Language Processors: Pascal to ARM Compiler

Christopher Schlumberger-Socha
chris.socha09@imperial.ac.uk

March 2011

1 Introduction

The aim of this project is to write a compiler which processes Pascal code, and outputs ARM assembly code using the FLEX and BISON/YACC toolset. This report will look at the different stages required to do this including:

- Pre-processor
- Lexical analysis (FLEX)
- Parser (BISON/YACC)
- Code generation/optimisation

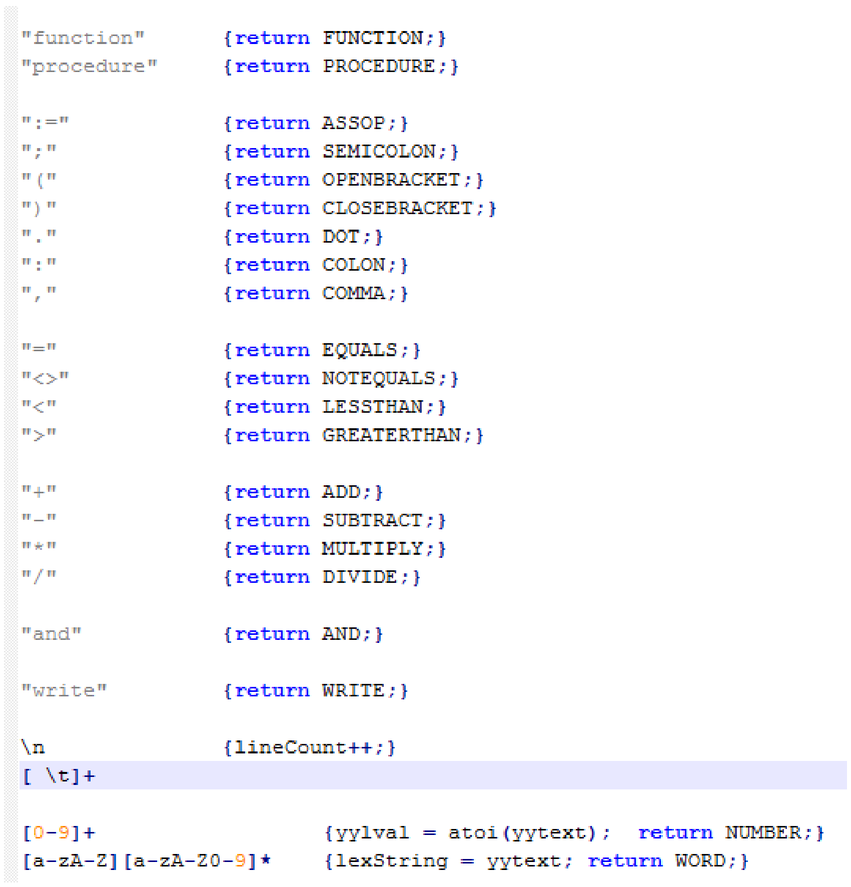
2 Pre-Processor

Pascal has three different types of comments, which start with `(*` and end with `*)`, or start with { and end with }. Some IDEs also allow for C++ style comments for entire lines such as `//`. Before analysing and parsing the Pascal source file, we need to remove all comments; this is the job of the pre-processor. After the comments have been removed (`pre-processor.cpp`), the input source file is passed to the lexical analyser before being parsed.

3 Lexical Analyser

FLEX, our lexical analyser takes an input file and turns it into a stream of characters. These characters are then isolated as tokens, ready to be passed to the parser. In my LEX file, I have defined Pascal keywords, identifiers, relational operators, numbers and words. As can be seen in figure 1, this is mostly simple, and regular expressions have been used to form a robust definition for numbers and words.

As can be seen, `NUMBER`s and `WORD`s are at the bottom of the LEX file as I want to check the input string against all keywords before it is finally assigned as a `NUMBER` or `WORD`. `NUMBER`s are passed through variable `yylval`, and I have defined a variable `lexString` to return a string for `WORD`s. I have also added a `lineCount` variable which is incremented at the end of every line, as this will help later on to specify which line of input Pascal code, an error has been found.



```
"function"      {return FUNCTION;}
"procedure"     {return PROCEDURE;}

":="            {return ASSOP;}
";"             {return SEMICOLON;}
"("             {return OPENBRACKET;}
")"             {return CLOSEBRACKET;}
"."             {return DOT;}
":"             {return COLON;}
","             {return COMMA;}

"="             {return EQUALS;}
"<>"            {return NOTEQUALS;}
"<"             {return LESSTHAN;}
">"             {return GREATERTHAN;}

"+"             {return ADD;}
"-"             {return SUBTRACT;}
"*"             {return MULTIPLY;}
"/"             {return DIVIDE;}

"and"           {return AND;}

"write"         {return WRITE;}

\n              {lineCount++;}
[ \t]+

[0-9]+                  {yylval = atoi(yytext);  return NUMBER;}
[a-zA-Z][a-zA-Z0-9]*    {lexString = yytext; return WORD;}
```

Figure 1: FLEX code.

4 Parsing

I am using the BISON/YACC toolset to parse the input source file. I have written grammar production rules which I have put in 7 sections:

- Program start
- Variable declaration
- Assignments
- Functions
- Loops
- Conditionals
- Terms

To ensure my code is clear, I have visually separated the grammar for each section in my BISON file.

