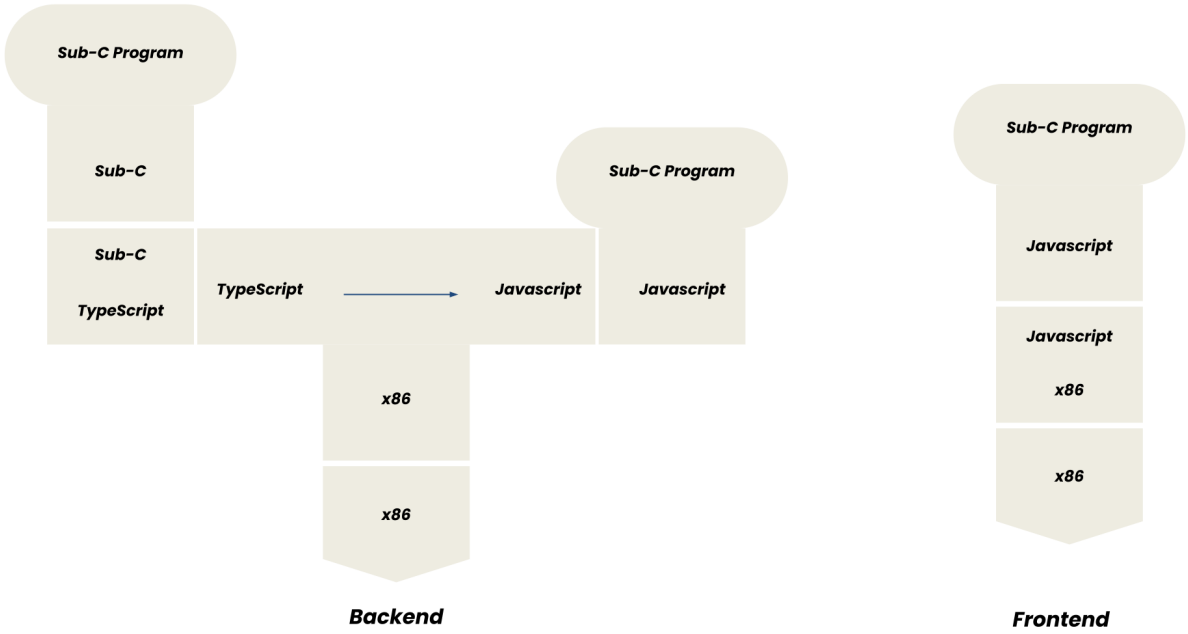
Sub-C Interpreter in Javascript

Pawandeep Singh A0218389N
Wei Sicheng A0268823U
Tutorial Group: 3
Github repo link: https://github.com/Sicheng-Wei/cs4215

Project

In our project, we create an interpreter for a sublanguage of C in Javascript. Our implementation of the interpreter will be via an explicit control evaluator derived from Module 3 of our course, which will allow us to evaluate the control flow of the program. We provide a Web-based implementation based on the Source Academy front-end and derived from the js-slang backend as our starting template.

To achieve our project, we have created a T-diagram of the major language processing steps that our project utilises. The T-diagram is shown below:

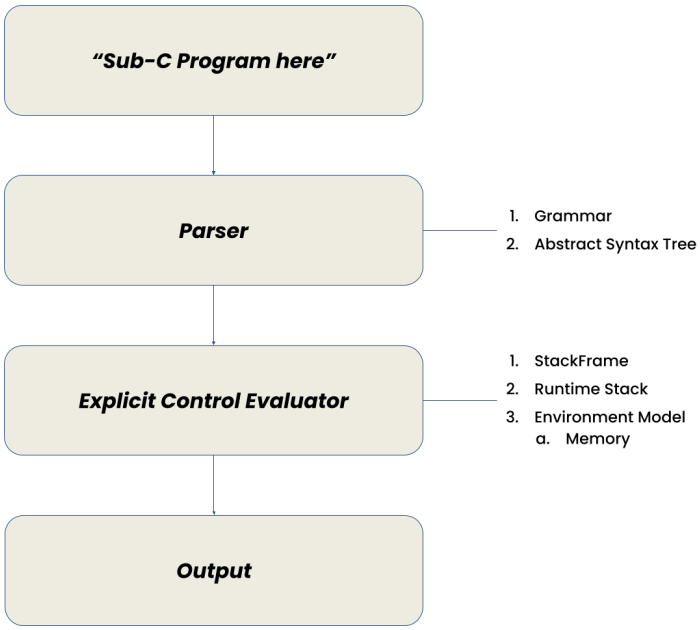


The main objectives of our project are as follows:

