CScript Language

Building an Interpreter in Java

(Recursive Descent Parser)

Ambiguous Grammar

- expression → literal | unary | binary | grouping ;
- literal → NUMBER | STRING | "false" | "true" | "nil" ;
- grouping → "(" expression ")" ;
- unary → ("-" | "!") expression ;
- binary → expression operator expression ;
- operator → "==" | "!=" | "<" | "<=" | ">" | ">=" | "+" | "-" | "*" | "/" ;

Why do we need unambiguous grammar?

- Example of expression to be parsed :- 6 / 3 - 1
- Precedence determines which operator is evaluated first in an expression containing a mixture of different operators. Precedence rules tell us that we evaluate the / before the - in the above example. Operators with higher precedence are evaluated before operators with lower precedence. Equivalently, higher precedence operators are said to “bind tighter”.
- Associativity determines which operator is evaluated first in a series of the same operator. When an operator is left-associative (think “left-to-right”), operators on the left evaluate before those on the right. Since - is left-associative, this expression:

Precedence and Associativity Table for CScript

Name	Operators	Associates	Precedence Index
Equality	== !=	Left	1
Comparison	> >= < <=	Left	2
Addition	- +	Left	3
Multiplication	/ *	Left	4
Unary	! -	Right	5

Unambiguous Grammar for CScript

- expression → equality ;
- equality → comparison (("!=" | "==") comparison)* ;
- comparison → addition ((">" | ">=" | "<" | "<=") addition)* ;
- addition → multiplication (("-" | "+") multiplication)* ;
- multiplication → unary (("/" | "*") unary)* ;
- unary → ("!" | "-") unary | primary ;
- primary → NUMBER | STRING | "false" | "true" | "nil" | "(" expression ")" ;

The above grammar is in Extended Backus-Naur Form. The symbols in the grammar are as follows:

→ - left hand side may derive any of the right hand side.

* - expression may repeat more that once.

? - expression can be optional

About Recursive Descent Parser!

- It is considered a top-down parser because it starts from the top or outermost grammar rule (here expression) and works its way down into the nested subexpressions before finally reaching the leaves of the syntax tree. This is in contrast with bottom-up parsers like LR that start with primary expressions and compose them into larger and larger chunks of syntax.
- A recursive descent parser is a literal translation of the grammar's rules straight into imperative code. Each rule becomes a function. The body of the rule translates to code roughly like:

Grammar notation	Code representation
Terminal	Code to match and consume a token
Nonterminal	Call to that rule's function
|	if or switch statement
* or +	while or for loop
?	if statement

- It's called "recursive descent" because when a grammar rule refers to itself—directly or indirectly—that translates to recursive method calls.

Requirement for handling syntax error in a Parser

- There are a couple of hard requirements for when the parser runs into a syntax error:
- It must detect and report the error. If it doesn't detect the error and passes the resulting malformed syntax tree on to the interpreter, all manner of horrors may be summoned.
- It must not crash or hang. Syntax errors are a fact of life and language tools have to be robust in the face of them. Segfaulting or getting stuck in an infinite loop isn't allowed. While the source may not be valid code, it's still a valid input to the parser because users use the parser to learn what syntax is allowed.
- Be fast. Computers are thousands of times faster than they were when parser technology was first invented. The days of needing to optimize your parser so that it could get through an entire source file during a coffee break are over. But programmer expectations have risen as quickly, if not faster. They expect their editors to reparse files in milliseconds after every keystroke.
- Report as many distinct errors as there are. Aborting after the first error is easy to implement, but it's annoying for users if every time they fix what they think is the one error in a file, a new one appears. They want to see them all.
- Minimize cascaded errors. Once a single error is found, the parser no longer really knows what's going on. It tries to get itself back on track and keep going, but if it gets confused, it may report a slew of ghost errors that don't indicate other real problems in the code. When the first error is fixed, they disappear, because they merely represent the parser's own confusion. These are annoying because they can scare the user into thinking their code is in a worse state than it is.

Panic mode error recovery

- Of all the recovery techniques devised in yesteryear, the one that best stood the test of time is called—somewhat alarmingly—"panic mode". As soon as the parser detects an error, it enters panic mode. It knows at least one token doesn't make sense given its current state in the middle of some stack of grammar productions.
- Before it can get back to parsing, it needs to get its state and the sequence of forthcoming tokens aligned such that the next token does match the rule being parsed. This process is called synchronization.
- To do that, we select some rule in the grammar that will mark the synchronization point. The parser fixes its parsing state by jumping out of any nested productions until it gets back to that rule. Then it synchronizes the token stream by discarding tokens until it reaches one that can appear at that point in the rule.
- Any additional real syntax errors hiding in those discarded tokens aren't reported, but it also means that any mistaken cascaded errors that are side effects of the initial error aren't falsely reported either, which is a decent trade-off.
- The traditional place in the grammar to synchronize is between statements.

