CM30171

Compilers: Coursework report

Section 1 – Choices and justifications

Interpreter

The interpreter takes a program represented by an abstract syntax tree (obtained from the previous lexical analysis and parsing stages) and translates it to give the output of the program. It does this by traversing down the AST and evaluating each node as it appears.

To implement this, I made a recursive function with a series of switch-case statements. Each node that can appear in the AST has a different semantic requirement, which made switch-case very suitable for the problem; functions were made for each case to implement their semantic behaviour. Recursion is used to evaluate the child nodes of each node, which is important as the operands of some expression may be formed from further sub-expressions.

A key feature of abstraction for high level programs is the use of functions and variables. Attached to this is the idea of scope and extent of variables. In some programming languages, it is common to see the idea that variables defined within a function are local to that function, last until the end of the function, and can only be accessed within the function. For C--, lexical scope is used, which means that variables defined within a function can also be accessed by functions that are nested within the function in which the variable was defined. To implement these concepts, an environment to store the bindings of variables within a function is needed. This is implemented using structs from C; a frame represents an environment, that contains some bindings, which represents the attachment of a variable to its contents. Then in each function, variables can be accessed through this frame – each variable defined within a function is stored within its respective frame, hence any variables that are outside the scope cannot be accessed. However, an exception is needed at times due to lexical scoping; nested functions need some method of accessing the frame of the enclosing function. Therefore, a pointer is used to point to the frame of the enclosing function.

Functions can be seen as some encapsulation of sematic behaviour; semantics are defined by the abstract syntax tree and the nodes that appear within it, so functions are essentially abstract syntax trees. As mentioned above, scoping of variables is important and an environment is needed to implement this. Thus, a function is a combination of some abstract syntax tree and its environment. This forms a closure, which is represented as a struct in C – this includes a pointer to the starting node of the AST that represents the function, and its environment frame.

Since the behaviour of functions is defined by AST’s, we can view the entire program as some function. This requires there to be an environment that is attached to it - the global environment. Global variables are variables that are accessible everywhere within the program. This is inherently implemented using lexical scope if variables are defined within the global environment, as functions within the program are essentially nested functions of the whole program. Another useful feature that comes with having a global environment, is some access-point for the program to start executing. In C--, like with C, this is some function called main(). This is implemented as some pre-scan of the AST representing the whole program. From here, global variables and functions are declared, and the global environment is initialised, which is used later when the interpreter interprets the program.

Functions would not be very useful if they were only capable of being run once. Therefore, the program’s flow of execution must not be simply sequential. This idea of changing the flow of execution comes in many forms: from conditional selection statements, to recursion, and of course – function calls. Selection statements – like if-then and if-then-else – require a new data type of Booleans. The Boolean datatype can be easily represented using a 1 or a 0 but can also be represented by the flow of the program (in compilers, these are the two techniques used to implement this). I use a 1 for representing true conditions, and 0 for representing false ones for simplicity. Any name in the program can therefore represent one of three different things: integer, functions, and Booleans. I use a union within a value struct to represent the values that a name can have. The struct for bindings will therefore include as one of its elements, the value struct that represents the value a name is bound to. Function calls and recursion are implemented easily with closures. Since the closure points the node of the AST that represents that function’s semantics, the function can be run by interpreting the function using its closure.

Compiler

An easy, and popular approach to compiler writing is to divide the compilation process into two parts – a front-end and back-end. This is known as the analysis and synthesis model (Aho, Sethi, & Ullman). In the analysis stage, an intermediate representation is generated. This allows for portability of code, as front/back-ends of compilers don’t have to be rewritten for every machine architecture of programming language; they can be re-used and attached to a different back-end for the desired target machine. Intermediate representations also allow for machine-independent optimizations to be performed (Aho, Sethi & Ullman).

For this coursework, intermediate representations are converted into assembly code, which is then further processed by an assembler. Assembly languages allow for an easier code-generation process, but at the cost of assembler at the end of this process (Aho, Sethi & Ullman). The assembly language used is MIPs, a reduced instruction set computing architecture with 32 general purpose registers.

Intermediate Code

The intermediate representation required for the coursework was Three-address code. This was implemented using quadruples – TAC with the general form: destination = source 1 op source 2. Quadruples were represented using a struct in C – as they are essentially records of fields, making structs very suitable.

