A Realistic Explicit Control Evaluator for C

Project report for **CS4215**

Lester Leong (A0269245W),
Lui Wen-Jie, Benjamin (A0214362N)
School of Computing
National University of Singapore
April 2022

Frontend: `https://github.com/cs4215-2023/CS4215-frontend`
Language Processor: `https://github.com/cs4215-2023/c-interpreter`

Contents

1 Introduction

C is a programming language that has memory models present such as stack and heap. Thus, we decided to implement a realistic explicit control evaluator for C (RECEC) to align closely with the memory models present.

The first section, Section 2, will cover the features implemented. The second main section, Section 3, will include the technical specification of our RECEC.

1.1 T-Diagrams

T-diagrams of all major language processing steps that the project utilizes

2 Features

This section will list the various features that were implemented and certain design choices that we made. The system can be built and tested by following the instructions of the language processor repository, followed by the frontend repository accordingly. Alternatively, one can use the dedicated code editor at `https://cs4215-2023.github.io/CS4215-frontend/`. Note that if there is no output on right panel, the return value should be taken as `NaN`.

The main entry point of each program is a `main` function **without any arguments**. This is implicitly called by the interpreter itself. Also note that comments are currently **not supported**, but for the clarity of code examples, they will be use just like how it is implemented in C. Not that these examples are **not valid programs** in RECEC because of the comments present.

2.1 Primitive Type

RECEC supports the basic types present in C, primarily, `int`, `void`, `float`, `char`. These types can be defined with their respective pointer and array types.

An example of types in RECEC:

```
int a;
float a[];
char* a_pter;
```

2.2 Basic Operators

RECEC supports various basic operators.

- Unary `!`, `&`, `*`, `+` and `-`.
- Arithmetic operators `+`, `-`, `/`, `*`, `%`
- Bitwise operators `&`, `|`, `^`, `>>`, `<<`
- Logical operators `&&`, `||`, `^`
- Update operators `++`, `--`

This is an example of how basic operators are used in RECEC:

```
!(1); // 0
!(0); // 1
int a = 1;
&a; // Gives address of variable a
*a; // Dereference variable a
a++; // Variable a contains the value of 2
```

As we are not supporting `_bool_` type of C, logical operators and unary `!` will give an integer value of `1` if `true` and `0` otherwise.

For equality checking, `==` and `!=`, we check if the values are the same, regardless of whether they have the same address or not.

2.3 Variable and Function Declarations

Each variable declaration must come with a type. The valid types are present in 2.1 section. This flexibility, however, is not the same for function declarations.

For function declarations, only the basic types are allowed. If the return type is `char`, return values of characters will be converted to their respective ascii value.

Here is an example of some variable and function declarations used in RECEC:

```
int* a;
int a[4] = {1,2,3,4};
char foo() {
return 'a'; // returns 97
}
```

2.4 Conditionals

There are 2 types of conditionals in C, namely `if-else` statements and conditional expressions. Example conditionals can be written as so:

```
int a = 1;
// if-else statement
if (a == 0) {
// do something
} else {
// do something else
}

// conditional expressions
a == 0 ? 2 : 3;
```

2.5 Loops

Similar to C, this sublanguage also implements the 3 main loops found in C, namely the for-loop, while loop and the do-while loop. These loops do not return any values at the end if there are no `return` statements within the body of the loop itself. Furthermore, loops do implement intermediate control functionalities such as `continue` and `break`, however, `return` statements can be used in loops.

In the case of for loops, **all initialization, condition and update components are required**. Here are examples of each of the loop:

```
// for loop
for (int i = 0; i < 10; i++) {
// do something
}

while (i--) {
// do something
}

do {
// something
} while (i--);
```

2.6 Arrays

Arrays in RECEC have 2 main requirements. Firstly, the type declarations cannot contain pointer or another array type declaration. Secondly, if an array is initialised, all the values should be of the same type with no variables in it. This could be better explained with a sample program here:

```
int a[] = {1, 2}; // valid array
int b = 2;
int c[] = {1, b}; // invalid array; presence of variable in array
int* d[]; // invalid array; array of pointers
```

2.7 Pointers

Like C, arrays are pointers as well. Simple pointer arithmetic is supported such as `++`. However, more complex operations such as `*(c + 1)` is currently not supported. The dereferencing of pointers works similarly to the C language.

