Catscript Compiler Portfolio

CSCI 468

Spring 2024

Bryce Lehnen

Section 1: Program

The source code for this course can be found at https://github.com/BryceLehnen/csci-468-spring2024-private/tree/main. The project was to implement a working compiler including pieces like a tokenizer, parser, and bytecode. Zipped source code can also be found in the capstone directory.

Section 2: Teamwork

My teammate provided me with tests and the documentation. I was fully responsible for the implementation of the Catscript Compiler.

Section 3: Design Pattern

The memoization design pattern was used, and it is located inside of CatscriptType.java under the parser directory. This design pattern was used to avoid the wasteful code that was originally there. At first, it would also create a new instance of a ListType even if the exact same one was already created elsewhere. Now, through memoization, it looks up the type in a HashMap aptly named cache to first see if a ListType has already been created. If it has, then it simply returns it. If not, then it creates a new instance, stores it in the cache, and returns the proper ListType. This creates a non-wasteful way to store the various ListTypes.

Section 4: Technical Writing

The technical writing can be found under the name Catscript.md, and it is located in the directory Capstone. It may also simply just be accompanying this document as a separate file. It is also added below in the original markdown format:

````
# Catscript Guide

This is the documentation regarding the technical aspects of the Catscript
Language.
It contains information about the expressions and statements that make of
Catscript
as well explanations and examples of each.

## Introduction

Catscript is a simple scripting language.  Here is an example:

```
````
````

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

Statements

For_Statement
The technical outline for a for_statement is:
```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
                '{', { statement }, '}';
```
A for_statement is a for loop that iterators over an expression, and evaluating various statements
inside of it. There three main parts to a for_statement: the Identifier, the expression, and the body.
The Identifier is a variable that will change as the for_statement is being iterated over. The expression
will denote the 'boundary' of the Identifier. Lastly, the body can be comprised of any number of statements
even zero. A more realistic example is shown below:
```
for (x in [1, 2, 3]) { print(x) }
```
The for_statement will iterate over the list_literal [1, 2, 3], and run the body which is the print_statement
in this example. The output of this for_statement would be:
```
1
2
3
```

If_Statement
The technical outline for an if_statement is:
```
if_statement = 'if', '(', expression, ')', '{',
                    { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

An if_statement allows for branching paths and logic to be added into Catscript. The most basic if_statement
comprises an expression and a body. If the expression is evaluated to be true then the statement body is also
evaluated. However, an if_statement can also contain additional parts to produce branching paths. It can have
either or both additional else_if_statements and one else_statement. A more realistic example is shown below:

```
if (false) { print(1) } else if (false) { print(2) } else { print(3) }
```

The expressions in the parenthesis are just false, but these are just basic examples. It's easy to see how
the branching paths would work as well. If the first expression were true, then a 1 would be printed. If the
second expression were true, then a 2 would be printed. And if neither are true, then a 3 would be printed.

Print_Statement

The technical outline for a print_statement is:

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

A print_statement is exactly what the name implies. It prints the expression that is inside its parenthesis.
That's really all there is to it. The print_statement is extremely useful for tests and debugging since it
can clearly show an output, but it isn't complicated compared to many of the other statements in Catscript.
A more realistic example is shown below:

```
print("Catscript is awesome!")
```

Variable_Statement

The technical outline for a variable_statement is:

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

The variable_statement is exactly what the name implies. It is used to create variables that can be used as
references to whatever value it 'holds'. There are two basic types of variable_statements. The first uses
logic to give its best guess as to what the type of the variable is, like the one at the start of this document.
The other explicitly denotes the type of the variable as shown below. Variables are extremely useful as
the value they hold can be changed or modified while still calling it the same reference/Identifier. A more
realistic example is shown below:

