**Compiler Report**

**Introduction**

We were set an assignment to create a language processor capable of compiling Pascal code into runnable ARM assembly language. This was done in C++ using the tools Lex and Bison to read input and parse the input files. These were then processed using my code to generate the output .s file suitable for execution. The project was split into three main sections: basic control flow and data structures; loops, procedures and functions; advanced data types including Arrays and Pointers. The output code generated was to be runnable and was tested using the ARM debugger suite to ensure correctness of outputted code.

**Features**

1. How to Build and Run the Compiler

2. Declaring Variables and Constants

3. Variable Assignment

4. Arithmetic Operations

5. Conditional Statements

6. Printing Results

7. Loops

8. Functions

9. Arrays and Pointers

10. Memory Management

11. Error Detection and Correction

12. Compiler Errors

**1. How to Build and Run the Compiler**

To compile and run the compiler, enter the following into the terminal:

make

./CW1 <pascalfile.pas>

For example,

make

./CW1 sample1.pas

**2. Declaring Variables and Constants**

This was the starting point of the compiler. Lex was first configured to recognise patterns in the input file so as to be able to return tokens for processing by Bison. Extra information is passed between Lex and Bison using a created class, “Token”. This allows semantic information to be passed alongside the token to be processed. For example, when an integer is found, the value of the integer is passed with an accompanying token of its literal value as a string. This value can then be utilised by Bison to process correctly.

Grammar rules were then created to allow context based processing of the information passed back by Lex based upon the content of the user specified Pascal text file. The most basic of this processing was to allow the user to specify variables for use in the program. To facilitate this two structs were created, one for registers and one for variables. These structs contain the different essential attributes of registers necessary for contextual processing of input files.

Arrays were then created of the structs to represent multiple instances of the objects. An array of sixteen registers was created to represent the sixteen registers of an ARM CPU, and a list of fifteen variable structs was created. Fifteen was chosen at this time as fifteen of the sixteen registers could be utilised by the program (the sixteenth is uneditable as it holds the processor program counter).

When a variable is declared in the user code, an instance is initialised in the variable bank containing its name, type, and a bool showing it is in use. From then on, the compiler is able to reference the variable and ensure that it is declared and so valid for code use.

The variable was then assigned to a register in the register bank for physical processing on the hardware.

This structure for the compiler provided the starting point from which all else was built. Structs and arrays were expanded as needed to facilitate new functionality as progress was made through the assignment.

Such expansions were the restriction of user variables in registers zero, eleven, twelve, thirteen, and fourteen. These were reserved for the print function, memory addresses, temporary operations register, stack pointer and link register.

The variable list was also expanded to allow up to fifty variables once I had implemented a satisfactorily operational memory management system. This necessitated the addition of memory addresses to variables to keep track of their position in memory.

Constants can be declared under the section CONST and declared in the same way as a var with an assignment immediately next to it. These variables are not allowed to be changed during execution of the program.

**3. Variable Assignment**

The next logical step for the compiler was to allow users to, having created variables, assign values to them for use in code. This is done in Pascal using the set equal (‘:=’) operator. To do this, a function was created, setAssignment(string, string). This was designed so as to allow variables to be set equal to either other variables, or to literals. This is done in ARM using the MOV instruction.

The setAssignment first checks that the destination, the first string, is a valid variable using varCheck(string). Once this is established, the function checks the nature of the source, either a literal number or another register. If a literal number, the function checks that the destination function is of the integer type. If the source is another variable, the function checks that the variable exists and is of matching type using checkTypeMatch(string, string). Providing the input is valid, the function retrieves the register number for the variable(s) using getRegister(string) which scans the register bank for the location of the sought variable and returns its register number. MOV Rd <source> is then outputted.

Later, following the addition of constant declarations, a check was added to disallow assignment to a variable declared constant.

**4. Arithmetic Operations**

Arithmentic operations were added by implementation of the addition (‘+’), subtraction, (‘-‘) and multiplication (‘\*’) operators and their associated precedence’s. Division by non power of two was not added and throws an error as there was not time to provide a macro ARM code to perform the operation as there is no native non floating-point instruction.

Once an arithmetic calculation is found by Lex and returned to Bison, it is parsed to output the correct code. This is done using three functions, addOp(string, string), subOp(string, string) and mulOp(string, string).

These functions take account of the four possible combinations of each operation, register-register, register-literal, literal-register, literal-literal. The reversal of register-literal, literal-register is a relevant subtlety for subtraction where the order is relevant and so necessitated the use of RSB instead of SUB for subtraction a register from a literal (as the literal cannot be in the second instruction position). numCheck(string), varCheck(string) and getRegister(string) are all used to process and output the correct code.

Optimisations were included for multiplication operations. If a number/variable is multiplied by one, nothing is outputted. If a number/variable is multiplied by a power of two, the compiler calculates the logical left shift needed to do this multiplication, saving on the massive overhead associated with the MUL instruction. Similarly, division by powers of two is supporting using logical shift right.