The address that a pointer points to is taken to be an `int` type. This value will be represented in base 10 instead instead of the usual hexadecimal representation.

```
int a = 1;
int* a_pter = &a;
*a_pter; // value of 1
a_pter++;
*a_pter; // value of 2
```

2.8 Builtins

RECEC supports 3 primary builtin functions, namely, `printf`, `malloc` and `free`.

2.8.1 `printf`

`printf` prints to the console instead of the UI display. To view the result of printf, one is required to look at the console itself to see the value printed. Format specifiers supported by `printf` are `"%c"`,`"%d"` and `"%f"`. Type casting is done here as well, based on the format specifier provided in the input string. Note that the input **must be a string**.

Here are some sample `printf` in RECEC:

```
printf("hello world");
printf("hello worl%d", 3); // prints hello worl3
printf("%d", 1.0); // prints  1
```

2.8.2 `malloc`

`malloc` allocates a variable amount of heap memory based on the argument provided. The argument is an expression which specifies `n` number of nodes in the heap to be allocated to the variable. Here is a sample for `malloc` in RECEC:

```
int* c = malloc(3); //length of c is 3
    int a = 5;
    c[0] = 5;
    c[1] = 2;
    c[2] = 3+a;
    c[2]; //value of 8
```

2.8.3 `free`

`free` deallocates the specified heap memory based on the argument provided to it, which is a variable. The variable has to be initialized with `malloc` first before `free` can be used to free up memory used by that variable.

Here is a sample for `free` in RECEC:

```
int* c = malloc(3); //length of c is 3
    c[0] = 5;
    c[1] = 2;
    c[2] = 4;
    free(c);
```

2.9 Type Checking

In RECEC, type checking is done before evaluation takes place. This means that the typechecker parses the entire program and check **every line** for its type. If there is a mismatch in types, the type checker throws a type error. Unlike C where warnings are thrown when a type mismatched is detected, RECEC will terminate the program instead.

The following are some implicit type casting examples that works in RECEC:

```
2.0/ 4; // evaluates to a float type
int a = 4 + 'a' // evaluates to an int
char b = 4 + 'b' // evaluates to a char
```

3 Specifications

In this section, we define the semantics of RECEC and describe the implementation of RECEC in detail.

For steps on running RECEC program, please refer to Section 2. Once again, you can find the latest tagged release of RECEC at , and follow the instructions in the README to setup RECEC to run locally.

We first begin by defining the syntax of RECEC. We then bring in the type system and define it using static semantics. This is then synergised with how our interpreter behaves, described using the framework of structural operational semantics. Finally, we explain how the memory management is performed in RECEC.

3.1 Syntax

We divide the syntax of RECEC into two categories, *expressions* and *statements*.

3.1.1 Statement

This is what the statement BNF looks like:

program	::=	*statement...*	program
statement	::=	*expression* `;`	expression statement
	|	`return` *expression* `;`	return statement
	|	*if-statement*	conditional statement
	|	`while (` *expression* `)` *block*	while loop
	|	`do` *block* `while (` *expression* `) ;`	do-while loop
	|	`for (` *expression* `;` *expression* `;` *expression* `)` *block*	for loop
	|	*type name* `(` *parameters* `)` *block*	function declaration
parameters	::=	*type name*`(` `,` *type name* `)` ...	parameters
block	::=	{ *statement...* }	block statement
if-statement	::=	`if (` *expression* `)` *block*	
		[`else (` *block* | *if-statement* `)`]	conditional statement

In for-loops, the expressions are referred to as `initialisation`, `test` and `update`. The for-loop is syntactic sugar for a while-loop. Firstly, `initialisation` is converted to an `expression statement`. Then, `test` is placed as the condition for the while loop. `update` is then appended to the end of the for-loop body. `body` refers to the original for-loop body. What this looks like in the underlying implementation is something like the following:

```
initialisation;
while (test) {
body;
update;
}
```