```
var x : String = "Catscript"
```

The only other part of variable_statements is the fact that no two variables (in the same scope) can be named
with the same Identifier. Obviously, this would cause confusion, but it would also break the entire program.
Safety measures are put in place to give warnings about incorrect naming of variables.

Assignment_Statement

The technical outline for an assignment_statement is:

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

The assignment_statement is very similar to the variable_statement that was described above. However, it is
not used to create new variables, but rather, it assigns a new value to a variable that already exists. A more
realistic example is shown below:

```
x = "Catscript"
```

The import part of the assignment_statement is that the value being assigned to the variable must be of
a compatible type. This means that if when x was created, it must be of type String or the above
assignment_statement would throw an error.

Function_Declaration
The technical outline for a function_declaration statement is:
```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                       [ ':' + type_expression ], '{',  {
function_body_statement },  '}';
```
The function_declaration statement is the most complicated type of statement. There are a lot of moving
parts to it, so more and more complicated examples will be given starting with the one below:
```
function fun() { print(1) }
```
The above statement denotes a function with the name 'fun'. All it does is print out the integer '1'. This is the
most basic type of function as it does not contain any parameters in the parameter_list or returns in the
function_body_statement.
```
function fun(a) { print(a) }
```
The above statement denotes a function with the name 'fun'. It has a single parameter with the name 'a', and
the function_body_statement comprises a single statement which will print out the value of 'a'. The type of
'a' is not explicitly stated, so it could be anything from an Integer to String or anything in-between.
```
function fun(a : string) { print(a) }
```
The above statement denotes a function with the name 'fun'. It has a single parameter with the name 'a' and it
is of the type 'string'. The function_body_statement comprises a single statement which will print out the
value of 'a'.
```
function fun : int (a : int) {
    var y : int = (a + 5)
    return y
```

```
}
```
The above statement denotes a function with the name 'fun'. It has a single parameter with the name 'a' and
it is of the type 'int'. The function_body_statement comprises two statements. The first of which is a
variable_statement which takes the value of 'a', adds 5 to it, and stores it in a variable called 'y'. The second
is a return_statement which returns the value of 'y'.

Return_Statement
The technical outline for a return_statement is:
```
return_statement = 'return' [, expression];
```
The return_statement does exactly as the name implies. It either returns back out of a function, or returns a
value. The first is just a simple 'return' without anything following it. It is used to break out of a function,
and it can be used in various ways. The most common would be to simply stop the rest of a function from
running. The second type of return_statement is shown below:
```
return x
```
This type of return_statement returns the value or referenced value that follows the 'return'. In this case, it
will return the value of 'x'. This is best used in a function that calculates some value where the function
is used multiple times throughout a program. The value it returns can then easily be used elsewhere in the
program or stored using either a variable_statement or an assignment_statement.

Expressions

Expression

Expression is used as a start to all the various expressions (except type_expression). It is essentially
a placeholder that is used to describe statements since the statements themselves do not know what
type of expression will be placed inside of them. The first step in evaluating an expression is
the equality_expression which is described next.

Equality_Expression
The technical outline for an equality_expression is:
```
equality_expression = comparison_expression { ("!=" | "==")
comparision_expression };
```
A basic equality_expression is quite simple. It can either evaluate down to just a comparison_expression
which is the next step, or it will evaluate to basic expression that is seen in so many languages.
A more realistic example is shown below:
```
x != 10
```

Comparison_Expression
The technical outline for a comparison_expression is:
```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" )
additive_expression };
```
A basic comparison_expression does just as the name implies. It compares two additive_expressions. A
more realistic example is shown below:
```
x >= 10
```

Additive_Expression
The technical outline for an additive_expression is:
```
```

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression
};
```
A basic additive_expression does just as the name implies. It adds (or
subtracts) two factor_expressions. A
more realistic example is shown below:
```
x + 10
```

Factor_Expression
The technical outline for a factor_expression is:
```
factor_expression = unary_expression { ("/" | "*" ) unary_expression };
```
A basic factor_expression covers two operations. Those two being
multiplication and division of two
unary_expressions. A more realistic example is shown below:
```
x * 10
```

