

Cover Page

Compilers - CSCI 468

Spring Semester

Andrew Anselmo

Team member: Gabe Ewsuk

Section 1: Program - include a link to the zip file mentioned above

See attached zip file

Section 2: Teamwork - Discuss your partners documentation and testing contribution, as well as your primary work on the project

My partner Gabe Ewsuk acted as the primary documentation / testing engineer over my project. His responsibilities to my project included writing the documentation for the language that we were writing our compiler for, Catscript, and writing three unit tests, which I embedded in my testing folder in my project src zip

(csci-468-spring2023-private/src/test/java/edu/montana/csci/csci468/CapstoneTest). The purpose of the documentation is to get anyone up to speed on the syntax and semantics of Catscript if you have never used this programming language. The documentation starts off by introducing the Catscript programming language. Then my teammate details each of the major features in Catscript between the expressions and statements. For each feature, he provided an example of a code snippet, the parameters, what it returns, and the general form. As I previously mentioned, my teammate was also responsible for writing three unit tests to ensure my compiler was working properly. My teammate wrote a test to ensure that a for loop and an if statement works in a function that takes multiple arguments. Additionally, he wrote me a test to see if it is possible to call a function from within a statement. Lastly, he wrote me a test to see if a function could take multiple boolean arguments. In the scheme of the whole project, I wrote the entire compiler, which was about 90% of the project, and Gabe was responsible for the unit testing and documentation, which was 10% of the project.

Section 3: Design pattern

A design pattern that we implemented in our projects was the memoization design pattern. The purpose of using memoization is to reduce the amount of redundancy in a program. Instead of repeating a task to get a result we have already found, we cache all of our results in a hash map so we can search for a pre-computed quickly and easily. In this example we are using the memoization design pattern to return list types. Before memoizing this function, a new list type would be created and returned everytime the function was called. Now, the user sends a type to this function and the program searches through the hashmap of already created list types for a given type. If a type has previously been sent to the function, we created a list type for it and stored it in the hash map. The program will look for a match in the hash map for that given type and will simply return it if it finds a match. Otherwise we will create a new type, store it in the hashmap for future use, and then return that to the user. This is an example of how design patterns such as the memoization design pattern can save us time and memory in a program.

Memoization design pattern implemented:

```
2 usages
public static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
4 usages  ▲ Carson Gross *
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 - Include the documentation generated by your partner for the catscript programming language

Catscript Documentation

Introduction

Catscript is a lightweight, compiled programming language that is built on Java. Catscript is a functional-based programming language with key aspects you may see within other languages such as arithmetic, if statements, for loops, functions, comparison operators, and a type system.

Features

Statements -

For Statement:

The For statement often referred to as the for loop is used for iterating over values that can be accessed in the statement.

General Form:

```
for (IDENTIFIER in expression )
    { statement }
```

Example

```
for ( x, in, [1,2,3])
    {print(x) }
Output: 1\n2\n3\n
```

Parameters

- Identifier
- expression
- statement

Returns

Commands executed within the statement.

If-else Statement:

The if statement is a conditional that is used to execute when the conditions are met, and the else is a complement to the if so that if it fails, the else statements are executed.

General Form

```
if ( expression ) {
    { statement }
} [ else ( { { else statement } } ) ]
```

Example:

```
if(true){
print(1)}
else{print(2)
}
output:1\n
```

Parameters

- expression
- statement
- Optional-
- else statement

Returns

Executes if statements when conditions are met, or else statements if they are not met.

Print Statement:

The print statement allows expressions to be output to the console.

General Form:

```
print(expression)
```

Example

```
print("Hello Word")  
Output: Hello World
```

Parameters

-expression

Returns

output to the console.

Variable Statement:

The variable statement allows the user to define expressions under variable names, these variables can have types as shown in the general form.

General Form:

```
var IDENTIFIER  
    : type_expression = expression
```

Example

```
var x : int = 1;
```

Parameters

-Identifier

- expression
- type expression

Returns

Adds a variable with a type onto the stack available for access later.

Assignment Statement:

An assignment statement is similar to a var statement, but the type is dynamic and pertains to the type of the expression.

General Form:

```
IDENTIFIER = expression
```

Example

```
x = 1
```

Parameters

- Identifier
- expression

Returns

Adds a variable to the stack available for access later

Function Declaration Statement:

The function declaration statement is used for defining a function that can hold a barrage of statements. This allows the programmer to abstract what is happening. Executing dozens of lines of code by calling one function.

General Form:

```
function IDENTIFIER(parameter_list): type_expression {  
    function_body_statement }
```

Example

```
Function foo(x:int):int  
{  
    print(x)  
    return(x)  
}
```

Parameters

- Identifier
- parameter list
- function body statement
- Optional-
- type expression

Returns

Creates a function that is later accessible through function call statements for execution.

Function Call Statement:

The function call statement calls the function that was previously declared and executes the statements.

General Form:

```
IDENTIFIER (argument_list)
```

Example

```
foo("hello", "world")
```

Parameters

- Identifier
- argument list
- A previously defined function

Returns

Executes the statements in the function called and potentially returns an expression if explicitly stated in the expression.

Return Statement:

The return statement allows the function to hand a value back to the program. This can later be accessed if the function call is set to a variable or printed to the console.

General Form:

```
return expression
```

Example

```
Function foo(x:int):int  
{  
  print(x)  
  return(x)  
}
```

Parameters

- expression

Returns

Returns the expression given in the expression when the function is called by the function call statement

EXPRESSIONS

Equality Expression:

The equality expression is used for comparing whether two expressions are equal or not equal ie. !=.

General Form:

```
expression == expression
```

OR

```
expression != expression
```

Example

```
if(1==1){print(true)}  
if(1!=2){print(false)}
```

```
output:true\nfalse\n
```

Parameters

-Comparison Expression (Right and Left-Hand side)

Returns

Returns true or false based on if the comparison expressions meet the requirements or not

Comparative Expression:

The Comparative expression is used for checking whether an integer is greater than, greater than or equal to, less than, & less than or equal to. It returns a boolean value and therefore is great for conditional statements.

General Form:

```
expression{ (">" | ">=" | "<" | "<=" ) expression };
```

Example

```
if(1>1){print(1)}  
if(1>=1){print(2)}  
if(0<1){print(3)}  
if(3<=1){print(4)}
```

output:2\n3\n

Parameters

-additive expression (Right and Left-Hand side)

Returns

Returns true or false based on if the comparison expressions meet the requirements or not.

Additive Expression:

The additive expression allows the user to perform addition or subtraction on expressions.

General Form:

```
expression { ("+" | "-" ) expression
```

Example

```
X = 1+1  
Y = 2-1  
print(X)  
print(Y)
```

Output: 2\n1\n

Parameters

-factor expression (Right and Left-Hand side)

Returns

Performs arithmetic expression of addition or subtraction of 2 factor expressions and then returns the result.

Factor Expression

The Factor expression allows the user to perform arithmetic in the form of division or multiplication on expressions.

General Form:

```
expression { ("/" | "*" ) expression }
```

Example

```
X = 4/1
Y = 2*1
print(X)
print(Y)
```

Output: 4\n2\n

Parameters

-unary expression (Right and Left-Hand side)

Returns

Performs arithmetic expression of division or multiplication of 2 unary expressions and then returns the result.

Unary Expression:

The unary expression allows the user to negate a boolean or create negative integers

General Form:

```
( "not" | "-" ) unary_expression | primary_expression
```

Example

```
print(not(true))
print(-(1))
```

Output: false\n-1\n

Parameters

-unary expression or primary expression

Returns

Not returns the negation of a boolean value, - returns the negative of an integer

Lexical 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 } '];

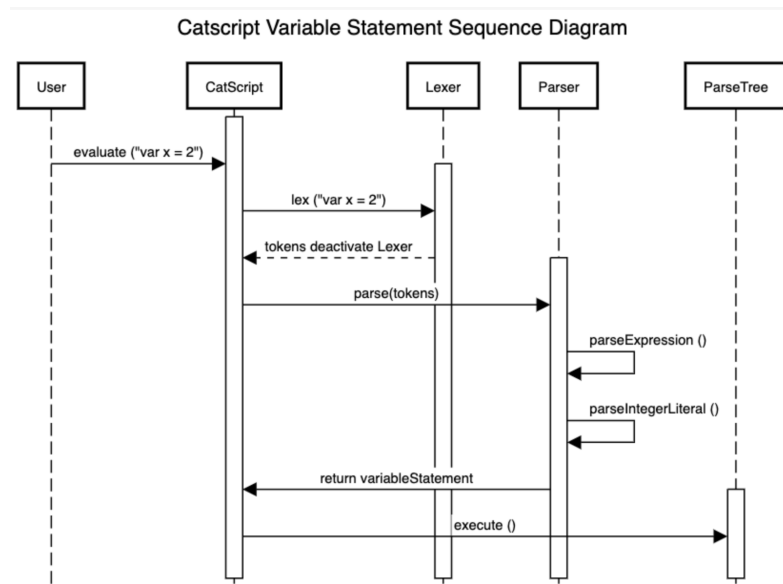
function_call = IDENTIFIER , '(' , argument_list , ')'

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

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

Section 5: UML - Discuss

For my UML diagram I wrote a sequence diagram to display how a variable statement works in catscript. First the user does some variable assignment somewhere in the program, like “var x = 2”. From there the Catscript program layer receives that call and invokes the lexer to get individual tokens from the statement. A “tokens deactivate lexer” call is made back to the Catscript program layer before invoking the parser on the tokens. Once in the parser, the program will work its way through the recursive descent methods, which give our tokens metadata and create a new variableStatement object. From there, the Catscript program layer sends the variableStatement object to the parseTree where it will execute, then compile, and then we have a complete variable statement ready to be used in other places in the program.



Section 6: Design trade-offs

One of the major design trade-offs that we made in our projects has to do with the expression problem. The expression problem describes how we can have it be easy to add a new operation, and hard to add a new class or we can have it be hard to add a new operation, and easy to add a new class. Throughout the course of the class we have been referencing the book *Crafting Interpreters* by Robert Nystrom and in his book he prefers to have it be easier to add operations and harder to add new classes. To achieve this he uses something called the visitor pattern. Through his implementation if you needed to add a new class then you update all the visitors and if you need to add a new operation, you can add a new visitor. The visitor pattern does a great job of upholding the principle of separation of concerns. Separation of concerns is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern. In our project we decided to not use the visitor pattern and opted to just put individual methods into every class and use method overriding where needed. While we

are violating the principle of separation of concerns we are addressing many other concerns that are arguably more important such as simplicity, ease of debugging, locality of behavior, and ease of reading. By not using the design pattern we are also making it harder to add new operations, however, with modern IDE's such as IntelliJ, this is actually not all that complicated to so it is an easy trade-off to cope with.

Section 7: Software development life cycle model

For our method of software development we used test driven development. Test driven development is a software development practice that focuses on creating unit test cases before developing the actual code. It is an iterative approach combining programming, unit test creation, and refactoring. In this case the test cases had been pre-written for us and we had to write the code to pass the tests. The test driven cycle goes as follows. First you go to a file full of test cases for a specific file, you run them all to see what is failing to see what you should be implementing. The test cases are often very helpful in telling you what you should be implementing. For instance, you may have a test case that wants to check that a method can properly add $2 + 2$ to get a result of 4. You would then go over to the file where the actual implementation needs to be done and you would fill out the code for the method you are testing. Once you have some code you think would pass you can run the test again. If the test fails you know that you did not do the implementation properly and you need to write better code before running the tests again. If it passes you are able to refactor the code to optimize it or improve it in any way and then you can test it again. You will work your way through all of the test cases which should cover all the methods/ functions in a given class or file. This is the basic cycle of test driven development. Test, code, analyze, refactor and then did it all over again.