Similarly, the implementation of do-while is also converted to a while-loop. The only difference is that the body is first executed first, before checking the condition. Hence, we execute the body independently, and then we follow it with a while-loop immediately after. `body` refers to the original do-while loop body and `test` refers to the condition in the while statement. Here is what the underlying implementation looks like:

```
body;
while (test) {
body;
}
```

3.2 Expressions

Now, we will go into details of what `expression` was referring to in 3.1.1, but firstly, this is the BNF for expressions:

expression::=	*integer*	primitive integer expression
	| *float*	primitive float expression
	| *char*	primitive char expression
	| *string*	primitive string expression
	| *name*	name expression
	| *expression binary-operator expression*	binary operator combination
	| *unary-operator expression*	unary operator combination
	| *expression binary-logical expression*	logical composition
	| *expression* `(` *[expressions, ...]* `)`	function application
	| *expression* `?` *expression* `:` *expression*	conditional expression
	| *assignment*	assignment
	| *expression*`[`*expression*`]`	array access
	| `[` *expressions* `]`	literal array expression
	| `(` *expression* `)`	parenthesised expression
	| *type*`*` *name*	pointer expression
	| `*`*pointer*	pointer dereference expression
	| `&`*pointer*	pointer reference expression
	| *name(postfix-operator)*	postfix expression
binary-operator ::=	`+` | `-` | `*` | `/` | `%` | `==` | `!=`	
	| `>` | `<` | `>=` | `<=`	binary operator
unary-operator::=	`!` | `-`| `+`	unary operator
binary-logical::=	`&&` | `||` | `^`	logical composition symbol
bitwise-operator::=	`<<` | `>>`| `&`| `|`	bitwise operator

`integer`, `float`, `char` refers to values that fall within the range of `int`, `float` and `char` represented in C respectively. To differentiate between `char` and `strings`, RECEC looks at the type of quotation marks that was parsed and process it accordingly. `strings` are stored as an array of `char` and is terminated by a null byte, `\0`.

In RECEC, there is a **restriction** imposed on values of arrays. For **all values** in the array, they **must** be the same kind. This means that an array can **only contain all** numbers or all floats, or all variable names. Here is an example program to better illustrate this point:

```
int a[] = {1 , 2}; // ok
int a[] = {1, 1.5}; // not ok, syntax error is thrown here
int b = 1;
int a[] = {1, b}; // not ok, mixture of variable and numbers
```

3.3 Static Semantics of RECEC's Type System

Not all statements in RECEC make sense. For example,

```
'a' + 1;
```

does not make sense, because we expect both operands to be of integer type but `'a'` is a character type. We thus say that this statement is *ill-typed*, because a typing condition is not met. Statements that meet these conditions are what we call *well-typed* in RECEC.

An expression `x + 3` within a statement may or may not be well-typed, depending on the type of `x`. Thus we define a type environment, denoted by Γ. Γ is a partial function from names to types, with which a name x is associated with type $\Gamma(x)$.

We define a relation $\Gamma[x \leftarrow t]\Gamma'$, which constructs a new type environment where $\Gamma'(y)$ is t if $y = x$ and $\Gamma(y)$ otherwise. The set of names, on which a type environment Γ is defined is called the domain of Γ, denoted by $dom(\Gamma)$. We define the empty type environment $\Gamma = \emptyset$.

3.3.1 Free Names

We need to be able to find out what names that are bounded by enclosing function definitions and those that are not. For example, the name `foo` occurs free in

```
void bar(int x) {
return foo(x);
}
```

since it is not declared by any surrounding `function` expression but `x` is bounded by `int x` declared in the parameters. Formally, we are looking for a function FV: RECEC $\rightarrow 2^V$ that defines the set of free names of a given RECEC program/expression. the relation FV on expression is defined by the following rules:

$$\frac{}{FV(x) = x} \qquad \frac{}{FV(n) = \emptyset} \qquad \frac{FV(E) = X}{FV(p_1[E]) = X} \qquad \frac{FV(E_1) = X_1 \; FV(E_2) = X_2}{FV(p_2[E_1, E_2]) = X_1 \cup X_2}$$

p_1 is defined as the set all unary operations, including those that exclusive to pointers. p_2 is defined as the set of all binary operations.