- Develop a parser to translate the input code into an AST (Abstract Syntax Tree) representation.
- Implement an explicit control evaluator to evaluate the control flow of the program.

    - Able to perform on language constructs: variable and function declarations, blocks, conditionals statements and expressions, while loops
- Implement an abstraction for C's runtime stack to keep track of function definition and function calls.
- Implement an environment model that handles memory management which includes heap allocation for dynamic memory allocation and track variable values.
- Implement a symbol table to keep track of variable declarations and assignments.
- Interpret the input code and output the program's results to the front-end.

To achieve our objectives, we created an implementation overview of our project, which will discuss more in the later sections below:



Sublanguage C Program

Our sublanguage of C is derived from the C11 standard derived from antlr and includes only a subset of the language features. Specifically, our version of the language includes:

- support for basic calculator operations (such as addition, subtraction, multiplication, and division), unary operators, functions, assignments, and variables.
- We do not include more advanced language constructs such as if-else statements, type declarations, or pointer arithmetic.
- Additionally, we assume that commonly used functions such as printf(), malloc(), and free() are built-in and do not include declarations for these functions.

By focusing on this subset of the language, we aim to provide a more manageable and straightforward language for our interpreter to interpret.

Parser

Once we formalised our sublanguage C program, we moved towards the parser. The first step in our language processing is tokenisation, where the input program is broken down into a series of tokens based on our sublanguage C grammar. These tokens are then passed through the parser, which builds an abstract syntax tree (AST) based on the input program. The AST is then simplified by removing complex constructs and simplifying the syntax before being passed to the explicit control evaluator to generate the final output code.

Grammar

Our Sub-C grammar is derived from the C11 standard using EBNF (Extended Backus–Naur Form) syntax and includes a simplified version of the language. Our version includes only calculator (binary, unary operations), functions, assignments, and variables, while, if-else, type, and pointer arithmetic. We do not include declarations of header files, the grammar consists of the following rules:

```
grammar CJs;

program : stat | stat program;

stat: assg ';'
    | expr ';'
    | ifStat
    | def
    | whileStat
    | block
    | return ';'
    ;

expr: ID
    | INT
    | CHAR
    | STRING
    | funCall
    | unaryOp expr
    | expr binaryOp expr
    | '(' expr ')'
    | arrAccess
    ;
```

In the CJs grammar, the *program* token is defined as a sequence of one or more *stat* tokens. The *stat* token represents a statement in the sublanguage C and can take on several forms, including

- *assg* - an assignment statement, where an identifier is assigned an expression.
- *expr* - an expression statement, where an expression is evaluated
- *ifStat* - an if-else statement, where a condition is evaluated, and one of two blocks of code is executed based on whether the condition is true or false.
- *def* - a definition statement, which can be either a variable definition or a function definition.
- *whileStat* - a while loop statement, where a condition is evaluated and a block of code is executed repeatedly as long as the condition is true.
- *block* - a block of code enclosed in curly braces {}. This can contain any number of stat tokens and is used to group statements together.

