trial-and-error methodology of coding.