3.3.2 Expressions

The following rules denote the set of expressions in RECEC. Let i be an integer, f be a float and c be a char.

$$\frac{}{\Gamma \vdash \texttt{i : int}}\,[\text{IntegerT}] \qquad \frac{}{\Gamma \vdash \texttt{f : float}}\,[\text{FloatT}] \qquad \frac{}{\Gamma \vdash \texttt{f : char}}\,[\text{CharT}]$$

As `char` in C tend to be implicitly converted to their ASCII representations when binary operations are executed on them, RECEC will do the same. Furthermore, the resulting type of a binary operation depends on the type of both arguments. The precedence follows as such, `float`, `int`, `char`. What this means is that if one of the arguments is a `float` and the other is an `int`, the resulting type would be a `float`.

Furthermore, addition of `char` results in an `int`. The reason for this is that `char` is implicitly converted to its ASCII value before operations are carried out.

The table below shows the resulting type from 2 argument types. For simplicity, only the primitive binary operation of $+$ is shown, but all the other primitive binary operations (logical included) follows the same rules.

operator	argument 1	argument 2	result
`+`	integer	integer	integer
`+`	char	integer	integer
`+`	integer	char	integer
`+`	integer	float	float
`+`	float	float	float
`+`	float	integer	float
`+`	float	char	float
`+`	char	float	float
`+`	char	char	integer
`==`	any	any	integer
`!=`	any	any	integer
`!`	any		integer
`-`	any		any
`+`	any		any
`*`	any		any
`&`	any		integer

For conditional expressions, we also enforce that the consequent and alternative expressions have the same type:

$$\frac{\Gamma \vdash E \ : \ \texttt{int} \qquad \Gamma \vdash E_1 \ : \ t \qquad \Gamma \vdash E_2 \ : \ t}{\Gamma \vdash E \ \texttt{?} \ E_1 \ \texttt{:} \ E_2 \ : t}\text{[ConditionalExpression]}$$

3.3.3 Statements

A given statement might consist of a sequence of variable declarations, return statements or expression statements. We define a typing relation on such statements in two steps. First, we extract all type declarations and extend the given type environment by bindings of the names to the types of the type declarations.

$$\frac{\Gamma[x \leftarrow t]\Gamma' \quad \Gamma' \vdash S : t'}{\Gamma \vdash x = t; \ S : \ t'}\text{[Sequence]} \qquad \frac{\Gamma \vdash E : \Gamma(x)}{\Gamma \vdash x = E;}\text{[VarDeclaration]}$$

$$\frac{\Gamma \vdash S_1 : t_1 \quad \Gamma \vdash S_2 : t_2 \quad S_2 \text{ is not a return statement}}{\Gamma \vdash S_1 S_2 : t_2}\text{[SequenceStatements]}$$

Functions in RECEC are allowed to avoid return statements. This means that the implicit return type would be `void`. For simplicity, we assume that every function then has a return type, which is defined in the rules below.

$$\frac{\Gamma \vdash E : t}{\Gamma \vdash \texttt{return}E; \ S : return(t)}\text{[ReturnStatement]}$$

$$\frac{\Gamma(f) = t_1 * ... * t_n > \text{undefined} \quad \Gamma[x_1 \leftarrow t_1]\Gamma_1 \cdots \Gamma_{n-1}[x_{n-1} \leftarrow t_{n-1}]\Gamma_n, \Gamma_n \vdash S : return(t)}{\Gamma \vdash \texttt{return(t)}\, f(x_1, ..., x_n)\{S\}; \ : undefined}\text{[FunctionDeclaration]}$$

In the case of loops, type checking is done for the loop bodies. No typing is enforced for `test` to be a boolean since in C, booleans are defined as either 0 or 1.