The *program* and *stat* tokens are important in defining the structure of a sublanguage C program, as they provide the basic building blocks for constructing valid code.

expr is a non-terminal in the Sublanguage C grammar that represents an expression. It has ten productions that can generate expressions involving identifiers, integers, characters, strings, function calls, unary operators, binary operators, parentheses, and array accesses.

```
return : 'return' (expr)?;

varDef: type ID
    | type assg
    | structInit
    | arrDef
    ;

structInit: 'struct' ID ID ;

assg: ID '=' expr
      | '*'ID '=' expr;


whileStat : 'while' '(' expr ')' block;


ifStat  : 'if' '(' expr ')' block 'else' block ;


def : varDef ';' | funDef | structDef;


funDef : type funcName '(' ')' '{' program '}'
       | type funcName  '(' (varDef) (',' varDef)* ')' '{' program
'}';
funcName: ID;


funCall : funcName '(' expr (',' expr)* ')'
        | funcName '(' ')';
```

Continuing the grammar, we also have:

- *return* - a keyword in the C language that indicates the return of a value from a function. In the grammar, return is used to indicate the return statement in a program. It can optionally be followed by an expression that returns a value.

- *varDef* - is a non-terminal that defines a variable declaration in the grammar. It can be used to declare variables with or without an initial value. A variable declaration includes a type and an identifier.

- *structInit* - is a non-terminal that defines the initialization of a struct in the grammar. It is used to initialise a variable of a struct type, which is declared using the *structDef* non-terminal. It takes two identifiers as arguments, where the first identifier is the name of the struct type, and the second identifier is the name of the variable being initialised.

- *assg* - is a non-terminal that defines an assignment statement in the grammar. It is used to assign a value to a variable. It takes an identifier and an expression as arguments. It also supports the assignment of a value to a dereferenced pointer using the * operator.
- *whileStat* - is a non-terminal that defines a while loop in the grammar. It is used to execute a block of code repeatedly while a condition is true. It takes an expression as an argument, which is used to evaluate the condition for the loop.
- *ifStat* - is a non-terminal that defines an if-else statement in the grammar. It is used to execute a block of code conditionally based on the value of an expression. It takes an expression as an argument, which is used to evaluate the condition for the statement.
- *def* - is a non-terminal that defines a variable or function definition in the grammar. It is used to define variables or functions in a program. It can be used to define a variable using *varDef*, a struct using *structDef*, or a function using *funDef*.
- *funDef* - is a non-terminal that defines a function declaration or definition in the grammar. It is used to define a function with a return type, a function name, a parameter list, and a block of code. It can also be used to declare a function with an empty parameter list.
- *funcName* - is a non-terminal that defines a function name in the grammar. It is used to specify the name of a function in a function declaration or definition.
- *funCall* - is a non-terminal that defines a function call in the grammar. It is used to call a function in a program. It takes a function name and a list of arguments as arguments. If the function takes no arguments, an empty parameter list can be used.

Continuing the grammar, we also have:

```
unaryOp :   '++'
          | '--'
          | '!'
          | '*'
          | '&'
          | '+'
          | '-'
          ;


binaryOp : '*'
          | '+'
          | '-'
          | '/'
          | '%'
          | '&&'
          | '||'
          | '=='
          | '<'
          | '>'
          | '>='
          | '<='
          ;


block : '{' program '}';


type  : 'void'
      | 'int'
      | 'char'
      | 'void*'
      | 'int*'
      | 'char*'
      ;

structMember: type ID ';' ;
structDef: 'struct' ID '{' (structMember)* '}' ';'
;
arrDef: type ID'[]' '=' '{' expr (',' expr)* '}' ;
arrAccess: ID ('[' INT ']')* ;
```

- *unaryOp*: This is a non-terminal that represents a unary operation that can be applied to an expression. It includes operators like increment (++), decrement (--), logical NOT (!), dereference (*), address-of (&), unary plus (+) and unary minus (-).
- *binaryOp*: This is a non-terminal that represents a binary operation that can be applied to two expressions. It includes arithmetic operators like addition (+), subtraction (-), multiplication (*), division (/) and modulus (%). It also includes relational and logical operators like less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equal to (==), not equal to (!=), logical AND (&&) and logical OR (||).
- *block*: This is a non-terminal that represents a block of code enclosed in curly braces ({}). It can contain zero or more statements.
- *type*: This is a non-terminal that represents a C data type. It includes void, int, char, void*, int*, and char*.