4.1 Start

This section deals with initialising the ARM program by declaring the area, and giving labels for software interrupts (`SWI`). `COUT` is the label for outputting a number to the ARM debug console, and `FINISH` is the label for ending the program.

4.2 Variable Declaration

This section deals with the declaration of global variables, functions and procedures.

The code allows for the declaration of global variables of `integer` type. Attempting to declare other types (`real`, `bool`, `char`) will result in an error message, as other types are not yet supported in this version of my compiler. Variables are assigned to registers (in `regop.cpp`), and are kept track of to ensure these are not overwritten, and that there are sufficient registers for the number of global variables.

This section also allows for the declaration of functions and procedures. Local variables can be defined (these are also assigned to registers, and tracked), and a return register is assigned to functions. Before the main content of the functions/procedures, the `STMED` command is used to push all the global variable registers, and `R14` to the stack so these are not overwritten. (Branch with Link (`BL`) is used when jumping to functions. This moves the program counter (`PC`) to `R14`, so it can be POPed back to the `PC` when the function is finished, returning to after the initial branch).Following the function content, the `LDMED` command is used to POPs all the stored global variables back from the stack, and returns the value to `PC`.

Functions have a (limited) understanding of scope, and this is something I would work on further if I had more time to develop the compiler. Recursion is another function which I havent managed to incorporate in this version of the compiler, but again this is something I would be interested in implementing if I had more time.

4.3 Assignments

This section focuses on assignments and mathematical expressions.

My assignments are rather simple grammars, which can either assign an expression or a function to a variable. As previously explained, functions use branch with link (`BL`) to ensure that the program can return to correct place in the code.

Mathematical expressions were one of the more difficult aspects of my compiler to get working, particularly due to operator precedence and due to the order of operands in subtract. Some problems faced include; when to assign temporary registers, when temporary registers are no longer needed and can be re-used, how to organise operator precedence and how can the code be optimised. I will take an in-depth look at how I solved some of these problems later in this report.

4.4 Functions

This section includes function/procedure calls, and the function to write out a number to the ARM debug console.

Functions calls have been broadly mentioned above. The inputs for the functions are moved to the appropriate registers for the local variables, followed by a branch and link (`BL`) to the function, finally the return value is moved back into the correct global variable.

The write function is a standard construction to output a single ASCII character to the ARM debug console. The main parts of this code are to ensure the correct digit is in the correct place (ie. units, tens hundreds), and to add ASCII `0` to the output value to ensure it is the correct ASCII number.

4.5 Loops

This section caters for while and for loops. While loops have a label, followed by the content of the loop and then a compare (`CMP`), and a conditional branch back to the label at the top. A problem occurs if the while loop is within another loop (while/for), where the program will run through the while content once before realising it had already met the condition. To rectify this, we need to have a compare and conditional branch (using opposite conditional execution code) to the end of while loop.

For loops are structured such that there is an assignment for the for loop index, followed by a label and then incrementing the index by one. The for loop content follows this, then finally a compare instruction and a not equals (`NE`) conditional to loop back.

4.6 Conditions

This section includes conditions (ie. `A < 7`) and `if then else` statements. These `if` statements rely heavily on conditional execution; statements following an `if` have one conditional execution code, while statements following `else` have the opposite conditional execution code. I have had problems with 2 shift/reduce errors which originate from the `if then else` grammar, although this is a known case called the dangling else ambiguity. The warning for these 2 errors have been suppressed to keep the console clean while compiling.

4.7 Terms

This section covers `NUMBER`s, `WORD`s and brackets. `NUMBER`s and `WORD`s are placed into a vector of a structure (with details about the term type and value), with global variable termTracker, being passed between grammar rules, allowing for functions to extract the needed data easily from the structure

5 Code Generation/Optimisations

Code generation in many cases is simple to implement, but this is not the case with mathematic expressions. Expressions have been split into `ADD`/`SUB` and `MUL`, which allows the tokens to filter through my grammar rules to obey operator precedence.

In the case of multiply, I have decided that a new temporary register is always declared. The `MUL` command can only have two register operands (no numbers), as such if either operand need to be a number, this first needs to be placed in another temporary register then multiplied. To optimise the code somewhat, if both operands are numbers, the calculation is done at compile time, and the number is simply moved to the destination register.

In the case of additions and subtractions, new temporary registers are only needed if there are no multiply commands, or there is an `ADD/SUB` at the end of a longer expression. It should be noted that numbers can only be used as the second operand, and so the operands may need to be switched around if one is a register and the other is a number (This may require the use of RSB if swapping `SUB` operands). Similar to multiplications, if both operands are numbers, we can compute this at compile time and move the solution to the destination register.

6 Summary

In summary, my compiler can successfully parse large portions of the Pascal instruction set to ARM assembly code. The compiler gives suitable error reporting where problems have been encountered, can perform simple optimisations to code and can compute some more complex expressions and nested loops and conditionals.

In retrospect, there are many things that I would do differently if given the chance to do this project again. Ensuring the code is more modular earlier on in the development process, and using suitable unit tests throughout are some of the more structural changed I would implement. I would try and implement arrays, pointers as well as ensure that special cases for some segments of code are properly incorporated. While I do have some error checking, next time I would try and expand this to provide as much information to the end user as possible.