EC

An easy programming C language

C Programming Final Project

Suzan T S Awinat

Laura Cuellar, Zaid Alsaheb, Sebastian Perilla, Sofia Vitorica

**INTRODUCTION**

EC is a programming language that transforms the coding experience into a playful and interactive conversation. Getting started with programming can often feel a bit overwhelming. Our programming language aims to provide more natural expressions instead of the usual keywords in C to make Programming more fun and easier to engage with. Our goal is that the user feels as if they are having an informal conversation with someone in this case, a computer

Our aim is to introduce the world of programming to a special group of individuals: little kids as we know that young minds can be brilliant too and this would be a way of initiating an early interest in programming and technology with an entertaining tool. It will also be aimed at very beginner coders that might feel intimidated to learn how to program, this way they would grasp coding principles easily.

**FEATURES**

Our programming language has many features that help deliver the learning experience as smoothly as possible, such as:

* Dynamic typing: Variables are defined at first using the keyword ‘store’ and the datatype is managed appropriately without intervention.
* Garbage collection and memory management: No need to worry about managing memory as reallocating and resizing is handled automatically.
* Scope Handling: Inside every block of curly brackets the scope changes and is handled appropriately
* Functions and closures: users can define their own functions and pass in any number of arguments,

**OUR IMPLEMENTATION**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*SCANNER/LEXER*

The scanner is a crucial component in the EC programming language, responsible for lexical analysis or tokenization of the source code. The scanner reads the source code character by character and generates a sequence of tokens that represent the fundamental building blocks of the EC language.

Key Uses and Responsabilities:

**Initialization**: The *initScanner* function initializes the scanner by setting the start, current, and line attributes of the Scanner structure.

**Character Classification**: The file contains helper functions (*isAlpha, isDigit, isAtEnd*) for classifying characters, helping in identifying alphabetic characters, and digits, and checking if the end of the source code is reached.

**Advancement and Peeking**: Functions like *advance*, *peek*, and *peekNext* facilitate the movement of the current character pointer and provide a look-ahead capability without consuming characters.

**Matching Characters**: The match function checks if the current character matches an expected character and advances the pointer if there is a match.

**Token Creation**: Several functions (*createToken, makeToken, errorToken*) are responsible for creating different types of tokens based on the recognized lexemes.

**Whitespace and Comment Handling**: The *skipWhitespace* function skips over spaces, tabs, and comments, ensuring they do not contribute to the tokenization process.

**Keyword Recognition**: The *checkKeyword* and *identifierType* functions handle the recognition of keywords in the EC language, determining the appropriate token type for identifiers.

**Lexeme Handling**: The functions identifier, number, and string are responsible for handling lexemes such as identifiers, numeric literals, and string literals.

Main Tokenization Function: The scanToken function is the main entry point for tokenization. It orchestrates the entire process, recognizing different types of tokens, including single-character tokens, keywords, identifiers, and literals, and handling errors for unexpected characters.

Integration with Other Components:

The output of the scanner, a sequence of tokens, serves as the input for the subsequent stages of the EC language implementation, including the parser, compiler, and interpreter.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Chunk (ByteCode)*

The chunk.c and chunk.h files are part of the EC programming language implementation. They play a crucial role in handling the bytecode generation and management of the EC virtual machine.

*initChunk*: This function initializes a new Chunk structure by setting its count, and capacity, and initializing arrays for bytecode instructions (code), source code line numbers (lines), and constant values (constants).

*freeChunk*: This function deallocates memory resources used by a Chunk. It frees the memory allocated for bytecode instructions, source code line numbers, and constant values.

*writeChunk*: This function writes a bytecode instruction to the Chunk. It dynamically expands the Chunk if necessary to accommodate the new instruction, updating the bytecode, line numbers, and count.

*addConstant*: This function adds a new constant value to the Chunk's constant array. It also maintains the stack for constant values during the process.

The chunk.h file provides declarations for these functions and defines an OpCode enumeration representing different bytecode instructions used in the EC language.

These components are vital for managing the intermediate representation (bytecode) of EC programs, facilitating execution by the virtual machine. The Chunk structure is a central data structure for storing bytecode instructions, source code line information, and constant values.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Compiler/Interpreter*

The compiler.c file you provided appears to be an implementation of a compiler for the "ec lang" programming language. Here is a high-level overview of the structure and functionality of the compiler:

**Includes and Dependencies:**

The file includes necessary header files such as "compiler.h", "memory.h", and "scanner.h".

It also includes standard C libraries like <stdio.h>, <stdlib.h>, and <string.h>.

Data Structures:

The *Parser* struct keeps track of parsing information, including the current and previous tokens, error flags, and panic mode.

The *ParseRule* struct defines rules for parsing expressions, including prefix and infix functions along with precedence levels.

The *Local* struct represents a local variable, including its name, depth, and whether it is captured by an upvalue.

The *Upvalue* struct represents an upvalue for closures.

The *FunctionType* enum distinguishes between regular functions, initializers, and scripts.

The *Compiler* struct holds information about the current compilation state, including the enclosing compiler, the current function being compiled, local variables, upvalues, etc.

Global Variables:

parser: A global instance of the Parser struct.

current: A global pointer to the current Compiler instance.

Functions:

**Lexer and Error Handling**:

*errorAt, error, errorAtCurrent*: Functions for reporting errors at specific tokens.