- *structMember*: This is a non-terminal that represents a member of a struct. It consists of a data type and an identifier, separated by a space and terminated by a semicolon.
- *structDef*: This is a non-terminal that represents a struct definition. It consists of the keyword struct, followed by an identifier for the struct type, then an open curly brace, a list of struct members, and finally a closing semicolon.
- *arrDef*: This is a non-terminal that represents the definition of an array. It consists of a data type, an identifier, an array notation with empty or explicit size, and a list of comma-separated values enclosed in curly braces.
- *arrAccess*: This is a non-terminal that represents the access of an element in an array. It consists of an identifier for the array followed by a pair of square brackets enclosing an integer expression that represents the index of the element to be accessed.

And final part of the grammar,

```
//====== Lexical components ======================

CHAR : '\'' .*? '\'';
STRING : '"' (' '..'~')* '"';
WS     : [ \t\n\r]+ -> skip ;
ADD    : '+' ;
SUB    : '-' ;
MUL    : '*' ;
DIV    : '/' ;
MOD    : '%' ;
ARRAY  : '[]' ;
COMMA  : ',';
COLON  : ';';
EQ   : '=';
AND : '&&' ;
OR : '||' ;
NOT : '!' ;
LPAREN : '(' ;
RPAREN : ')' ;
LCURLY : '{' ;
RCURLY : '}' ;
INT : [0-9]+ ;
ID: [a-zA-Z_][a-zA-Z_0-9]* ;
```

The lexical components in this grammar define the tokens that can be used to construct valid CJs programs. These tokens are matched by the parser to generate a parse tree representing the program's syntax.
Here is a brief description of the lexical components used in the grammar:

- CHAR: matches a character literal enclosed in single quotes (e.g., 'a', '\\n')
- STRING: matches a string literal enclosed in double quotes (e.g., "hello world")

- WS: matches one or more whitespace characters (spaces, tabs, newlines, etc.) and is ignored by the parser
- ADD, SUB, MUL, DIV, MOD: matches arithmetic operators +, -, *, /, % respectively.
- ARRAY: matches the array operator []
- COMMA: matches the comma, used to separate function arguments and array elements
- COLON: matches the semicolon ; used to terminate statements
- EQ: matches the equals sign =
- AND, OR: matches the logical operators && and || respectively
- NOT: matches the logical operator!
- LPAREN, RPAREN: matches the left and right parentheses (and)
- LCURLY, RCURLY: matches the left and right curly braces { and }
- INT: matches integer literals (e.g., 123, 0xFF, 0b1101)
- ID: matches identifiers (variable names, function names, etc.) that start with a letter or underscore and can contain letters, digits, and underscores.

The grammar includes production rules for programs, statements, expressions, functions, variables, structures, and operators. It also includes lexical components such as keywords, identifiers, operators, and literals. The parser uses the grammar rules to validate and parse Sub-C programs, breaking them down into smaller components that can be executed by the interpreter. By adhering to the Sub-C grammar, the interpreter can ensure that programs are syntactically correct and can be executed according to the rules of the language.

Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a data structure that represents the hierarchical structure of the source code according to the grammar rules. Once we have defined our grammar, we can use a parser generator like ANTLR to build an AST. ANTLR generates a parser that can build parse trees and also generates a visitor interface that makes it easy to respond to the recognition of phrases of interest. We can use the visitor interface to change how the tree is traversed; we can return values from each visitor function and even stop the visitor together.

In our implementation, we used the antlr4ts library from ANTLR to generate the necessary files for our parser; interpreter, tokens, lexer, listener, and generic visitor.

From the visitor generated from ANTLR library, we implement our visitor called “TreeBuilder”. This visitor implements a TreeNode interface to build our AST of Sub-C. The “TreeNode” interface provides a single interface for building the AST, which has benefits and drawbacks. On the one hand, it simplifies the building process since all nodes have a uniform structure, but on the other hand, it limits the flexibility of the implementation.