Furthermore for `if-statement`, each block does not need to have the same resulting type. Type checking in RECEC does not perform execution, hence all possible branches are checked for their types. If there are no return statements in the blocks, the resulting return type would be undefined.

$$\frac{\Gamma c_n : int \quad \Gamma S_1 : t_1 \quad \Gamma S_2 : t_2 ... \Gamma S_n : t_n}{\Gamma \vdash \texttt{if}(c_1)\{S_1\}\ \texttt{else if}(c_2)\{S_2\}\ ...\texttt{else}\ \{S_n\};\ : undefined}\text{[if-statement]}$$

3.4 Structural Operational Semantics of RECEC

We define result of executing a program `p` according to our structural operational semantics as the value `v`, where $(\epsilon_s, \Delta_0) \rightrightarrows_s^* (v, \epsilon_a, \Delta')$ and where $\rightrightarrows_s^*$ is the reflexive transitive closure of $\rightrightarrows_s$. Here ϵ_s is the empty stash, ϵ_a is the empty agenda, and Δ_0 is an initial environment.

In other words, we execute a program `p` by making $\rightrightarrows_s$ transitions starting with the empty stash, `p` as the only item on the agenda, and the environment Δ_0 until the agenda is empty. The stash then has the result of the program execution as its only agenda item. In this case, we values on the stash are actually addresses. A conversion from address to value is performed as the final step before returning the value. If the stash is empty however, then we return `undefined` instead.

For the purpose of defining the semantics, we shall assume that the values are pushed onto the stash instead of the address in the actual implementation.

3.4.1 Sequences

We guarantee that a sequence of statements are evaluated in the order they are presented, with the empty statement returning `undefined`. Let $C = \{c_1, c_2, ...c_n\}$ be the set of statements. Then we can define a sequence as follows:

$$\frac{}{(s, C.a, \Delta) \rightrightarrows (s, c_1.POP.c_2.POP...c_n.a, \Delta)}$$

`POP` removes the top most instruction from the stash.

3.4.2 Names and Blocks

Names and blocks are interpreter on the fly. Since we are able to tell the difference between a declaration and an assignment, we can first place the value in the current environment.

Upon assignments, we then obtain the variable from the current environment/look at previous environments and set the appropriate value. For blocks, environments are restored after the block via a `RestoreEnvironment` instruction.

$$\frac{}{(s, int x = e;.a, \Delta) \rightrightarrows (s, e.ASSIGN\ x.a, \Delta)} \qquad \frac{}{(s, x = e;.a, \Delta) \rightrightarrows (s, e.ASSIGN\ x.a, \Delta)}$$

3.4.3 Conditional Statements

The entire is conditional is pushed onto the agenda but with `test` on top. Depending on the outcome of `test`, we then decide which branch to take.

$$\frac{}{(s, if\ (e)\ p_1\ else\ p_2.a, \Delta) \rightrightarrows (s, e.BRANCH\ p_1\ p_2.a, \Delta)} \qquad \frac{}{(1.s, BRANCH\ p_1\ p_2.a, \Delta) \rightrightarrows (s, p_1.a, \Delta)}$$

$$\frac{}{(0.s, BRANCH\ p_1\ p_2.a, \Delta) \rightrightarrows (s, p_2.a, \Delta)}$$

3.4.4 Loops

In RECEC, all loops are essentially some variable of the `while` loop. Here, we will describe the semantics of the `while` loop, which will cover all 3 loops present in RECEC. Since loops do not produce values, we push a empty statement, S_e onto the agenda

$$\frac{}{(s, while(e)p.a, \Delta) \rightrightarrows (s, e.while(e)p.S_e.a, \Delta)} \qquad \frac{}{(1.s, while(e)p.S_e.a, \Delta) \rightrightarrows (s, p.POP.e.while(e)p.S_e.a, \Delta)}$$

$$\frac{}{(0.s, while(e)p.S_e.a, \Delta) \rightrightarrows (s, S_e.a, \Delta)}$$

If the condition e evaluates to true, the body, `POP` and the entire while loop is pushed onto the agenda. Otherwise, we don't push anything onto the agenda.

