C COMPILER

By Patrick Engelbert

Introduction

This document is a brief explanation of the working of a compiler for C89 to ARM assembly code. IT is able to process a small part of C including straight line code, functions, structs and loops. The compiler produces useful error messages and can be expanded to allow for more features present in C.

Compiler Structure

The compiler is divided into four sections, the lexer, the parser, the Abstract Syntax Tree (AST) checks and the translator. These sections each take in the output of the previous section and transform it to allow the next section to use it, ending in assembly code.

Lexer

The lexer uses regular expressions to convert the input file into a stream of tokens that the parser can then link together to form a coherent structure of the program. The tokens are very distinct as many of them have different uses depending on the context that it is found which can be better handled by the parser. However the lexer does help with custom type checking where every ‘id’ like token is first checked against all know types to determine whether it is type or a an ‘id’, be it a variable or function name. This is only possible as the parser calls for the lexer to get the next token and then tries to parse it. As custom types are defined before they are used, they can be found by the parser and added to a data structure of custom types, allowing the lexer to immediately use these names as a ‘type’ token rather than an ‘id’ token. This method also reduced the processing required by the parser in relation to custom types as the fact that it is a type will have already been established. For the lexer, nearly all tokens are based on standard C keywords.

Parser

The second part of the compiler is the parser. This section uses production rules to match a pattern of tokens to mean a specific C construct. The parser then builds these rules together in a set of nodes to produce an AST which can then be interpreted by later components of the parser. The AST has two main node types: the parserNode, a branching node used for anything from expression to a list of statements and control flow, and the variableNode, which contains information on an individual variable declaration or use. To allow them to be built together easily, all of the nodes used in the AST inherit from a base node which contains some basic information such as: node type, node value and node id. These values allow the compiler to know the class of the node for type casting, information on the action being performed in the node and the name that may have been assigned to that node.

In general the AST is built in a top-down manner with new nodes being added at the bottom of the far right sub tree. This is done to allow for accurate scope checking later on in the compiler. To do this, whenever a new node statement is to be added to the current tree, a temporary pointer traverses to the down the right sub tree until it reaches the end of the ‘NULL’ (non-relational) nodes, at which point it has found the end of the list of statements. It will then create a new ‘NULL’ node there, copying the current node into one of its branches. This is done for statements or lists of ‘id’

type tokens to allow for scoping to work correctly. All other nodes are built bottom-up by building a node and passing a pointer to it up to the parent production rule.

AST Checks

The third part is a set of functions to traverse the AST and check that everything that has been parsed is valid to the C89 standard. This involves three traverses of the AST, the first to trim out useless nodes to reduce calculation time, the second to check that all variables and functions have been declared before they are used and the third to check that all types are correct for expressions and functions.

For every 'NULL' node, new map is created and pushed onto a stack of maps. This map is the scope of that node while the maps on the stack are all the preceding scopes, going all of the way to the global scope. A variable is deemed in scope and thus useable if it is in any of these maps. When a new variable is declared, first all scopes are checked to see if it has been declared before. For example if a variable x has been declared in the global scope and then in a function, the one in the function will be deemed invalid as it is declaring the variable from the global scope again. Once the sub-trees of this node has been checked, the map is popped from the stack and thus all of the variables that were declared in that scope are now out of scope. However this relies on declaration for variables being on the left side of a 'NULL', as otherwise it will never get added to the scope. For structs and functions, there is only one scope which is global. Once they have been found, they are added to their scopes and can be accessed at any time.

Once the variable declaration checks have been completed, the types of expressions must be compared. To do this, for every node the types of the two sub trees are determined. Then if they are equal this type is assigned to the current node and so on. For variable nodes, the type will have been previously determined by the variable checks and so they will be the base point upon which the checks are build. Explicit casts force the node to take on a different type to its actual one and so allows wildly different types to be compared. Implicit casts occur when both of the types being compared are either 'int', 'float' or 'double' at which point the type of the left operand is used. The same occurs when a pointer is assigned to an 'int' which forces the output to be an 'int' but will throw a warning. For certain nodes the checks are not produced as they do not perform an expression on the two sub trees and they themselves have either a fixed or no output value.