To generate the TAC, I used a recursive function to traverse an AST, with a series of switch-case statements to define what TAC statement needed to be generated for each node. The difference between the tree traversal here and the interpreter, is that here, it is not computing the result, but generating some code, which does.

Like the interpreter, I had different functions to generate TAC for different nodes. An Enum of operations was created, to represent the different types of operations a TAC statement can have. The selection of operators a TAC statement is allowed to have were chosen based on the fundamental high-level functionality that the program is allowed to have. For example, additions have their own operator “plus”, and so do if-statements, and so do relative operators. However, operators are not limited to arithmetic or ‘functional’ features. For example, operators were used for marking the start and end of a procedure, and things like labels were used to tell the machine where to jump for a given jump to that label. Using a smaller set of TAC operators that were high-level allowed for easier TAC generation and allows for future optimizations to be easier.

The TAC generated must be able to represent everything that is possible, so anything that can be done by the interpreter must also be represented by the TAC generated from the TAC generator. One such thing is variables. To bind variables in TAC, I used an abstract data structure called a symbol table – this was implemented using a hash-table of linked lists for its fast look-up speed, which is necessary for a symbol table, due to it being used to lookup names often. Symbol tables are essentially the environment equivalent for TAC; they keep track of scope and binding information for names (Aho, Sethi, Ullman).

As with the interpreter, this symbol table must be able to represent lexical scoping. To do this, I created a new symbol table for each declaration of a function; this symbol table would then have a pointer back to the enclosing function, and when a search of a variable is performed, if the variable is not found within its own symbol table, it will search the symbol table of the symbol table it points to. This can happen many times, so lexical scoping works for any number of nested functions. Later, in the machine code generation section, symbol tables are used to store activation records – a record holding information required for the call and execution of a given procedure.

The symbol table is populated with names as the function to traverse the AST comes across them in the tree. As mentioned, each new declaration of a function creates a new symbol table, and every variable defined within that function is placed within this symbol table. The relative memory locations of the names can be found by performing a count – each name corresponds to a given amount of memory, which can be stored in an ‘offset’. This offset variable represents the offset of each variable from the memory location storing the start of the procedure. This can then be used later to find the correct location to store the variable, and the correct location of where to find it. The final count of offset (greatest offset) also allows for the size of memory that a procedure needs to be calculated quite easily. This allows for more efficient memory allocation, in the later stages of compilation.

Adding to the point of optimization, basic blocks are the fundamental units used for optimization. These are a sequence of TAC statements that have no jump in control. When the flow of execution of a program changes, certain registers need to be saved, as otherwise they may be overwritten and their contents lost; to allow for optimization, we need to ensure that our optimizations do not change anything that could affect code elsewhere in the program, as the optimized code must be equivalent to the unoptimized code in semantic behaviour. The easiest way to ensure this is to optimize the program in chunks of basic blocks, where each basic block is some TAC sequence that starts with a leader, and ends with either the end of a program, or some jump to another basic block. In my compiler, I implemented the basic block to allow for optimization, but did not have time to actually implement the optimization.

Machine code

As mentioned, three-address-code is converted into assembly code. A simple way to do this is by using template-generation. For each TAC statement, we can generate some assembly code that is equivalent to the TAC statement. For example, a TAC statement that represents an addition operation between two operands requires the two operands to be loaded into registers, then added and stored in some destination register. This is the general form of solving the problem of creating machine code for TAC with addition operators.

However, template-based generation is quite inefficient, as names may already be in registers, and they do not need to be loaded again. In addition, loading names into registers that contain other names may cause some to be overwritten. To solve this problem, an address descriptor and register descriptor can be used, to describe the locations of each name, and to describe the contents of each register respectively. The address descriptor was stored in the symbol table, as the symbol table is used for name lookup, it is suitable for holding where each name is in memory.

Following from the address descriptor and register descriptor, we get the idea that we can select some more ‘optimal’ register to use for a given TAC statement of form x = y op z. The function Getreg, as described by (Aho, Sethi, Ullman), is a function that select a register to store the results of the described TAC statement. It looks at what each register contains, where the current operands are located, and the next uses of the names. A next use of a name is where the value of that name is used again within the same basic block, where there is no alteration of the value that name holds between the first TAC statement and the following TAC statement. By looking at next uses, we can determine whether it is important to keep a value in a register, as registers are limited but fast, so we want to keep useful values in there as long as possible. Since memory is slow, values should be kept in registers, and should ideally only be removed if they are no longer needed.