To illustrate how the AST is built, we take the example of the Sub-C program:

```
int sum(int x, int y) {return x + y;}
int main() {
    printf("%d", sum(6,8));
    return 0;
}
```

The Sub-C program defines two functions - "sum" and "main". The "sum" function takes two integer arguments "x" and "y" and returns their sum. The "main" function prints the sum of 6 and 8 using the "sum" function, using the printf function to output the result to the console. It then returns 0 to indicate that the program executed successfully.

The resulting AST has a "Program" node as the root, which has two children: a "Statement" node representing the function definition of "sum," and another "Statement" node representing the function definition of "main."

The "main" function has two children representing a function call to "printf" and a "Return" statement. The function call to "printf" has two children representing a string literal and a function call to "sum."

Thus, based on the Sub-C program above, we get our AST as follows here.

Overall, using an AST allows for a more efficient and organised way of processing source code. It can simplify the analysis, transformation, and optimisation of code. Additionally, using a parser generator like ANTLR and a visitor interface like TreeBuilder allows for a more modular and reusable approach to building the AST.

Explicit Control Evaluator

After we have built the Abstract Syntax Tree (AST) using the TreeBuilder, the next step is to evaluate the AST by building an explicit control evaluator. In our Sub-C evaluator, we use the *tag* in the TreeNode to traverse the AST and perform the necessary operations.

To evaluate the AST, we implement a Runtime Stack derived from C. The Runtime Stack is used to handle function calls, return statements and local variables. We also implement an environment model which is used for variable mapping and memory management.

When evaluating the AST, we start from the root node of the tree and recursively traverse the tree, evaluating each node as we encounter it. We use the tag in each node to determine what operation needs to be performed.

Runtime Stack

The Runtime Stack is an implementation of the stack data structure used to manage the execution of functions during program evaluation, as defined here:

```
export class RuntimeStack {
    private frames: StackFrame[] = [];

    push(frame: StackFrame): void {
      this.frames.push(frame);
    }

    pop(): StackFrame | undefined {
      return this.frames.pop();
    }

    peek(): StackFrame {
      return this.frames[this.frames.length - 1];

  }
```

It is an integral part of the explicit control evaluator, which is responsible for executing the AST generated by the TreeBuilder.

In the code snippet provided, we can see that the Runtime Stack is implemented using an array of StackFrame defined below:

```
/**
 * Each StackFrame represents function data,
 *
 */
export interface StackFrame {
    name: string,
    returnType: string,
    params: any[],
    variables: Record<string, any> | undefined
    body: TreeNode | undefined
}
```

Each StackFrame represents the data associated with a function that is currently being executed. It contains information such as the name of the function, its return type, the parameters it takes, and the variables that are local to the function.

The StackFrame interface defines the structure of each frame in the stack. It includes the function name, its return type, the list of parameters it takes, and the variables local to the function. The body property represents the AST corresponding to the function's body.

During program evaluation, the Runtime Stack keeps track of the current function being executed. When a function is called, a new StackFrame object is created and pushed onto the stack. This StackFrame contains all the information required to execute the function, including its parameters and local variables. As the function executes, the stack is used to store and retrieve the values of these variables and parameters.

When a function completes execution, its StackFrame is popped off the stack, and the control is returned to the calling function. This process continues until the entire program has been executed.