3.4.5 Functions

In the case of function declarations, they are implicitly converted to lambda expressions and stored as `CLOSURE` on the stash. Each value x_n corresponds to a parameter of the syntax `<type> <name>`.

$$\frac{}{(s, voidfoo(x_1...x_n)p.a, \Delta) \rightrightarrows (s, CLOSURE(x_1...x_n)p\Delta.a, \Delta)}$$

Execution of an APPLY n instruction expects its components on the stash: n arguments (in reverse order), followed by the function to be applied. APPLY firstly checks that the number of arguments are valid. Then, it would perform a binding in the current CLOSURE environment to the respective parameter variables.

Finally a MARK instruction is left on the agenda so that execution of return statements can abandon subsequent statements. After MARK is reached, the current environment is removed.

Return statements leave a RESET instruction on the agenda. It will pop an item from the agenda and push itself back on the agenda if a MARK instruction has not been reached.

3.5 Memory

To account for memory usage in the RECEC, a stack and heap memory structure will be implemented. These are two distinct regions of memory used for different purposes. The stack is used for temporary data, and its size is typically limited. In contrast, the heap is used for data that needs to persist beyond the lifetime of a function or is too large for the stack.

Pointer tagging is used in the stack and heap memory to allocate type information and child nodes to each address in the memory.

Each valid address in the memory takes up 64 bits. Each block of 64 bits is divided into two blocks of 32 bits each for pointer tagging and value storage respectively. The functions `mem_get` and `mem_set` allow for reading and writing of individual blocks to memory. The memory only stores literal and pointer values.

```
public mem_get = (address: number) => this.memoryView.getFloat32(address)

public mem_set = (address: number, x: number) => this.memoryView.setFloat32(address, x)
```

3.5.1 Stack

The stack is a contiguous region of memory used to store local variables, function parameters, and return addresses. The stack utilizes 64 bits of memory per address. The first 32 bits are reserved for pointer tags. The last 32 bits are used for values.

The stack memory contains a `base_pointer`, which points to the base of the stack for the current scope, and a `stack_pointer`, which used to keep track of the current position of the stack and to allocate and deallocate memory from the stack as needed..

The stack contains two main functions, `push` and `pop`, which are responsible for allocating and deallocating memory. The `push` function returns the address of the allocated memory, while the `pop` function returns the value and type from the top of the stack.

```
public push(tag: number, x: number) {
  const address = this.stack_pointer
  this.mem_set(address, tag)
  this.mem_set(address + this.word_size / 2, x)

  this.stack_pointer += this.word_size
  return address
}

public pop() {
  const address = this.stack_pointer
  const val = this.mem_get(address - this.word_size / 2)
  const type = this.mem_get(address - this.word_size)

  this.stack_pointer -= this.word_size
  return [~~type, val]
}
```

Variables can be declared but not initialized. To account for this, a similar function can replace `push`, which is `allocate_one`. When a variable is declared, `allocate_one` will be called to allocate memory to the variable, but nothing is written to it.

```
public allocate_one() {
  //allocate one slot of data
  const address = this.stack_pointer
  this.stack_pointer += this.word_size
  return address
}
```

When a function is called, the current `base_pointer` will be pushed onto the stack, followed by the function arguments and declarations. This forms a stack frame. When the function returns, the stack frame will be popped to obtain the old `base_pointer`, which will restore the `base_pointer` to its position in the previous scope.

```
 public enter_scope() {
  this.push(TAGS.base_pointer_tag, this.base_pointer) // save old pointer
  this.base_pointer = this.stack_pointer
}

public exit_scope() {
  while (true) {
    const [tag, value] = this.pop()
    if (tag === TAGS.base_pointer_tag) {
      this.base_pointer = value //reset base pointer to the previous scope
      break
    }
  }
}
```

3.5.2 Heap

The heap, on the other hand, is a region of memory used for dynamic memory allocation. It allows programs to allocate and deallocate memory at runtime, which is especially useful when the size of the data structures is not known at compile time. The stack utilizes 64 bits of memory per address. The heap is managed by the programmer and is not automatically freed by the runtime environment. Therefore, it's essential to explicitly free the memory once it's no longer needed to prevent memory leaks.

In C, memory allocation is performed by calling `malloc`, while memory deallocation is performed by calling `free`. For each node (each node represents one address) in the heap, the first 16 bits are reserved for pointer tags. The next 16 bits are used for the node's children. The last 32 bits are used for values.