To implement machine code for my compiler, I used template-based generation. I implemented the getreg function with the address descriptor and register descriptor but did not have time to make use of them, although they provide a good starting point for improvement, along with basic-block optimization.

When a function call occurs, the caller and callee must both perform actions to save and store important registers; since the caller and callee operate in black-box environments, they must work together to save what is important to them, as neither know exactly what is important for the other. An activation record is some record that holds information required for the execution of a procedure call. I implemented this in my symbol table, and it holds information about: the location of the memory address for the start of the procedure, a pointer to the caller of the procedure, and a link to the most recent activation record for any enclosing procedures – this is important for realising lexical scope. Again, the symbol table holds information regarding the different names in a function, thus it is suitable for also storing the activation records. A method to access the non-local variables is described by (Aho, Sethi, Ullman), where you use access-links (the link to the most recent AR) and some information about the nested depth of an activation, to follow the access-links until you get to the correct function where the non-local is defined.

During compile-time, we know the size of memory that a single function might use by looking at the size of the names within it. However, due to function calls, we need to dynamically allocate memory, using a heap, rather than some static location. To implement this heap, I used a syscall. I then moved the memory address to be stored in some register, with the AR’s being updated as required. One possible implementation I considered was to allocate some new memory each time a function was called, where the size of the memory was equivalent to the number of nested functions multiplied by the size of each memory address. Then each block in this new memory address can hold the address of the first memory address of each function that is required to be maintained in memory and can be accessed by moving down this chunk of memory until you get to the desired address. However, I did not have time to implement this.

Section 2 – Description of semantics

Fundamentals of translation

In both the interpreter and the compiler, we need to translate some node in an abstract syntax tree to another symbol or representation. These translations can be formally described using context-free grammars. In context-free grammars, we can describe the semantics of a programming language, by having a set of productions (rules) that dictate how to deal with each node of the tree (symbol). By inspecting what each node can give as output, we can write the appropriate code to either interpret it or translate it to TAC (which can be represented as a CFG for machine code generation).

Using CFG’s, we get the idea of syntax-directed translations, which are a generalization of a CFG (Aho, Sethi, Ullman). Syntax-directed translations allow for judicious design of a translation scheme, to ensure that the compiler’s functionality is sufficient.

Interpreter

As discussed before, the interpreter is made using a recursive function that evaluates the nodes in an AST using switch-case. There is a pre-interpreter phase (for the global environment) and the actual interpreter, as mentioned before.

Here are the semantics of each type of node that my interpreter handles.

**; (SEQUENCE)** – Represents a sequence of things to be interpreted.

Result: Interpret left tree. Then interpret right tree.

**D (FUNCTIONDEF) –** represents a function

Result: Make the environment for the function.

**D (FUNCTIONDEC) –** declaration of a function

Result: Interpret right tree (function).

**F(FUNCTIONPARAM) –** sub-tree of parameters of a function

Result: Interpret right tree.

**APPLY –** applying a function

Result: Call a function to get the arguments. Then call the function to perform the lexical call of the function.

**RETURN** – return some value

Result: Return the interpreted value of the left tree.

**PLUS –** Expression with addition of operands

Result: Call function to interpret the operands and add them.

**MINUS –** Expression with subtraction

Result: call function to interpret operands and subtract them.

**MULTIPLY –** Expression with multiplication

Result: Call function to interpret operands and multiply

**DIVIDE –** Expression with division

Result: Call function to interpret operands and divide.

**EQUALS –** Binding of name

Result: Call function to add the binding to the name of the interpreted value of the right tree.

**IF –** If-then or if-then-else statement.

Result: Interpret condition, check if ELSE statement exists, then perform appropriate actions, either interpreting the left or right tree.

**LESSTHAN/GREATERTHAN/GE\_OP/LE\_OP/NE\_OP/EQ\_OP**– relative operations

Result: Interpret left and right tree, then compare values. Set Boolean to either true or false and return.

**LEAF –** leaf node of the AST

Result: Check if leaf is an identifier, if it is, find the binding for the name. Otherwise return the value of the leaf.

