Hayden Perusich
4/28/2025
Compilers, CSCI 468
Spring 2025

Section 1 - Program :

Compiler zip link: source.zip

Section 2 - Teamwork:

For teamwork on our compiler project me and Aiden Rasmussen collaborated on the technical documentation for catscript. We also created a custom test for each other to test our finished compiler.

From this experience I learned that working with a partner and larger scale projects allows you to create a new perspective on how certain tasks can be done and how to more efficiently and semessly get more done. I found that Aiden's tests tested parts of my code some of my tests didn't go through. This lets me create a more robust and reliable system.

Section 3 - Design Pattern:

In this compiler project I used memoization in the getListType function. This function is used to assign types to a list in the tokenizer. For list<int> , going through and doing the necessary logic to place “int” as an INTEGER can take a bit of time.

To speed up this process I use memoization which is a design pattern for optimization that caches the result of logic.If the same inputs are seen again this function will be able to use the hashmap to return the saved value rather than running through the logic again. This is able to speed up the time it takes to getListType as possible returns are saved rather than having to run the same function with the same input more than once.

```
// memoize this call
    static Map<CatscriptType, CatscriptType> MEMOIZATION_CACHE = new
HashMap<>();
    public static CatscriptType getListType(CatscriptType type) {
        CatscriptType existingType = MEMOIZATION_CACHE.get(type);
        if (existingType != null) {
            return existingType;
        }else{
            ListType listType = new ListType(type);
            MEMOIZATION_CACHE.put(type, listType);
            return listType;
        }
    }
```

Section 4 - Technical Writing:

CatScript Grammar Documentation - Hayden Perusich and Aiden Rasmussen

Catscript uses 32 bit integers, java style strings, null types, and object types of any value. It's also statically typed, meaning that variables are declared at compile time. This makes it so that users must declare variable types. The compiler for CatScript is written in recursive descent which the grammar illustrates.

Programs

In CatScript all programs start with the

```
catscript_program = { program_statement };
```

as the head. This denotes that one catscript program is made of 0+ program_statements.

A program_statment is either a statement or a function_decloration.

```
program_statment = statement | function_decloration.
```

This lets you create a program with a function as the head or just a line of statements depending on the use case.

Statements:

A statement is many of either an if_statement, print_statement, variable_statement, assigment_statement, return_statement, or a fuction_call_statement. This can look like

```
print(1); or if (true) { return true };
```

Along with many others.

For Statements

A for_statement is made up of

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

This looks like for (x in [1,2,3]){ print(x) } which would print 1 2 3. This allows users to loop through elements and run statements more than once.

If Statement

An if_statement is made up of

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

This looks like if(true) { return (true) }. This statement type allows users to branch their code allowing for multiple possible code chunks to be executed depending on some expression.

Print Statement

A print_statement is made up of

```
print_statment = 'print', '(', expression, ')';
```

This statement allows users to print expressions to the terminal allowing them to view outputs from the compiler and allow for basic debugging. The print statement is used like print(1) which would print “1\n” to the terminal.

Variable Statement

A variable_statement is made up of

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

This allows users to store values onto a field or slot allowing them to use those values later. In practice this looks like var x = 1;. Here the value 1 is stored into a field because it's global. This value can be changed and used later.

Function Call Statement

A function_call_statement is made up of only one funciton_call.

```
Function_call_statement = funciton_call
```

A fuction_call’s grammar is

```
function_call = IDENTIFIER, '(', arugment_list , ')';
```

This call allows the user to call functions and specify a return type. The argment_list allows for multiple inputs into a function_call.

Assignment Statement

The assigment_statment implements variable assignment which allows the user to use IDENTIFIERS to store values.

```
assigment_statement =  IDENTIFIER , '=' expression;
```

An example would be var x = 1 which would store the int 1 into memory under the variable name x. That way you could call x later in the code and 1 would be copied from memory.

Function Declaration

For the function_decloration the grammar goes.

```
function_decloratoin = 'function', IDENTIFIER. '(',
```

```
parameter_list, ')' + [ ":" + type_expression  ], '{', { statement },  '}';
```

This declaration allows users to write chunks of code with parameters and run them with the function call. An example would be, function foo (x) { print(x) }\n foo(1). This would print the variable x which in this example would be 1.

Parameter List

The parameter_list menditon above is described as

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

This allows more than one parameter to be inputted into statements such as the function_decloration such as, foo (1, 2, 3).

A parameter is written as

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

This implements a variable with the possibility of a type. If the type isn't explicit it will be assigned as compile time.

Return Statement

For a return which can be used in a function declaration its written as

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

This allows the function to return expressions as output to the global scope. function foo(x) {return x;}\n print(foo(1)) would print 1 to the terminal.

Expressions:

A expression is made up of equality expressions

```
expression = equality_expression;
```

Equality Expression

The equality_expression is outlined as

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

This would be like true == true which would compile to true.

Comparison Expression

The comparison_expression implemented in the equality_expressoin is outlined as

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

This adds the function of comparison for integers which will be evaluated to a boolean value. For example 1 > 2 would be evaluated to false.

Additive Expression

The additive_expression added addition and subtraction to the system.

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

This allows you to add 2 factor expressions together, giving you 1 + 1, which is evaluated to 2.

Factor Expression

The factor_expression adds multiplication and division.

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

This give you 1 * 2 which evaluated to 2. The benefit of having factor_expression below addative_expression in the grammar is it will evaluate factor_expression before addative_expressoins which will give you the correct mathematical output. For 1 + 2 * 2 you will be given 5 which is correct rather than 6 which is incorrect.

