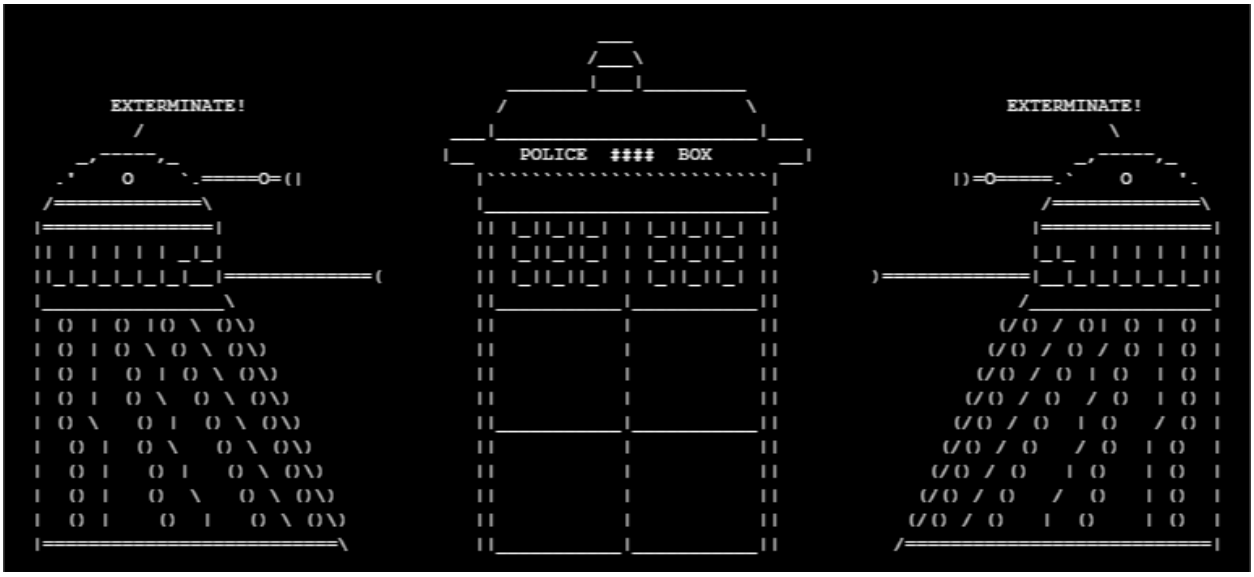
Horia “Ryder” Stancescu
hstances - 0721385

The Final Report

The Ramblings of a Doctor



The Process

In this report I will be detailing the order and rationale of events being performed. I feel I should start with a disclaimer here stating that the code I have created although functional is by no means optimal as I was creating it as functionality was needed, there are areas that are highly coupled and un-modular which could use restructuring such as the assembly generation however as mentioned the code does work, and fulfills almost all if not all the criteria specified by the specification. That said this section will be broken down into the areas associated with the source code files, starting with the Parser which generates the Abstract Syntax Tree, leading into the Symbol Table, followed by Abstract syntax tree, the Intermediate Representation generated during the tree traversal, finally followed by the Assembly code generated from the Intermediate code traversal.

The Parser

In order to do a conversion from Cflat to Assembly, one must first understand and process the contents of a file before any type of conversion can be made. The Cflat program has a series of rules about declarations, and Typdefs. Mainly Typedefs must occur before any declaration instructions, and declarations may only occur before functions calls and before statements within function code blocks. This is important in laying out grammatical rules for understanding what a valid Cflat program looks like, for example if a file contains Typedefs after global declarations the file is invalid, and we can report errors at this stage of the process without needing to continue. That said the parser is responsible for populating the Abstract Syntax tree and Symbol Table, for my program I have chosen to only populate the abstract syntax tree with statements and expressions. Statements and expressions are anything which can either manipulate data, or modify control flow, thereby if statements, loops, function calls, are all treated as expressions along with standard arithmetic, relational, and assignment expressions. This means that declarations and Typedefs will not get stored in the tree, the reason for this is when it comes to doing Type Checking so long as a Symbol Table exists which can be referenced during Tree Traversal, it does not matter whether this data is stored in the ABS tree or not so long as a link is available when needed. This gives the benefit of not cluttering the Abstract Tree with unneeded statements that do not need to be analyzed and interpreted when creating IR code. That said it also does not make sense to parse a file twice that is why

during the Parsing process the code appends values into the symbol table as grammatical rules get verified. It is at this point I feel it is important to mention that Yacc is a bottom up parser, meaning the rules get hit in reverse order, therefore I couldn’t simply say when a declaration occurs add that variable to my symbol table. This becomes evident when given a declaration with multiple variables such as “int x,y,z”, the declaration rule won’t occur until after the x,y,z rules get interpreted meaning I can’t simply append values as they are read. The fix I have come up with is to add these variables as they are read with a “Standby” type, then when the declaration rule gets hit, overwrite all variables with the “standby” type with the type requested. So for example x,y, and z, will be stored as “standby”, and then once the “int” gets read they get converted to int. This same approach is used for identifying what scope a variable belongs in, but instead of having a standby variable they get appended to a “Tmp” scope, and once the function rule is read, all declarations get relocated to the function’s scope.