Grammar for Statements

- program → statement* EOF ;
- statement → expressionStatement | print ;
- expressionStatement → expression ";" ;
- printStmt → "print" expression ";" ;

Grammar for var declaration

- program → declaration* EOF ;
- declaration → varDecl | statement ;
- statement → exprStmt | printStmt ;
- varDecl → "var" IDENTIFIER ("=" expression)? ";" ;
- ? means optional
- To access a variable, we define a new kind of primary expression:
- primary → "true" | "false" | "null" | NUMBER | STRING | "(" expression ")" | IDENTIFIER ;

Assignment Grammar Syntax

- expression → assignment ;
- assignment → IDENTIFIER "=" assignment | equality ;

Block Syntax and Grammar

- statement → exprStmt | printStmt | block ;
- block → "{" declaration* "}" ;

Conditional Execution Type

- We can divide control flow roughly into two kinds:
- Conditional or branching control flow is used to not execute some piece of code. Imperatively, you can think of it as jumping ahead over a region of code.
- Looping control flow executes a chunk of code more than once. It jumps back so that you can do something again. Since you don’t usually want infinite loops, it typically has some conditional logic to know when to stop looping as well.

Conditional Grammar

- statement → exprStmt | ifStmt | printStmt | block ;
- ifStmt → "if" "(" expression ")" statement ("else" statement)? ;

'Dangling else' problem

- The seemingly innocuous optional else has in fact opened up an ambiguity in our grammar. Consider:
- if (first) if (second) whenTrue(); else whenFalse();
- Here's the riddle: Which if statement does that else clause belong to? This isn't just a theoretical question about how we notate our grammar. It actually affects how the code executes.
- If we attach the else to the first if statement, then whenFalse() is called if first is falsey, regardless of what value second has.
- If we attach it to the second if statement, then whenFalse() is only called if first is truthy and second is falsey.
- Since else clauses are optional, and there is no explicit delimiter marking the end of the if statement, the grammar is ambiguous when you nest ifs in this way. This classic pitfall of syntax is called the "dangling else" problem.
- It is possible to define a context-free grammar that avoids the ambiguity directly, but it requires splitting most of the statement rules into pairs, one that allows an if with an else and one that doesn't. It's annoying.
- Instead, most languages and parsers avoid the problem in an ad hoc way. No matter what hack they use to get themselves out of the trouble, they always choose the same interpretation—the else is bound to the nearest if that precedes it.
- Our parser conveniently does that already. Since ifStatement() eagerly looks for an else before returning, the innermost call to a nested series will claim the else clause for itself before returning to the outer if statements.

Logical Operators

- Since we don't have the conditional operator, you might think we're done with branching, but no. Even without the ternary operator, there are two other operators that are technically control flow constructs—the logical operators and and or.
- These aren't like other binary operators because they short-circuit. If, after evaluating the left operand, we know what the result of the logical expression must be, we don't evaluate the right operand. For example: false and sideEffect();
- For an and expression to evaluate to something truthy, both operands must be truthy. We can see as soon as we evaluate the left false operand that that isn't going to be the case, so there's no need to evaluate sideEffect() and so it gets skipped.
- This is why we didn't implement the logical operators with the other binary operators. Now we're ready. The two new operators are low in the precedence table. Similar to || and && in C, they each have their own precedence with or lower than and. We slot them right between assignment and equality:

Logical Operators Grammar

- expression → assignment ;
- assignment → identifier "=" assignment | logic_or ;
- logic_or → logic_and ("or" logic_and)* ;
- logic_and → equality ("and" equality)* ;

Loop Grammar

WHILE LOOP:

- statement → exprStmt | ifStmt | printStmt | whileStmt | block ;
- whileStmt → "while" "(" expression ")" statement

FOR LOOP:

- statement → exprStmt | forStmt | ifStmt | printStmt | whileStmt | block ;
- forStmt → "for" "(" (varDecl | exprStmt | ";") expression? ";" expression? ")" statement ;