Unary Expression

The unary_expression

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

allows you to have negation. This expression is what implements not true which would evaluate to false or -1.

Primary Expression

The primary_expression allows for branching in our language.

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

This is what implements our ability to use many different variables, calls, or expressions in our code depending on the use case.

List Literal Expression

The list_literal allows us to have lists of many different types.

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

This lets us store more than one value under the same variable name which can be used later.

Argument List Expression

Our argument_list is used in the function_call allowing the user to implement more than one argument for the same function.

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

Type Expression

Finally at the bottom level of our grammar we have the type_expression

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

This gives us our types in the grammar. If we wanted to have a variable of type INTEGER we would denote var int x = 1;. This would create a variable of explicit type INTEGER.

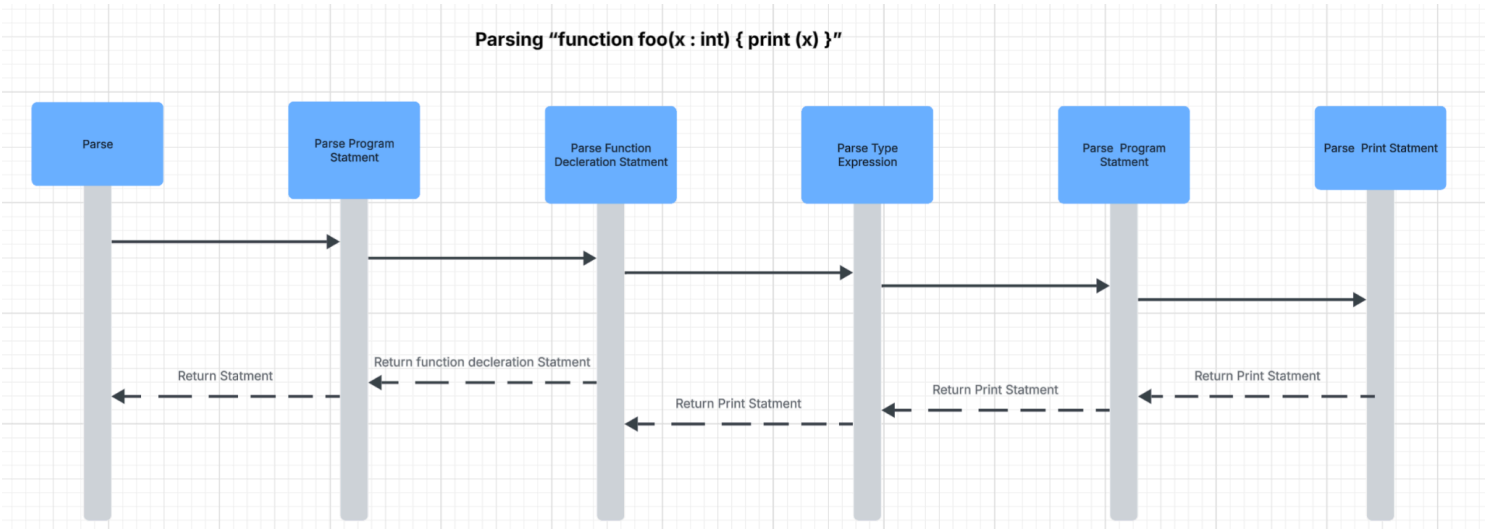
Full Grammar

```
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:

UML Sequence Diagram for parsing “function foo(x : int) { print(x) }”



Section 6 - Design Trade-Offs:

Recursive descent vs. Parser generator

For my compiler design I decided to go with a Recursive descent approach rather than a Parser generator; which is another common approach. In the end both Recursive descent and parser generators are able to create a compiler but the methodology behind both of them is quite different.

Recursive descent is a design choice when building a compiler that uses recursion on the grammar to allow a programmer to build everything from the tokenizer to the final compiler. This allows us to create in almost any capable programming language building up the recursive tree as we go. The benefits of creating a compiler in Recursive descent are mainly that the programmer through building is able to create a deep understanding of how their program is able to function and act in the real world. It also allows us to gain an understanding on how the functions we write work ‘under the hood’. This is because with recursive descent each logical step in the compiler is created by hand requiring an understanding of these concepts.

Contrasted to this is a parser generator. A parser generator is able to take a language specification called a lexer and uses that to generate a parser. This type of program is highly complex but able to generate all types of languages. The upside to this is once a programmer understands how to create these lexers and use a specific parser generator generating whole new languages is much faster than recursive descent. The downside is that because you're just creating a lexer you lose some of the deeper sense on how your language is actually built and how it will respond to certain situations. I believe that using recursive descent is the better option out of the two because it allows the programmer to create a deeper

understanding of their language and it demystifies how programming languages altogether are created.

Section 7 -Software development life cycle model:

For this project I mainly focused on test-driven development. This means that there was a suite of tests I knew my project needed to pass in different stages of development. From my experience there's a range of upsides and downsides to this style of development.

The upsides of test-driven development allowed me to create a mental model for what my code needed to do at different stages. This allowed me to break large pieces of code into manageable problems that I was able to solve one at a time. It also gave instant satisfaction through the stages of development when certain blocks of code had a direct impact on the tests I was able to pass.

For me the downsides of test-driven development are when I find myself writing code to explicitly complete the tests rather than creating functional code. For example when I was writing the compiler logic for the Variable_Statment I found I was just looking at the tests themselves and writing code to complete each one rather than developing logic that made sense for the Variable_Statment as a whole. For me the best way to combat this would be to be very methodical with how the tests are designed. This way when the tests are run in totality they are able to check the main function of the code as well and edge cases ensuring that even if im only developing to pass tests Im still building a robust, reliable system.