Symbol Table

Now that we have a brief understanding of how variables are added to the symbol table, we can move onto how types are linked. Typedefs can be tricky as there are no restrictions as to how many times you can Typedef a variable, for example I can Typedef an “int” to “myInt” and then I can typedef “myInt” to “myNewInt” and it must be considered the basic type “int”. Because of this two tables are used in the SymbolTable.c file, one for variables and scope, the other is for Typedefs. The way I went about linking any typedef to a basic type, is every time a Typedef is requested I link the identifier with the keyword the user wants typedef’d, this occurs regardless what the keyword is. When it comes to type checking, what occurs is whenever a type is requested, the reference table will keep checking the links of the types until either a basic type is hit, or there is no further link available. Meaning in the example provided the keyword “myNewInt” gets converted into “int”, because it was typdeffed from “myInt” which represents an “int”, this allows for near endless Typedeffing in theory. I say in theory because the hash table I use (reference table) for Typedefs does not grow, it has a fixed size and eventually there will be no room to add additional Typedefs. In regards to scope, the way it works is there is a separate hash table used which has linked lists as the entries. The first entry in the linked list represents the functions scope, and all the other entries are the variables. Because variables can belong to a struct and be located inside the scope of a function or

global space, a separate variable is used within each entry to identify whether or not a variable belongs to a struct inside each scope, making the process simpler. For array handling an integer is used to determine the size of the variable, a value of zero represents a single item, whereby anything greater and the variable will be understood as an array which is useful when Intermediate code is needed to perform the necessary arithmetic to access the required index. One of the limitations I have introduced is structs can no longer be typedefed. For checkpoint one and two, structs were capable of being Typedefed, however the ambiguity which arose with nested structs being partially Typedeffed introduced more problems than I was willing solve so I took the shortcut of removing that functionality.

Abstract Syntax Tree

Now that the symbol table side of the parsing is understood let’s get into the ABS Tree. The abstract syntax tree is generated as one would imagine, a simple tree with nodes branching outward for each instruction. However I have chosen to keep my Abstract syntax tree a Binary tree, meaning every node can have max two branches a left and a right. The main reason I chose this was it allowed me to re-use the function calls already placed with only a requirement for “container” nodes. Container nodes are what I use to branch off separate expressions and function calls. This allowed me to retain all functionality and even perform simple tree traversal as opposed to adding extra clauses for expression nodes where multiple nodes are used. However I only speculate it’s easier because I had an easy time coding the abstract tree traversal, I have not coded the other method using n-nodes so I can’t say with certainty which method is easiest. That said utilizing the binary method allowed me to hack certain clauses during tree traversal such as checking a single depth when checking for parameters or function calls which may not work when using a multi node tree as there are separate nodes. During parsing it is also important to note that for loops do not exist in the ABS tree, instead the “while loop” representation of them are created, and similarly there exists no “if/else” in the abs tree. Instead the “else” rule gets parsed as the “**not”** of its parent if node, so a double if expression is generated in the tree. This does not affect the flow of the program but it simplifies the logic for me once creating IR code, as once I create code for parsing if, and while I’m done.