Translator

The last part of the compiler is the translator. This traverses the tree and converts each node into its appropriate assembly instruction. Combined with this is the register manager which keeps track of which variable uses what register and/or where that variable is currently stored, be it on the stack or in actual memory. The logic behind the translator is very simple due to C being a low level of the high level programming languages. This means that each statement in C tends to directly convert into one or a few assembly instructions that do not change depending to context.

The main exception to this are function definitions and function calls. To call a function, all variables currently residing in registers 4-11 must be moved to the stack. Then the link register must also be moved to the stack. After that, the first four parameters to be sent to the function are stored in register 0-3 as these are used for inter function communication. The remaining parameters are then pushed onto the stack. Once the function has returned, the link register and registers 4-11 can be restored from the stack and the return value retrieved from r0. To make this work, the called function has to retrieve all of its parameters from the stack so that the link register is at the top of the stack, ready for the other function to retrieve it. However this method does cause problems with the register manager as the variables change names between functions, forcing the use of alias names to be pushed and popped from a separate stack in the register manager to account for this.

Grammar

The grammar of C that is used revolves around left expressions and right expressions. A left expression is something that can be assigned to but is not an expression by itself. For example “int a” would be counted as a left expression. However as “int a” can also be used standalone, it is also added as a statement. A right expression is an expression that can be assigned to something. It is also counted as a standalone statement if need be. These include unary and binary operations as well as constants and variable uses.

These are then built into blocks of statements that can either be in a set of “{}” or on their own depending on the required use. These lists of statements can then make up the body of a different statement such as a loop or function or be used on its own. The highest level of the grammar is the program block which is anything that be written int eh global scope such as a variable declaration, strict definition or function. It can also include straight line code as long as they are in “{}”. A program contains 1 or more of these blocks in any order.

Design Decisions

There are several main design decisions that were made. The first one was to try and implement each stage and then go onto the next. This breadth first approach rather than the depth first was chosen as it allowed each section to be made relatively independently from each other and more importantly meant that each section was designed to allow for many general C type information to be encoded with minimal changes. If a depth first approach were to be taken, the likelihood was that once a small amount of compilation was possible, adding any new functionality into it would require the rewriting of the whole compiler, causing more delays than was manageable. The downside of using a breadth first approach was that too much time was spent on the first parts, especially the grammar, meaning that less and less was implemented at each section due to time constraints.

The second major design decision was to bypass the Intermediate Language (IL) in favour of going straight to the assembly code. This was possible due to the use of a dynamic register allocation system. While creating the system proved to be a challenge, mainly as it was made before the AST had been created and as such could be tested only with difficulty. It was also not designed to be used with functions, causing a large rewrite to be required as they were being implemented and the register allocation losing track of the variables being stored. However the benefit that was gained was that a second lexer and parser from the IL to assembly was not required, reducing the amount of work that had to be put in overall in this section.

Register Management

The register manager runs on Least Recently Used (LRU) algorithm. This algorithm allocates a variable the register that was used. Last. However as that variable may still be in use, every time it tries to allocate a filled register, the value in that register is stored in a preset location unique to that variable. This means that when that variable is called again, the location of it in memory is known and it can be loaded back in. This method allows for more operations to occur within the registers, increasing the speed at which the final code will run.

The register manager keeps track of all currently active variables through the use of two lists and an array. The array represents all of the registers 0 -11 (12 is used for the base of the memory addressing) while the lists represent the stack and memory. A currently active variable can exist in any of these locations, and can be retrieved from the memory or registers at will. The stack requires explicit push and pop instructions to occur before the values can be retrieved.

The method used currently has two major bugs/flaws that need to be addressed. The first is that variables are not scoped in the register manager/translator. This means that if there are two variables that have the same name, they will be counted as the same by the register manager. This may not be an issue seeing as they will never interact due to being in different scopes but that is not guaranteed, especially with pointers. The second flaw is that temporary variables are too frequently assigned. Every node gains its own temporary location and even if this is only used for one instruction, it does mean that that location is blocked, perhaps causing multiple loads and stores to be executed to free up the space. For example, currently the C instruction “int a = 1;” will translate to “MOV R0 #1 MOV R1 R0” as R0 is used for the temporary constant 1 and R1 is used as the variable a. Then R3 is used for the temporary variable relating to the whole expression, even though it cannot output a value.