Above is a brief description of what occurs for each node seen in the AST in the interpreter. As mentioned, there is also a phase before the interpretation of the program, where the global environment is created. In this function, a single pass is done through the AST, and any declarations of functions and/or variables are handled and placed into the global environment. For functions, closures are created, where the environment is the environment for the function, and the code in the closure is the node following the function declaration node; in this pass, the nodes within the functions are not evaluated, as that is reserved for the main interpretation phase. Variables are placed within the global environment. After this phase finishes, the global environment is returned from this function and the main function is found. These two are then used to call the recursive interpretating function described before.

Function calls are done using a lexical call method; a new environment is created for the called function, and the arguments and formal parameters are bound together. The frame is attached to the current caller’s environment, so that non-local variables may be accessed, as is required in lexical scope.

The arithmetic and comparison features are done in the same way; any non-integer or terminal operand is interpreted, until a terminal is reached. Then, these are returned, so that any of the operations that required them can be completed, until the first operation to start this chain is completed.

The sequence node represents multiple things happening one after the other. The implementation of this requires the left and right subtree to be interpreted in turn, which is what happens in the interpreter.

A leaf node may contain either an identifier (variable), or an integer value that should be returned. The node is checked, and what happens determines on whether the leaf node is an identifier or value. If it is an identifier, then the identifier is found and taken from the environment. Then, its correct value can be returned. If it is some integer value, then that value is returned.

Compiler

As outlined before, the compiler is divided into an intermediate code generation phase and a compilation phase. The compilation phase is essentially an interpreter for the TAC, which is the intermediate representation used.

Syntax of TAC

The TAC used is a quadruple. This is of the form x = y op z. There are 4 parts to the TAC, hence the name quadruple. Although not every TAC statement is required to have all 4 parts, I left the syntax of the TAC like that for every TAC statement, to speed up implementation, but a more optimal solution would be to use the union keyword in C to make a nested struct for TAC.

TAC can have the following operators, represented by op:

* **Plus** : addition
* **Minus** : subtraction
* **Multiply**: multiplication
* **Divide** : division
* **Ge**\_**op** : >=
* **Le**\_**op** : <=
* **Ne**\_**op** : !=
* **Eq**\_**op** : ==
* **Less**\_**than** : <
* **Greater**\_**than** : >
* **None**: load integer
* **Assignment**: assign value to variable
* **Proc**: marks start of procedure
* **Endproc**: marks end of procedure
* **Label**: label for jump
* **Go**\_**to** : jump to label
* **If**\_**then** : if statement

The three other elements are represented by TOKEN\* structs.

They have a type, lexeme (string/name of token), an integer value, and a pointer to the symbol table that should be searched if they are required.

Semantics of TAC - (translation to MMC)

The interpretation of TAC is done by using a switch-case for op:

**PLUS:** Call function to generate MC for plus. This involves loading the values of the operands into registers and storing the result of the addition in another.

**MINUS:** Call function to generate MC for minus. This involves loading the values of the operands into registers and storing the result of the subtraction in another.

**MULTIPLY:** Call function to generate MC for multiplication. This involves loading the values of the operands into registers and multiplying. Results are then stored in the floating-point registers, which are moved to some other registers.

**DIVIDE:** Call function to generate MC for division. This involves loading the values of the operands into registers and storing the result of the division in another. Results are then stored in the floating-point registers, which are moved to some other registers.

**GE\_OP:** Call function to generate MC for GE\_OP. This involves loading the values of the operands into registers and storing the result of the comparison in another.

**LE\_OP:** Call function to generate MC for LE\_OP. This involves loading the values of the operands into registers and storing the result of the comparison in another.

**EQ\_OP:** Call function to generate MC for EQ\_OP. This involves loading the values of the operands into registers and storing the result of the comparison in another.

**NE\_OP:** Call function to generate MC for NE\_OP. This involves loading the values of the operands into registers and storing the result of the comparison in another.

**LESS\_THAN:** Call function to generate MC for LESS\_THAN. This involves loading the values of the operands into registers and storing the result of the comparison in another.

**GREATER\_THAN:** Call function to generate MC for GREATER\_THAN. This involves loading the values of the operands into registers and storing the result of the comparison in another.

**NONE:** Call function to generate MC for NONE. This involves loading the immediate value into the register designated by the TAC.