Concept of Desugaring using FOR LOOP

- Example of for loop:
- for (var i = 0; i < 10; i = i + 1) print i;
- That's a lot of machinery, but note that none of it does anything you couldn't do with the statements we already have. If for loops didn't support initializer clauses, you could just put the initializer expression before the for statement. If it didn't have an increment clause, you could simply put the increment expression at the end of the body yourself.
- In other words, CScript doesn't need for loops, they just make some common code patterns more pleasant to write. These kinds of features are called syntactic sugar. For example, the previous for loop could be rewritten to:
- {
- var i = 0;
- while (i < 10) {
- print i;
- i = i + 1; } }

Concept of Desugaring using FOR LOOP

- That has the exact same semantics, though it's certainly not as easy on the eyes. Reducing syntactic sugar to semantically equivalent but more verbose forms is called desugaring. We'll use this technique inside our interpreter. Instead of directly interpreting for loops, the parser will consume the new syntax and translate it to more primitive forms that the interpreter already knows how to execute.
- Oh, how I wish the accepted term for this was "caramelization". Why introduce a metaphor if you aren't going to stick with it?
- It's not saving us a ton of work in this tiny interpreter, but desugaring is a powerful tool to have in your toolbox and this gives me an excuse to show it to you. In a sophisticated implementation, the backend does lots of work optimizing each supported chunk of semantics. The fewer of those there are, the more mileage it gets out of each optimization.
- Meanwhile, a rich syntax makes the language more pleasant and productive to work in. Desugaring bridges that gap—it lets you take a robust expressive grammar and cook it down to a small number of primitives that the back end can do its magic on.

Funtion

- unary → ("!" | "-") unary | call ;
- call → primary ("(" arguments? ")")* ;

Checking Arity

- The other problem relates to the function's arity. Arity is the fancy term for the number of arguments a function or operation expects. Unary operators have arity one, binary operators two, etc. With functions, the arity is determined by the number of parameters it declares:
- fun add(a, b, c) {
- print a + b + c;
- }
- This function defines three parameters, a, b, and c, so its arity is three and it expects three arguments. So what if you try to call it like this:
- add(1, 2, 3, 4); // Too many.
- add(1, 2); // Too few.
- In CScript it throws runtime error if arity doesn't match.

Function Declaration

- declaration → funDecl | varDecl | statement ;
- funDecl → "fun" function ;
- function → IDENTIFIER "(" parameters? ")" block ;
- parameters → IDENTIFIER ("," IDENTIFIER)* ;

Return Statement Grammar

- statement → exprStmt | forStmt | ifStmt | printStmt | returnStmt | whileStmt | block ;

- returnStmt → "return" expression? ";" ;

Implementing Return statement from calls or execution in Java

- Interpreting a return statement is tricky. You can return from anywhere within the body of a function, even deeply nested inside other statements. When the return is executed, the interpreter needs to jump all the way out of whatever context it's currently in and cause the function call to complete, like some kind of jacked up control flow construct.
- For example, say we're running this program and we're about to execute the return statement:

```
fun count(n) {
  while (n < 100) {
    if (n == 3) return n; // <--
    print n;
    n = n + 1;
  }
}
count(1);
```

Implementing Return statement from calls or execution in Java

- The Java call stack currently looks roughly like this:
- Interpreter.visitReturnStmt()
- Interpreter.visitIfStmt()
- Interpreter.executeBlock()
- Interpreter.visitBlockStmt()
- Interpreter.visitWhileStmt()
- Interpreter.executeBlock()
- LoxFunction.call()
- Interpreter.visitCallExpr()
- We need to get from the top of the stack all the way back to call(). I don't know about you, but to me that sounds like exceptions. When we execute a return statement, we'll use an exception to unwind the interpreter past the visit methods of all of the containing statements back to the code that began executing the body.
- The visit method for our new AST node looks like this:

Classes Grammar

- declaration → classDecl | funDecl | varDecl | statement ;
- classDecl → "class" IDENTIFIER "{" function* "}" ;
- function → IDENTIFIER "(" parameters? ")" block ;
- parameters → IDENTIFIER ("," IDENTIFIER)* ;

Instance Properties Grammar & Get/Set instance properties

- call → primary ("(" arguments? ")" | "." IDENTIFIER)* ;
- assignment → (call ".")? IDENTIFIER "=" assignment | logic_or ;

Superclass and Subclass Grammar

- classDecl → "class" IDENTIFIER ("<" IDENTIFIER)? "{" function* "}" ;
- primary → "true" | "false" | "null" | "this"
- | NUMBER | STRING | IDENTIFIER | "(" expression ")"
- | "super" "." IDENTIFIER ;