SIMPLE INTERPRETER IN JS

David Melisso
AT in CS: Compilers & Interpreters
May 31, 2018

INTRODUCTION

In the class *Advanced Topics in Computer Science: Compilers and Interpreters*, I have learned how to make a simple interpreter and then compiler for a language similar to Pascal. For our final project, we were assigned with the task of creating an interpreter for a language we called SIMPLE (not the real SIMPLE language). We were supposed to use a recursive descent parser with one character look ahead. Luckily, we had already made this sort of parser for the course. An easy way to finish our final project would be to modify our original parser to match the given grammar.

That is not what I decided to do. I decided to completely re-write the parser in TypeScript, a language that compiles into JavaScript; the original parser was written in Java. Aside from enjoying coding in JavaScript, I decided to code in TypeScript because it had strict typing and would be a fun challenge. This document documents the design for this interpreter.

DESIGN

GRAMMAR

Original Grammar

Below is the grammar that we were given in the assignment document.

```
Program         => Statement P
P               => Program
                 | ε
Statement       => display Expression St1
                 | assign id = Expression
                 | while Expression do Program end
                 | if Expression then Program St2
St1             => read id
                 | ε
St2             => end
                 | else Program end
Expression      => Expression relop AddExpr
                 | AddExpr
AddExpr         => AddExpr (+ | -) MultExpr
                 | MultExpr
MultExpr        => MultExpr (* | /) NegExpr
                 | NegExpr
NegExpr         => - Value
                 | Value
Value           => id
                 | number
                 | ( Expression )
```

Modified Grammar

I modified the grammar so that it could be correctly parsed by the interpreter.

```
Program         => Statement P
P               => Program
```

```
                    | ε
Statement          => display Expression St1
                    | assign id = Expression
                    | while Expression do Program end
                    | if Expression then Program St2
St1                => read id
                    | ε
St2                => end
                    | else Program end
Expression         => AddExpr WExpression
WExpression        => relop AddExpr WExpression
                    | ε
AddExpr            => AddExpr WAddExpr
WAddExpr           => (+ | -) MultExpr WAddExpr
                    | ε
MultExpr           => MultExpr WNegExpr
WNegExpr           => (* | /) NegExpr WNegExpr
                    | ε
NegExpr            => - Value
                    | Value
Value              => id
                    | number
                    | ( Expression )
```

Explanation

The original grammar cannot be parsed by our parser because it would cause infinite recursion in multiple places. The first term in `Expression`, `AddExpr`, and `MultExpr` is itself; in execution, this would mean that the first statement in each parsing method would be a recursive call. For example, in `Expression`, the parser has no way to tell the difference between `Expression` and `AddExpr`, and so the first statement would be `parseExpression()`. To fix this problem, I made a `WExpression` (W standing for While) non-terminal, which handles the recursion. Because of the way `Expression` is defined, it must start with an `AddExpr`. Therefore, the first thing to parse would be an `AddExpr`. Then, we would parse a `WExpression`. This separates the parsing decision into two options: one starting with **`relop`**, and the other being an empty string. This way, the parser knows which option it should parse: the decision depends on whether the current token is a

relop. If a **relop** is found, the parser will `AddExpr`, and to handle the recursion, another `WExpression` is added. Each grammar is the same; it will handle a whole number of `AddExpr`s separated by **relop**s. This is the same with `AddExpr`, and `MultExpr`.

DECOMPOSITION

Methods

The `parseProgram` method will return a `Program` object, which contains an array of `Statement`s. The method will include the parsing of the non-terminal `P`.

The `parseStatement` method will return a `Display`, `Assignment`, `While`, or `If`, all of which are children of the abstract class `Statement`. It will return an object based off of the current token at the beginning of the method call. The method will include the parsing of the non-terminals `St1` and `St2`.

The `parseExpression`, `parseAddExpr`, `parseMultExpr`, and `parseMultExpr` methods will return any child of the abstract class `Expression`. The method will include the parsing of the non-terminal `WExpression`, `WAddExpr`, and `WMultExpr`. Each will parse the first non-terminal, and then if it reaches the correct terminal, it will return a `BinOp`. Otherwise, it will return an `Expression` of the same type as the non-terminal that it parsed.

The `parseNegExpr` method will return any child of the abstract class `Expression`. If the current token at the beginning of the method call is a horizontal dash, then it will return a `BinOp`; otherwise it will return an `Expression` of the same type as was parsed in the method `parseValue`.