**5. Conditional Statements**

The compiler supports both ‘if then’ and ‘if then else’ conditional statements. These were implemented using carefully thought out grammar rules and a series of utility functions. For the if then, a comparison is written to evaluate the condition of execution, with a conditional branch to exit to be taken if the condition fails. The ‘if then else’ is slightly more complicated with two exit labels written and two branches, one to be conditionally taken if the initial if condition fails to jump to the start of the else commands, and another to jump to the exit of the section if the condition passes and the then statements have executed.

Initially this was done very simply using exit labels written in order, however it was realised that this did not allow the use of nested conditional statements. A vector was therefore created that allows the exits to be written in the correct order within their loops and keep the code outputted correct. I think this was an effective method of solving the problem.

Initially I had designed the compiler to output conditional endings to instructions so as to make them run conditionally after comparison and save the overhead associated with branching. However, because this did not allow for nesting, branching was utilised.

Global variables and stacks were created to keep a track of entry and exit labels.

**6. Printing Results**

Printing of results can be done using write() in Pascal and in the compiler was implemented using the armWrite(string) function. The ARM code for this operation was provided for us, and all that the compiler does is move the variable to be written to R0 and branch and link to the provided PRINTR0\_macro function.

**7. Loops**

For and while loops are supported by the compiler, included nested. This was done with the necessary grammar rules and supporting functions to process and generate correct output code. Appropriate labels are outputted around the repeating code with a comparison and branch to exit to end the loop when needed.

Following the addition of memory management, tracing had to be implemented to revert any changes to variables in the registers during a loop so that when the loop is restarted, its initial conditions are the same and maintain continuity.

**8. Functions**

Functions are declared alongside the variable declarations. A new struct was created to hold information on name, label name, temporary variable name, the register in which this variable must reside, and its types.

As the ARM code for the function is pre-written and static, the temporary variable register always stays the same. To call a function, the input variable is moved to the temporary variable register and a branch link to the function label is made. This is the same as for the PRINTR0\_ function.

The compiler carries out type checking for the input variable and ensures that the temporary register is free for use with the memory management. If the register is taken, the necessary variable is swapped in, saving the existing variable to its position in memory.

Because of the way functions are implemented in the compiler, recursive functions are supported and function correctly.

**9. Arrays and Pointers**

Due to an unfortunate accident, a version of the compiler with Arrays near completed was lost the day before the submission deadline and a backup from several days before had to be reverted to. As a result, there was insufficient time remaining to recover this lost work.

From what had been implemented, Arrays used a similar principle to variables just with wider ranging memory (four\*number of elements). The memory management was then used to swap and save the elements of the array into registers as and when needed.

Making these arrays usable in for loops for example, was very tricky and needed some complex grammar however had been completed when the Bison grammar file was lost. At that time, arrays were not working in for loops printing the array elements however as the grammar was not recognising the array and outputting the correct code to swap the correct element into the registers to be printed.

It was working for assignments however in conditional statements and loops and was able to check indices range and indicate segmentation faults etc.

**10. Memory Management**

With the restriction of only R1-R10 being available for use by variables, the compiler implements memory management to allow the use of hardware limited amount of variables. Whenever a variable is used in code, the compiler checks the current register configuration for that variable. If it is not in registers, it is swapped in with the old variable being committed to memory and the new variable loaded.

Continuity errors can be caused in loops by swapping variables around during the loop code. To correct this and ensure correct functionality, changes during loops are tracked and reverted. This ensures that the registers are in the state that they were in at the beginning of the loop and variables are not lost/overwritten.

**11. Error Detection and Correction**

Detecting and correcting errors in the compiler was a long hard a tedious process. Often after changing the compiler or adding new features it would throw up segmentation errors on compilation. Often the only way to debug was to insert cout’s throughout the code and work out where the seg fault was occurring.

Other errors with the output code required reanalysis of the generating code to establish how to correct the outputted code.

The final generated code for each sample up to six has been analysed and run on the ARMulator to assert its correctness with regard to the input Pascal code.

**12. Compiler Errors**

The compiler is able to analyse input code and detect common errors. Not only this, but it is able to tell the user why an error was detected and where in their input file the error occurs with line and column numbers. This has proved useful many times when attempting to pinpoint syntax errors in my grammar and establish which elements of the input code contained grammar “errors” not supported by the compiler grammar.

This allows the compiler to be useful to the programmer and make for construction help with correcting incorrect input code.

**Personal Experience**

The writing of this compiler has been a bumpy and at times deeply frustrating challenge. It has, however, at times been deeply rewarding too. It is a great feeling to have a logic challenge that you are able to conquer.

Having never written a program with more than a couple of hundred lines of code, it is a big jump to the 1300 that make up the compiler.

This assignment has taught me a lot about the writing of complex software, demonstrating clearly the advantages of modular programming, considering design and structure carefully, breaking down issues to the lowest level possible, constant testing, how to debug efficiently and most of all the importance of backing up working versions before embarking on major improvements or changes. This assignment has also given me a new found respect for compilers, and the incredible amount of work put into them to make them what they are today.

I feel my ability as a programmer has greatly improved thanks to this project. The unfamiliar task and limited ARM resources available online also helped to make this the challenging and interesting project that it was. In future, for large projects, I think I shall write the code using a tool such as GIT to track changes and maintain the code.