Referring to our earlier sum example mentioned earlier above, when we evaluate the function definitions and function call of sum() and main(), the evaluator gets called in the specific switch case:

```
case 'FunDef':
                const fnName = this.evaluate(node.children!.funcName!, env)

                // Save the function definition to environment
                const returnType = this.evaluate(node.children!.returnType!, env)
                const params = node.children?.args!.map(param => this.evaluate(param, env))
|| []
                const bdy = node.children!.nextProg!
                const frame: StackFrame = {name: fnName, returnType, params, body: bdy,
variables: {}}
                env.define(fnName, frame)

                 // if it is the main function <= evaluate it, since its function definition
is execution
                 if (fnName == 'main') {
                    RTS.push(frame)
                    return this.evaluate(node.children!.nextProg!, env)
                }

                // Return since we're not actually evaluating the function here
                return;

            case 'FunCall':

                // do lookup of functions
                const fn_name:string = this.evaluate(node.funcName!, env)

                // map the value in the fn call to fn def params
                const fnCallParams = node.args!.map(param => this.evaluate(param, env)) || []
```

```
if (fn_name == 'printf') { // check for builtIns
    const string = this.evaluate(node.args![0], env)
    const va =  env.lookup(fn_name)(string,resultStack)
    this.output.push(va)
    return va;
}

const fn = env.lookup(fn_name)
if (!fn)
    throw new Error(`Function ${node.funcName!.text!} is not defined`);

const localEnv: Environment = env.extend()
for (let i = 0; i < fn.params.length; i++) {
    localEnv.define(fn.params[i].varName, fnCallParams[i])
}

// add to Runtime Stack
RTS.push(fn)
const r = this.evaluate(fn.body!, localEnv)
resultStack.push(r) // add final value to ResultStack
RTS.pop()
return r;
```

In the *FunDef* case, a new *StackFrame* is created and populated with the relevant information, such as the function name, return type, parameters, and body. This StackFrame is then saved to the current environment by calling the define method of the environment object. If the function being *defined* is the *main* function, the *StackFrame* is pushed onto the *RTS* and the body of the function is immediately evaluated using the current environment.

In the *FunCall* case, the function name and arguments are evaluated and used to perform a lookup in the current environment to retrieve the corresponding *StackFrame* for the function being called. A new local environment is extended and the function parameters are mapped to the arguments passed in the function call. The *StackFrame* for the function is then pushed onto the *RTS*, and the function body is evaluated using the new local environment. The result of the function call is pushed onto the *resultStack*, and the *StackFrame* is popped from the *RTS*. If the function being called is a built-in function, it is handled separately.

In the Sub-C program example, the function sum is defined with two integer parameters and returns the sum of the parameters. In the main function, the printf function is called with the result of calling the sum function with the arguments 6 and 8. The output of the program is the result of the printf function call, which should be 14.

When the sum function is called from main, a new StackFrame is created with the function name, return type, parameters, and body, and pushed onto the runtime stack RTS. Then, a new environment is created to map the parameter values to the parameter names in the function

definition, and the function body is evaluated with the new environment. Finally, the StackFrame is popped from RTS, and the return value is returned to the main function.

In the case of the printf function, a built-in implementation is used, and the argument is evaluated and output to the output array.

Environment

Environment

variable map

value table

vname	value
x	12
ch	‘c’
…	…
…	…

type table

vname	type
x	int
ch	char
…	…
…	…

address table

vname	type
x	Addr0
ch	Addr0+32
…	…
…	…

function map

vname	value
sum	args
val	args
…	…
…	…

1. For every Environment created, we separately store variables and functions in variable map and function map.
2. We use the define / fndefine function to add a variable or function to the environment, and the lookup / fnlookup function to find and match.

```
typeSize: (typeName: string) => {
        var typeSizeMap: {[key: string]: number} = {
            "void": 32,
            "int":  32,
            "char": 32,
            "void*":32,
            "int*": 32,
            "char*":32
        }
        return typeSizeMap[typeName]
    }
```

```
import { stdio } from "../clib/stdio";

export class Environment {
    ...
    // variable mapping
    values: { [key: string]: any } = {};
    address: { [key: string]: number } = {};

    ...

    private baseid = 1e9 + Math.floor(Math.random() * 8e9);

    ...

    define(name: string, value: any, varType: any): void {
      this.values[name] = value;
      this.address[name] = this.baseid;
      this.baseid += stdio.typeSize(varType)
    }

  }
```

- With the initialisation of the global environment, we generate and allocate a base address to it and update the offset with the definition of variables.
- 32-bit primitive types.

- We put the size map of basic types in the clib/stdio.ts for free(), and we delete the variable from the addressing table.