The `parseValue` method will return a `Variable`, `Number`, or `Expression`. If the current token began with a letter, the parser returned a `Variable`. If the current token began with a number, then it returned a `Number`. If it began with an opening parenthesis, it would return the `Expression` located between the parentheses.

Objects

- `Program`: the `Program` class contains an array of `Statement`s, and when executed, executes each `Statement` in that array in order
- `Statement`: the `Statement` class is an abstract class representing something that can be executed
    - `Display`: the `Display` class extends `Statement` and, when executed, prints the evaluated `Expression` it is given to the console, and optionally reads in user input to a `Variable`

    - `Assignment`: the `Assignment` class extends `Statement` and, when executed, sets a variable in the environment to the evaluated `Expression` it is given
    - `While`: the `While` class extends `Statement` and, when executed, executes a `Program` every time an `Expression` evaluates to true
    - `If`: the If class extends `Statement` and, when executed, executes a `Program` once if an `Expression` evaluates to true, and optionally executes a secondary `Program` if the given `Expression` evaluates to false
- `Expression`: the `Expression` class is an abstract class representing something that can be evaluated and resolved to either a boolean or integer
    - `BinOp`: the `BinOp` class extends `Expression` and, when evaluated, evaluates two sub-`Expression`s and uses an operator to perform an operation between the two, such as addition (+) or equality (=).
    - `Variable`: the `Variable` class extends `Expression` and contains a string, and, when evaluated, returns the value tied to that string in the environment
    - `Number`: the `Number` class extends `Expression` and, when evaluated, returns the number stored in it
    - `Boolean`: the `Boolean` class extends `Expression` and, when evaluated, returns the boolean stored in it

ADDED FEATURES

I added a type of `Expression` called `Boolean`, which evaluated to either true or false. It would be parsed within the `parseValue` method, which would return a `Boolean` if the current token was either “true” or “false”.

TESTING

PLAN

For testing, I used Mrs. Datar's (the teacher's) testing code. I used the first "simpleTest"s she created, and added my own code. First, it should display 3, assign x to 1, and then read to x. This tests that `Display`, `Assignment`, and `Variable` work. Then, while x is less than 10, it should display x and increment x. This tests that `While` works, and that `Assignment` and `Display` work inside of it. Then, it should display x, and then do an If statement. If x is 9, then it should set x to 25; if x is 10, it should set it to 35, and otherwise it should set it to 45. This doubly tests that `While` works as it should not execute if the read in x was greater than or equal to 10, as well as test if `If` works. Finally, it displays x and x+4. These test that `Assignment` works inside an `If`. Finally, it should display -600 and then the evaluation of x*5+4. This will test that the Parser itself works and parses binary operations in the correct order (order of operations, where multiplication should happen first).

So, if the input is equal less than 10, the output should be:

```
3
1
[Special]
10
35
39
-600
199
```

Where [Special] is every integer between the input and 9, inclusive.

If the input is greater than or equal to 10, the output should be:

```
3
1
[Special]
45
49
-600
249
```

Where [Special] is the input.

INPUT

```
display 3
assign x = 1
display x read x
while x < 10 do
   display x
   assign x = x+1
end
if x = 9 then
    display x
    assign x = 25
    display x
else if x = 10 then
    display x
    assign x = 35
    display x
else
    display x
    assign x = 45
    display x
end
end
display x + 4
end

display -600
display x*5+4
```

RESULTS

Given the input 1, the output was as follows:

```
3
1
1
```

```
2
3
4
5
6
7
8
9
10
35
39
```

Given the input 10, the output was as follows:

```
3
1
10
35
39
```

Given the input 11, the output was as follows:

```
3
1
11
35
39
```

All of these tests match my expectations outline above, and therefore the code works.

CONCLUSION

As this project should not challenge the everyday class member, as they could easily use their old code, adding a small extra challenge made this project enticing to me. Using TypeScript helped me learn this new language quite well. While I had already learned JavaScript, TypeScript added some features that I have found myself to appreciate; most obviously: strict typing. Without the strict typing, some JavaScript programmers may have to manually do type checking, a loathsome process. It is also difficult to read. TypeScript eliminates these difficulties, and makes the process of programming more simple. This project not only demonstrates my understanding of the course but also my love of learning in the field of computer science. I hope that in my future computer science classes, I will continue to be challenged and will continue to learn through those challenges.

APPENDIX: SOURCE CODE

The source code is attached in the upload of this file. It is written in TypeScript.