

Capstone Portfolio

CSCI 468: Compilers

Montana State University

Spring 2023

Brady Ash and **Turner Burchard**

Section 1: Program Specifications

All the code written for this project is included in this directory as “source.zip”

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

Turner worked well, finishing his project with plenty of time to spare. Turner would get a start early, and continue steady progress until each portion was completed. Turner's focus was on the coding portion of his project, doing at least 90% of the coding for his own project. If help was needed, Brady did assist with a couple segments of code. Turner did the large majority of the documenting and testing for Brady. After finishing the Bytecode tests, while he was working on Compilers, he spent probably 80% of his time documenting and testing. He was thorough, and provided both constructive feedback and praise for Brady. Similarly, Brady did 80% of the testing and documenting for Turner's code, including most of this document. Some of the testing Brady did for Turner is in the bottom of this document, three Java tests. Overall, each partner wrote almost all of the code for their own compiler, and did almost all of the documenting and testing for each other's compilers. We each believe that the teamwork was split evenly, and we worked well together.

Section 3: Design Pattern

One design pattern that we used in our code was the Memoization pattern. This was used for the typing system, in `CatscriptType.java`. It is a useful design pattern for optimization, as it reduces the overall amount of computations required in each compilation. The types are cached, so that if there is already a type in memory of the same type that we need later on, it can just be used from memory instead of creating a new one every time. This design pattern was great for speeding up the compiler, which can sometimes take a long time due to the inherent levels of complexity present. However, it could cause problems in the future if we were to change the way typing is done in Catscript, and the overhead of creating a cache and storing the values may have not been worth it, especially in short programs where only one or two types are actually used. It also would not work at all if we were to make the compiler multithreaded. The following is the implementation used for Memoization:

```
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 simple statically typed, functional language. Here is an example:

```
var x = "foo"  
print(x)
```

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.

Types

Catscript provides a short list of types, as follows:

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

Program Structure

A CatScript program is a collection of statements. A statement is either a function declaration or a program statement. A program statement can be any of the following:

- 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 a range of values. 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 allows you to conditionally execute a block of statements. The syntax of the if statement is as follows:

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

You can also use an else block to execute a different set of statements if the condition is not met. The syntax for an if-else statement is as follows:

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

You can also nest if-else statements to create more complex conditions.

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
```

The function_call must be a valid function call expression.

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() {  
    // Do something...  
    return;  
}
```

Notes

- The return statement can only be used inside a function.
- A function can only return one value. If you need to return multiple values, you can use an object, an array, or another data structure to encapsulate them.
- If a return statement is not used in a function, the function will return null by default.
- The return statement can be used with any data type, including objects, arrays, and functions.
- If the expression provided to the return statement is not of the expected type (e.g., if a function expects an integer and the expression is a string), a runtime error will occur.

Functions

Function Declaration

The function declaration allows you to create a new function. The syntax of the function declaration is as follows:

```
function identifier (parameter_list) [: type_expression] {  
    // function_body_statements  
}
```

The identifier must be a valid identifier, and the parameter_list is 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. The function_body_statements are the statements that make up the function body.

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]
```

Section 5: UML

The first UML diagram in this section is the diagram which displays the overall structure of the compiler. In general, the compiler is made up of many different categories of expressions and statements, which each serve a unique function. The second UML diagram is an example of a more detailed sequence diagram for a specific functionality, in this case the If Statement. In this statement, the code is first tokenized by the lexer, which then passes those tokens to be parsed by the parser with the recursive parsing functions, and is finally compiled and executed to give the result of "20".

Section 6: Design Tradeoffs

We had to choose between a parser generator and a recursive descent parser to parse our code. After careful consideration, we determined that the latter was more suitable for our needs. Parser generators generate code based on a given grammar and generally require less coding than the recursive descent parser. However, we determined that this method created greater separation between us and the machine, and the syntax of parser generators can be difficult to read and comprehend. In contrast, the recursive descent parser enables us to tailor our approach precisely to our language and maintain a closer relationship with the machine. Despite requiring more code lines and time, the flexibility and recursive nature of this parser make it well worth the investment. Moreover, this technique affords us a deeper understanding of the parsing process, which is an invaluable learning experience. Choosing the right parser is a vital step in developing high-quality code that performs as expected, and the recursive descent parser has become increasingly popular in recent years due to its flexibility and ability to handle complex grammars efficiently.

Section 7: Software Development Lifecycle

The development of our compilers was facilitated through the utilization of test-driven programming. Although not novel to me, as I have previously applied this methodology in my past coursework and professional endeavors, I have found it to be highly effective in creating software for practical use. This is especially so for intricate projects such as this, in which an extensive codebase contains numerous interconnected components. Since alterations to even one aspect can generate unforeseen impacts throughout the code, testing is critical in guaranteeing that everything continues to function correctly. Nevertheless, this approach often results in a trial-and-error approach to coding. Instead of test-driven development, we could have used a method like Agile development, where we have a list of tasks that need to be completed. I think test driven development worked a lot better than anything else would, since it ensured that everything worked as expected and let us do slow, incremental development over the course of the semester. Having clear, code-based requirements made it very easy to get everything done, and be certain we were writing quality code.

Partner tests written by Brady:

```
@Test
void forInsideIfWorksProperly() {
    assertEquals("1\n2\n3\n", executeProgram("var x = 10\n" +
        "if (x > 0){ \n" +
        "for ( y in [1,2,3]) { print(y) }\n" +
        "});");
}
```

```
@Test
void ifInsideFunctionWorksProperly() {
    assertEquals("20<\n42\n", executeProgram("function foo() : int {\n" +
        "    var x = 42\n" +
        "    if (x > 20) { print(\"20<\") }\n" +
        "    return x\n" +
        "}\n" +
        "print( foo() )\n"));
}
```

```
@Test
void nestedForStatementWorksProperly() {
    assertEquals("1\n2\n1\n2\n1\n2\n", executeProgram("for(x in [1, 2, 3]) {\n" +
        "    for(y in [1,2]) { print (y)}\n"));
}
```