**ASSIGNMENT:** Call function to assign a value to a variable and place it in the correct location in memory.

**PROC:** Call function for procedure start. Save and store old frame. Create new activation record for the procedure.

**ENDPROC:** Restore the necessary registers.

**LABEL:** makes a label for jumps to jump to

**GO\_TO:** makes a jump to the label given by destination

**IF\_THEN:** checks the condition given by TAC. If true, then it jumps to the label given by destination.

Above is a brief description of what the machine code that is generated must do to satisfy the semantic requirements of each type of TAC operator. I will categorise the TAC and describe the process of each more thoroughly below.

Arithmetic: for any arithmetic TAC, I followed a template of loading the operands, and performing and storing the result in a register. This is a very inefficient way to do this, as it may require more access to memory than needed, for example, if one operand was already loaded in a register. Also, simple optimizations which could have been performed on the TAC have not been done, so there may be instructions which are not necessary.

Conditionals: for conditional if-then and if-then-else statements, I decided that having only one if-then operator was enough for the TAC. This is because I already have the LABEL and GO\_TO operators, which allow me to produce TAC to represent the ELSE part, without having to have a separate operator for it. This therefore saved time in writing the code for the translation from TAC to assembly, as I didn’t need to have an entirely separate section to deal with ELSE operators. During TAC generation, if there is an IF node, then a go\_to jump TAC is generated after the if\_then TAC. This jump would cause the program to jump past the code that would be executed if the if\_then condition was true. Should an ELSE node be present, then all that happens is that the code for the ELSE portion of the if\_then\_else is translated into TAC statements and placed after the if\_then statement, and the go\_to TAC is generated at the end of this section of TAC, such that it jumps past the code for the IF section. In doing so, if a condition is true, then the if\_then will jump to its designated label, hence skipping any ELSE code if it is there, and if a condition is false, then if there is an ELSE part, it will be performed, otherwise it will just jump past the TAC representing the IF-statement

Functions: when a call is made, assembly code is generated to dynamically allocate memory, using the symbol table to calculate the size of memory required. Then, the frame is stored in this new memory and moved to another register. The frame pointer is then updated, memory is created for the new frame, and a new activation record is created, to point to the register in which the previous frame is held. The division of work between the caller and callee is due to the functions working in ‘black-box’ environments, where neither knows what registers are important to the other; neither do they know how many registers they might use within a function, so the reservation of registers would be inefficient, as one function may reserve more than they need. At the end of the execution of a function, a jump is made back to the caller, so that it can resume execution.

Section 3 – Example Test Cases

Interpreter

**Variables and Arithmetic**

Text

Description automatically generated

Text

Description automatically generated**Global function calls**

Text

Description automatically generated

**Nested functions and Lexical scoping**

**Global variables**

**Text

Description automatically generated**

**Conditionals**

Text

Description automatically generated**Text

Description automatically generated**

Compiler

**Three Address Code**

Variable assignment

**Text

Description automatically generated**

Arithmetic with variables

Text

Description automatically generated

Text

Description automatically generatedFunctions

Conditionals

*(If\_then)*

Text

Description automatically generated

*(if\_then\_else)*

Text

Description automatically generated

Basic blocks

Text

Description automatically generated

**MIPS**

Variable assignment + Function declaration

Text

Description automatically generated

Arithmetic

Text

Description automatically generated

Section 4 – Critical Analysis and Improvements

Interpreter

The interpreter could be improved by making it more robust. Due to time constraints, extensive testing was not done on the interpreter, which means that it may not work for all the cases that it should. The implementation of the interpreter was basic, but the functionality I implemented and tested for seemed to work. Perhaps more planning could have been done to allow for easier implementation and less time spent debugging logical errors. During the implementation of the interpreter. I read about the metacircular evaluator and the semantic directed translation which gave me an idea of how to implement it; ideas about recursively evaluating and applying were useful in helping my understanding. It was also quite applicable as C’s compiler itself was written in C. However, more material could have been covered to allow for a more informed implementation, as at times I did not know what to do.

Moreover, the interpreter is not a complete solution; many features, such as functional arguments do not work, although it was specified in the coursework specification. Another attempt at implementing the interpreter would involve more careful planning as well as additional background reading. Something to be discussed in more detail later is the usage of regression testing in the implementation of both the interpreter and compiler.