When `malloc` is called, the function `allocate_n` is used to allocate `n` nodes of memory to the heap based on the argument provided by malloc. `allocate_n` tries to allocate a new node which is the next child of the previously allocated node until the size has been reached, or the heap runs out of memory. Once it has successfully allocated all `n` nodes, an `END_OF_MALLOC` tag will be written to the last node to indicate the end of the allocated memory, replacing its child node.

```
  public allocate_n(n: number) {
    if (this.free === -1) {
      throw Error('heap memory exhausted')
    }
    let index = 0
    const initial_addr = this.free
    while (index < n - 1) {
      //get the current free node
      // this.memoryView.setInt8(this.free, tag)
      this.free = this.get_child(this.free)
      if (this.free === TAGS.END_OF_FREE) {
        throw Error('heap memory exhausted')
      }
      index++
    }
    //at the last allocated memory
    const address = this.free
    this.free = this.get_child(this.free)
    this.set_child(address, TAGS.END_OF_MALLOC)

    //return start of memory allocated to the variable
    return initial_addr
  }
```

When `free` is called, the function `free_up_memory` is used to ensure that the memory that is freed up can be reused by the system. Starting from the node designated to be freed, memory is freed up until it reaches the end of the allocated memory. The freed up nodes are then 'linked up' with the previous free nodes in the heap, and the `free` pointer starts at the designated node.

```
public free_up_memory(address: number) {
    const initial_free = this.free

    //set free to be start of memory that is freed up
    this.free = address

    //get the child address of the starting point
    let child_addr = address
    let prev_addr = address
    while (true) {
      prev_addr = child_addr
      this.set_tag_and_value(child_addr, 0, 0)
      child_addr = this.get_child(child_addr)

      if (child_addr === TAGS.END_OF_MALLOC) {
```

```
        break
      }
    }
    this.set_tag_and_value(prev_addr, 0, 0)
    this.set_child(prev_addr, initial_free)
  }
```

3.5.3 Pointers and Arrays

In the memory, pointers are assigned pointer tags indicating the type of the literal it is pointing to, and that it is a pointer type. This helps to distinguish if the value read from memory is an address or a literal. Pointer types in memory will hold an address to another part of the memory that is also a pointer or a literal of the same type.

For arrays, they are implemented as a pointer which points to a series of addresses, depending on the size of the array. For the stack, the array is implemented as a contiguous series of addresses, and the start of the array is assigned to the pointer (or variable) governing the array. For the heap, the array is implemented as a linked list, with each node possessing a child node. This excludes the end node, which is the last element of the array. The start address of the array is also assigned to the pointer governing the array. This means that after many `malloc` and `free` operations, it is possible for arrays in the heap to not consist of contiguous addresses but are fragmented instead.

3.6 Testing

Tests were written using Jest framework. In each key component of RECEC, tests were written extensively to ensure that the functionalities behave as expected. To run the tests, simply execute `yarn test`. The instructions can also be found in the README of the RECEC repository `https://github.com/cs4215-2023/c-interpreter`.

3.7 Future Work

Though we were pleased with what we achieved and learnt from this project, there is definitely more work to be done.

3.7.1 Unsatisfied stretched goals

Our stretch goals of function pointers and structures were not met. Such features are important in C and for RECEC to have them would be great.

In the case of structures, this builds on the system of some form of object oriented programming(OOP). Furthermore, these structures can be used as types too. Hence the implementation of structures would mean that we would need to incorporate OOP and user-defined types into RECEC.

For function pointers, this would mean that now every function would require some memory on the heap.

3.7.2 Static memory

Static memory in C is a type of memory allocation that occurs at compile time and remains fixed throughout the execution of the program. Static memory is allocated for variables and data structures that are declared with the `static` keyword or outside of a function. Even though static memory and the `static` keyword were not implemented in RECEC, it would be useful to implement it in the future, since it represents a more realistic memory model for the C programming language.

3.7.3 Frontend syntax highlighting

It would be helpful for the online code editor to be able to visually highlight the exact location where a semantics or typing is violated.