Unary_Expression
The technical outline for a unary_expression is:
```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```
A basic unary_expression covers both the negative sign for things like
integers and the "not" for things
like booleans. A more realistic example is shown below:
```
not true
```

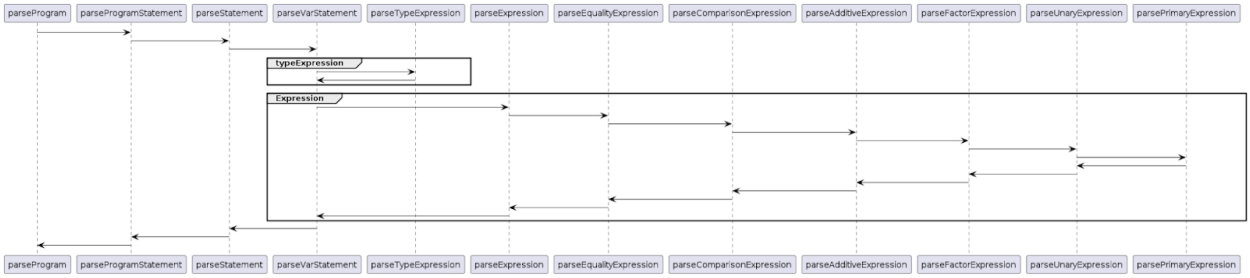
Primary_Expression
The technical outline for a primary_expression is:
```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" |
"null"|
                     list_literal | function_call | "(", expression, ")"
```

```
A basic primary_expression covers a whole host of things. These are the
most basic expressions. From
basic types like Strings and Integers to booleans and null values. Primary
expression also can evaluate
to list_literals which are, as the name implies, a list of expressions. A
more realistic example is shown
below:
```
"Catscript is awesome!"
```

### Type_Expression
The technical outline for a type_expression is
```
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' ,
type_expression, '>']
```
A basic type_expression simple evaluates to one of the 5 types listed
above. A null type is also
technically a part of Catscript, but it is of type 'object'. The
type_expression is only used in
a few statements for explicitly declaring what the type of the statement
is.

## Section 5: UML

The UML Diagram was created using PlantText Editor. It denotes the sequence diagram for the following code snippet: 'var x : int = 10'


```

Section 6: Design Trade-Offs

The main design trade-off was the use of recursive descent rather than a parser generator. Recursive descent parsers are typically hand-coded, offering greater control and visibility into the parsing process. This approach can be advantageous for simpler grammars or when performance is critical, as handcrafted parsers often yield faster execution speeds. However, recursive descent parsers can become unwieldy for complex grammars, leading to maintenance challenges and potential inefficiencies due to manual implementation. On the other hand, parser generators automate much of the parsing process based on a formal grammar specification, reducing development time and potential human errors. They excel in handling complex grammars and offer modularity, facilitating easier maintenance and extension. However, parser generators may introduce additional overhead, both in terms of learning curve and runtime performance, and can sometimes produce less readable or efficient code compared to handcrafted parsers. Thus, the choice between recursive descent and parser generators often hinges on factors such as the complexity of the grammar, performance requirements, development time constraints, and the desired level of control and visibility in the parsing process.

Section 7: Software Development Life Cycle Model

Employing Test-Driven Development (TDD) as a model for project development has proven invaluable in various aspects. By following the TDD approach, developers first write tests that define the desired behavior of the system. This clarifies requirements upfront and guides the development process towards meeting those specifications. Moreover, these tests serve as a safety net, enabling developers to refactor or modify code confidently, knowing that they can quickly identify regressions if any occur. TDD also encourages modular and loosely coupled code, as developers focus on writing code that can be easily tested in isolation. Furthermore, TDD promotes a more iterative and incremental development process, where small units of functionality are implemented and tested in rapid cycles, leading to early bug detection and quicker feedback loops. Overall, TDD was the best option for development as it gave me quick feedback on the code that I was writing. I would not have enjoyed this project nearly as much if TDD was not used.