Intermediate Representation

As I alluded to above there are areas of the program which have tight coupling, this section is one of them. When it came to generating the assembly code it was far easier for me to use variables which already exist, and since IR code must generate temp variables why not store these on the stack for each function call. It simplifies the process and will make the assembly code function very similar if not exactly to the IR code once it’s fully created, the problem is temp variables have no type as they do not exist during the Parsing stage. The simple remedy is to add these temp variables to the symbol table as they are generated. The temp variables are generated based on the current nodes index (in the ABS Tree), meaning if I’m currently at node x in the abstract syntax tree, the immediate left node is x+1, and once that branch is done the immediate right node’s index and therefore temp variable is “x+ left branch nodes”. Incase it was not clear up until now the Symbol Table is always live until termination and has the 1st index of each entry serve as the function scope. That is why in order to check if a variable is in scope all that is required is to traverse the current scope name the value is in, then global scope if the variable exists in either of the two we continue if not error. This same reason is why adding variables (in this case the temp variables) is simple, we only need to perform type checking on the variables, then store temp in the same scope as the type we get on the left hand side. This works because if a type error occurs although the information is wrong, the assembly generation will never be executed because errors were detected during this type checking stage. When Intermediate code is generated there are a few fields which are used for each entry: scope, label, op, leftvalue, rightvalue, result, isIf, state_addressed, and next/prev. The scope variable is just for bookkeeping to always have a reference as to the scope the variable is invoked from, the left, right, result, and op values are just to maintain the quadruple form as demonstrated in the slides. The isIf is used as a flag to determine whether or not a jump clause is needed, since for loops are converted to while loops, during the IR conversion, while loops are then converted to if’s. So when a while loop is created two if expressions are needed the original to decide whether to skip the while body, and an appended one to decide whether the loop should re-iterate or break out. The label is self-explanatory if a jump instruction is ever executed it must have a corresponding label to jump to, therefore whenever that field is not null it will be appended to the IR code to then be applied to the assembly. State_Addressed is another flag to determine whether or not a value is currently holding a value or an

address. This became quite important as whenever a temp variable is referencing a variable from the provided Cflat program, it always starts with the instruction “tempX = &b” where “b” is a variable in the program. This became important as when arithmetic events occurred such as the following code:

- int x;
- x=x+5;
    - temp1 = &x
    - temp2 = &x
    - temp3 = 5
    - temp4 = temp2 + temp3
    - temp1 *= temp4

If I didn’t have a way to identify between a stored address and value, I would be saying the address of x is equal to the address of x+5, and when requested I would never be able to store any value at any address. However it also meant I have to be extra careful with structs and arrays, this is rectified by changing the operators used to identify between address arithmetic and value arithmetic. Since address arithmetic only occurs with addition and multiplication special characters are used: “*_” and “+_”, this is just so when creating assembly, the algorithm can have an easier time determining whether an address is being manipulated or a value. As far as how the intermediate code is actually stored, it’s a simple double linked list which should be evident by the “prev/next” pointers.

Assembly

Finally the assembly, as far as assembly is concerned all that is required is to generate the equivalent assembly instruction as is represented by the IR code. Since temp variables always start by referencing values, whenever a new temp variable is used, the values it contains are whatever is stored at the address it’s referencing. Because of this and the fact that each temp is stored in the symbol table, each function simply requires a stack to be allocated for all variables including the temps and then perform the IR code taking note of the “State_Addressed” flag. The assembly code was by far the trickiest to finalize, this was mostly due to having to

trace stacks and ensure all mathematical operations in regards to addresses are correct along with ensuring floating point arithmetic utilizes the float registers. Aside from that the programs are created almost exactly from the IR code and there is not much more to say on this, whenever a size of a variable is needed for stack offsets, the symbol table is requested for either a variable size or the offset and that value is used. Different data is requested depending on the variable accessed, for example global variables require no stack offsets (instead a label), variables inside a stack require the stack offset + the variables stack offset, arrays require the offset of the array multiplied by the index and so on.

Conclusion:

Hopefully I have made clear the process I used in creating each component of this compiler. When it comes to error handling, messages are printed per section, and if a section contains errors the compiler will not continue on to the next step. This prevents buggy assembly from being written if a file contains a buggy symbol table or failed to pass the syntax analysis. The abstract tree and symbol table are created in unison from the initial parsing, the intermediate code is generated as the abstract tree is traversed and it populates the symbol table with temp values to be used during assembly. Finally the Assembly is generated via traversing the list of IR instructions. Unfortunately I did not have any source language issues and although the code may seem obtuse at times, it works as expected and I will be using the demo time to do a more thorough explanation of the code as writing a large report detailing the process of the compiler is the best I can do given that most of the code is generated from the concepts explained in class and on slides that I do not wish to repeat. As far as my testing goes the program performs well on all aspects required as shown in 1.cb, however it fails to do type checking on function parameters. I have not implemented type checking on parameters therefore my compiler will process code that passes more or less parameters to a function regardless what it requires, meaning you could overwrite variable initializations by passing more arguments than a function expects (is it a bug or a feature?). Type checking of values however works as intended, and the final restriction is my compiler can’t pass literals as parameters. Meaning if you wish to pass “5” to a function “foo”, you must store the value “5” in a variable, and pass that variable to foo. On that same note all variables are passed by value including structs, in 1.cb I have an example of a

struct having its contents modified in one function, passed to another and printing that variable as a

demonstration to show the values are all passed as opposed to just the struct’s starting address.