*advance*: Function to advance the parser to the next token.

*consume*: Function to consume the current token if it matches a given type; otherwise, report an error.

*check, match*: Functions for checking and matching the current token type.

**Bytecode Emission**:

Various functions (*emitByte, emitBytes, emitLoop, emitJump, etc.*) for emitting bytecode instructions into the chunk of the current function.

**Constant Handling**:

*makeConstant, emitConstant*: Functions for creating and emitting constant values.

**Compiler Initialization and Finalization**:

*initCompiler*: Function to initialize a new compiler instance.

*endCompiler:* Function to finalize the compilation and return the resulting function.

**Scope Handling**:

*beginScope, endScope*: Functions for managing lexical scopes.

**Variable Handling**:

*declareVariable, defineVariable*: Functions for handling variable declarations and definitions.

*resolveLocal, addUpvalue, resolveUpvalue*: Functions for resolving and managing local variables and upvalues.

**Expression Parsing**:

*expression, parsePrecedence*: Functions for parsing expressions with different precedence levels.

**Statement Parsing**:

*block, declaration, statement*: Functions for parsing blocks, declarations, and statements.

**Control Flow Statements**:

*ifStatement, whileStatement, forStatement*: Functions for parsing if, while, and for statements.

**Function Parsing**:

*function, funDeclaration:* Functions for parsing function definitions.

**Other Statements**:

*printStatement, returnStatement*: Functions for parsing print and return statements.

**Unary and Binary Expressions**:

*unary, binary*: Functions for handling unary and binary expressions.

**Literal Handling**:

*literal, number, string, etc.*: Functions for handling literals like true, false, nil, numbers, and strings.

**Compiler Entry Point**:

*compile*: The main entry point for compiling source code into an ObjFunction.

Garbage Collection Integration:

*markCompilerRoots*: Function to mark roots of the compiler in the garbage collector.

This compiler seems to be implementing a bytecode compilation approach for the "ec lang" language. It includes features like lexical scoping, closures, control flow statements, function declarations, and expressions. The generated bytecode is designed to be executed by a virtual machine.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*VM*

The vm.c file contains the implementation of the virtual machine for the EC programming language. The virtual machine is responsible for executing the bytecode generated by the compiler and interpreting EC code. Here's an overview of the key components and functionalities of the vm.c file:

**Global VM Instance :**

The vm variable represents the global instance of the virtual machine*.*

**Native Functions :**

Native functions such as clockNative are implemented to provide functionalities like getting the current time.

The defineNative function adds native functions to the global scope.

**Stack and Frame Management :**

resetStack initializes the VM stack and frame count.

push and pop are used for managing the value stack.

**Error Handling :**

runtimeError reports a runtime error with formatted output and prints information about the call stack.

**Initialization and Finalization :**

initVM initializes the VM, including the stack and other necessary structures.

freeVM frees memory used by the VM.

**Execution Loop :**

The run function is the main execution loop responsible for interpreting bytecode instructions.

**Value Stack Operations :**

call and callValue handle function calls.

closeUpvalues closes upvalues until a certain point in the stack.

captureUpvalue captures an upvalue from the local stack.

**Bytecode Execution :**

interpret function compiles the source code into bytecode and then executes it using the virtual machine.

**Other Functions :**

defineNativeFunctions initializes and defines native functions in the global scope.

Opcode Implementation

The run function includes the implementation for various opcodes such as OP\_CONSTANT, OP\_ADD, OP\_SUBTRACT, etc. It interprets these opcodes and executes the corresponding operations.

Overall Structure

The file contains a comprehensive implementation of the EC virtual machine, with functions for stack manipulation, error handling, bytecode execution, and more.

This file plays a crucial role in the execution of EC programs, bridging the gap between the compiled bytecode and the desired output or behavior.

**FUNCTION DECLARATION/SYNTAX**

EC features the same basic C arithmetic expressions such as +, -, /, \*, %, ++, –

For comparison operators we kept the ones in C (==, !=, >, <, >=, <=) but added some other options that could be used like “is” “same as” for ==. Logical operators remained the same.

All variables Int, Char, String, Double and Float are declared with the same keyword: “store”. This simplifies the declaration of strings as we do not have to specify any string length that the C structure “char string[10]” has.

**Example: Using the combine and is Keywords**

The combine keyword in EC Lang serves a dual purpose – it not only concatenates strings but also prints the result. Let's explore an example to illustrate the usage of the combine keyword, furthermore we have changed “=” to now work as “is” so that it is simpler to understand as is the intended purpose of EC Lang:

**Input:**

>var a is "domain";

>

>var b is "expansion";

>

>combine(a, b);

**Explanation:**

**Variable Declarations:**

var a is "domain"; Declares a variable a and assigns the string value "domain" to it.

var b is "expansion";: Declares a variable b and assigns the string value "expansion" to it.

**Combine Operation:**

combine(a, b);: The combine keyword is used to concatenate the values of a and b. In this case, it will print the combined string "domainexpansion".

**Output:** domainexpansion

**Summary:**

The combine keyword simplifies the process of concatenating and printing strings in a single operation. It enhances readability and reduces the need for explicit print statements when combining string values. The example demonstrates how to use combine to concatenate the values of a and b, resulting in the output "domainexpansion".

**CONCLUSION:**

Try it out!!

Thank you for the semester Suzan, we will miss you!