A lack of extensive features in the interpreter has made the testing of the compiler more difficult, as program output cannot be compared if features were not implemented in the interpreter, or if the interpreter was unreliable, then the comparisons would be unreliable – although either way, a difference in output would suggest error.

Furthermore, code quality was not excellent. There were many pieces of repeated code, such as the initial pass through the AST, which could have been made into one function using some Boolean to denote whether the pass is the initial pass or the actual interpreting phase. Comments could also have been more useful to aid debugging and implementation.

Compiler

Many things can be improved with the compiler.

Starting with TAC, basic block optimization could be performed to optimize the TAC in the intermediate code generation phase before the latter machine code generation phase. The code to produce basic blocks from a sequence of TAC has already been implemented, so the next step would be to implement the functions to optimize these basic blocks and refactor the machine code generator to take sequences of basic blocks rather than sequences of TAC. This peephole optimization process would be implemented iteratively, as one pass of the optimisation process may reveal new lines of TAC to be optimized; the iterative process would continue until the code is no longer changed. As mentioned before, a more optimal struct could also have been designed to represent the TAC. However, I think the types of TAC generated was suitable, as it was high-level enough to leave the machine-level details to the specific machine code generator, which would allow for more optimisation to be done, if it was done.

Furthermore, next uses were also computed. The use of the function getreg defined by (Aho, Sethi, Ullman) could further improve the process of code generation and use a more formal algorithm for deciding registers to use for operations. This would allow for a more efficient compiler and would also be more likely to produce correct code, as the algorithm has been used by many, and peer reviewed. As mentioned before, a more optimal struct could have been designed to represent the TAC. Next uses were stored in the symbol table, which was implemented as a hash table, where each entry of the hash table is a linked list of the symbol\_table\_entry struct. This allowed for fast searching for names in the symbol table, but perhaps memory efficiency could have been improved by first calculating the size that a symbol table needs to be, by first doing a count of some variables within a function for example.

In the machine code generation section, template-based generation was used. Different instructions have different costs for the computer; some require more cycles than others. Instruction selection is a technique that uses heuristics and contexts to choose some more-optimal instruction for a given TAC. This could have been implemented to improve the compiled code’s efficiency. Furthermore, some instructions may not even be required; this could be improved in another optimization phase after the TAC optimization phase. Peephole optimization could also be performed here. Other optimisations involve using data flow analysis to look at how to optimise control flow in a program. Furthermore, another area of improvement for machine code generation is evaluation order of arithmetic. Different evaluation orders may prove to be more efficient, as unnecessary statements can be removed. Although it is unlikely that the most optimal solution for this would be found, using some basic heuristic could be a potential step in improving how instructions are selected and dealt with.

Related to instruction cost is memory access and register allocation. Registers are very fast relative to memory but are limited in quantity. Accessing memory is slow, therefore it is important to keep data in registers for as long as possible. According to (Aho, Sethi, Ullman), data should only ever be removed from a register if it is no longer needed, or if it is necessary to store it in memory. As mentioned before, a register descriptor and address descriptor will help with this; since they have already been implemented, the next step would be to use them in the generation of machine code instructions.

Something important but not yet mentioned is the use of temporaries. Temporaries are temporary variables that were used in TAC generation. They are also stored in the symbol table. In my implementation, I assumed that I had unlimited temporaries for simplicity, however a better solution would be to have some form of register allocation strategy. One such strategy has already been introduced in the form of the ‘getreg’ function described by (Aho, Sethi, Ullman). Other ways of improving usage of temporaries relates to its storage. “If two temporaries are not live simultaneously, then they can be packed into the same location” – (Aho, Sethi, Ullman). Improving the storage of temporary names could save memory.

Testing and implementation

The ideal method of implementing the compiler and interpreter would be to implement the interpreter first so that it can be used in testing the compiler. If the output of some code varies, then a bug is evident, and it can be fixed. In later stages, where optimizations are performed, then regression testing can be performed using older, pre-optimized versions of the compiler/interpreter. In doing so you can confirm that the code produced by the optimized version is equivalent to the slower, but ‘correct’ code.

Ultimately correctness is the most important aspect of a compiler, which makes testing paramount. To do this more effectively, a script to run a series of tests should have been made and used throughout the implementation of the coursework, rather than hand-typing test-cases.

Regarding implementation, the solution given is not complete. Many features are missing, and implementing these missing features would be the logical first step to take when improving the work.