Building the System

Backend

```
1. Install the project dependencies:
   – yarn install
2. To build:
   – yarn build
3. Add "c-slang" to PATH:
   – yarn link
```

Frontend

```
1. Install the project dependencies:
   – yarn install
2. Link to backend:
   – yarn link “c-slang”
3. To run:
   – yarn run start
```

Ran and tested on Node version: 16.20

Test Cases

Test Case #1: Simple Calculation

```
int val(int a){
   return a * 3 + 2;
}

int main(){
 printf("%d", val(4));
 return 0;
}
```

Expected Output: “14”

Test Case #2: Address

```
int main() {
    int a = 1;
    return &a;
}
```

Expected Output: ???

Test Case #3: Sum

```
int sum(int x, int y) {
    return x + y;
}

int main() {
    printf("%d %d", sum(7,3), sum(6,8));
    return 0;
}
```

Expected Output: “10 14”

Test Case #4: Recursion

```
int val(int a){
    if(a > 100){
        return a;
    }else{
        return val(a + 1);
    }
}

int main(){
  printf("%d", val(4));
  return 0;
}
```

Expected Output: “101”

Test Case #5: Frontend Stack Overflow

```
int val(int a){
    if(a > 1000){
        return a;
    }else{
        return val(a + 1);
    }
}
```

```
int main(){
  printf("%d", val(4));
  return 0;
}
```

Expected Output: “Maximum call stack size exceeded”

Test Case #6: Factorial

```
int factorial(int n) {
    if(n == 0) {
       return 1;
    } else {
       return n * factorial(n-1);
    }
 }

int main(){
  printf("%d", factorial(4));
  return 0;
}
```

Expected Output: “24”

Test Case #7: while

```
int main() {
    int x = 0;
    while (x < 10) {
        printf("%d\n", x);
        x = x + 1;
    }
    return 0;
}
```

Expected Output: “0 1 2 3 4 5 6 7 8 9’

Test Case #8: Hello World

```
int main() {
    printf("Hello World");
}
```

Expected Output: “Hello World”

Test Case #9: Fibonacci

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }

}

int main(){
  printf("%d", fibonacci(10) );
  return 0;
}
```

Expected Output: “55”

Test Case #10: Max function

```
int max(int x, int y, int z) {
    if (x >= y) {
        if (x >= z) {return x;}
        else {return z;}
    }
    else {
        if (y >= z) { return y;}
        else {return z;}
    }
}
int main() {
    printf("%d", max(max(10, 20, 30), max(50, 25, 75), max(533, 231, 24)));
    return 0;
}
```

Expected Output: 533

Challenges and Learnings

- Developing the parser for the Sub-C language was challenging and took a long time to get right.
- Finalising the grammar was especially difficult because even small changes could impact the visitor and TreeNode used by the evaluator.
- The use of TypeScript's strict type safety helped to ensure that all data types used were compatible with the language.
- In the future, the team plans to further segment the sections of the TreeNode for specific semantics instead of having them all in one interface with optional fields.

- Currently, the evaluator only supports types of int and int pointer arithmetic, and other types defined in the grammar are not yet implemented.
- The team also plans to improve the language by supporting array and struct evaluation and handling error cases such as syntax errors and undefined variables in future iterations.
- One key observation we noticed from our current implementation, that our values work best as function values instead of the actual value, for instance:
    - This would give us an error:

```
int main() {
    printf("%d", 5); // undefined
}
```

    - This would work:

```
int return_val(int a) {
    return a;
}

int main() {
    printf("%d", return_val(5)); // displays “5”
    return 0;
}
```

    - Reason:
        - Our implementation heavily does a lookup on the function params as a priority than the local variables under the function body

Summary

We have created an interpreter for a sublanguage of C derived from C11 Standard in Javascript using antlr as parsing system to derive our grammar and build our Abstract Syntax Tree.

We have created an explicit control evaluator as our interpreter based on our abstract syntax tree. In the explicit control evaluator, we created a runtime stack derived from C to keep track of our function definitions and calls and an environment model to track variable mapping and memory management.

Developing the Sub-C language gave us a deep understanding of C syntax, semantics, and memory model, and despite the hurdles and challenges, the project also gave us an immense appreciation for the mental model behind implementing C in Javascript.