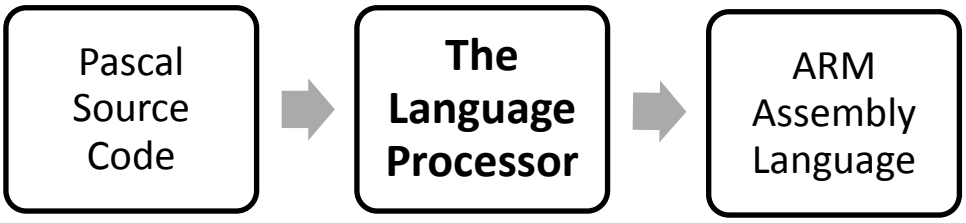
From Pascal to ARM assembly language

Olivier van Goethem – ISE2 2011

In this practical, we were required to design and implement a language processor that converts Pascal source files into their equivalent ARM assembly language source files.

The Flex and Bison tools were used to design the solution. The C code generated by these is then called inside a program written in C++.

The processor should be able to handle basic Pascal programs that consist of simple statements containing numeric variables, arithmetic operations, conditional statements, and printing of result. It should also be able to handle loops, procedures and functions.



1) FLEX

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads user-specified input files, or its standard input if no file names are given. Flex generates a C source file named “lex.yy.c” which defines the function yylex(). The file “lex.yy.c” can be compiled and linked to produce an executable. When the executable is run, it analyses its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

2) Bison/Yacc

Bison is a general-purpose parser generator that converts an annotated context-free grammar into an LALR(1) or GLR parser for that grammar. It can be used to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages. Bison is upward compatible with Yacc. C or C++ programming knowledge is required in order to use Bison.

3) Structure of the Compiler

To parse a pascal file, the language processor has two distinctive steps. First, the input file is scanned against a set of tokens defined in the Flex “pascal.l” file. If something that isn’t recognised by the scanner is present, an error will be thrown. To pass this step, the pascal code should be syntactically

correct. All of the Pascal Programming Language key words can be found in the “pascal.l” file, such as operators, variable types, begin/end keywords, etc. Flex is also a simple solution to pre-process your input file. It is easy to ignore comments, end of lines and blank spaces.

Let’s consider the case of IDENTIFIERS and DIGITS. These are declared in the “pascal.l” using regular expressions. Hence a digit requires at least one or more ciphers between 0 and 9 and an identifier requires any upper/lowercase character followed by non-necessary additional characters.

```
[0-9]+              {position += yyleng; varStr = yytext; return DIGIT;}
[a-zA-Z][a-zA-Z0-9]*    {position += yyleng; varStr = yytext; return IDENTIFIER;}
```

Fig 1. Declaring DIGITS and IDENTIFIERS

Flex allows the execution of C code when a token has been processed. In this case, notice the use of “yyleng” and “yytext”, two given variables used by Flex. “yyleng” is used to determine the number of characters that compose a token (ie: “if” would be the IF token. Its length is 2 characters long). On the other hand “yytext” is used to read the value of that token as a string. As we’ll see later, this will be very useful to pass a variable’s value to the bison file. Finally, “position” and “varStr” are two user-defined global variables that are used for error reporting and storing variable names respectively.

The “pascal.y” bison file contains the basic grammar rules for the Pascal Programming Language. Grammar rules allow the parser to know in which order the Flex tokens should appear by building a binary tree. Rules are declared the following way:

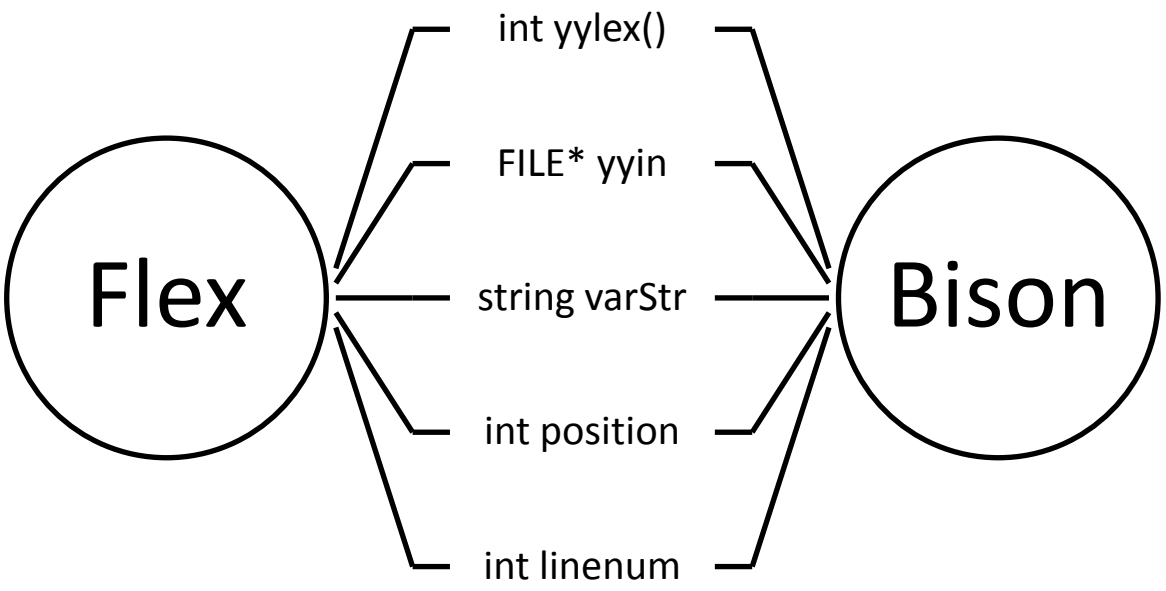
```
var_declaration : VAR variables COLON var_type SEMICOLON
        | var_declaration VAR variables COLON var_type SEMICOLON
        ;

variables : IDENTIFIER                  {addTable(varStr)}
        | variables COMMA IDENTIFIER    {addTable(varStr)}
        ;

var_type : INTEGER
        :
```

Fig 2. Variable declaration section in Pascal

Let’s consider the rule for the variable declaration section. Essentially, the following rule could have been shortened to “var_declaration: VAR IDENTIFIER COLON INTEGER SEMICOLON”. However, using

the bison syntax found in Fig 2 allows us to use a recursive grammar. Therefore it is easier to define multiple variables or in other cases arithmetic with more than two operands. The better solution would have been to use a “token.h” class. Thus each new variable would be declared as a token with access to multiple types.

There are multiple ways in which Flex and Bison can communicate. In my language processor, I decided to use global variables common to both the “pascal.l” and “pascal.y” files. Even though this isn’t the most robust way of communication, it is easy to understand and implement.



- **int yylex():** calls the lexical set from the “pascal.l” file to check if the token found is valid.
- **FILE* yyin:** yyin is a file pointer that points to the input pascal source file.
- **string varStr:** variable used to pass the string value of the IDENTIFIER and DIGIT tokens to “pascal.y”. This allows variables and functions to be stored in symbol tables.
- **int position:** variable used for error reporting. Indicates where a syntax error is present on a specific line.
- **int linenum:** variable used for error reporting. Indicated on which line there is a syntax error.

4) Breaking down the Bison file

Added to the grammar rules, the “pascal.y” file has several C++ functions that are used to generate the ARM assembly:

- **void yyerror(const char*):** This function prints any syntax errors to the console.

- **void addTable(string):** This function is used to add a new variable to the variable symbol table. Usually "varStr" is passed as a parameter for this function. The symbol table is composed of a vector of structures. Each structure contains the variable's name and register location. It was planned to contain the variable's value as well, but due to timing this wasn't implemented.

- **void addFuncTable(string):** This function is used to add a new pascal function to the function symbol table. The function symbol table is composed of a vector of structures. . Each structure contains the function's name.

- **void addFuncVarTable(string):** This function is used to add a local variable that is only visible within a function. The symbol table is composed of a vector of structures. . Each structure contains the variables's name and register location. It was planned to make this table part of the function table structure. However, due to timing this wasn't implemented.

- **int searchLocation(string):** This function is used to iterate through the variable or function-variable table. The parameter is the name of the variable and it returns its register location. In case the variable hasn't been declared, an error is thrown.

- **int searchFunc (string):** This function is used to iterate through function table. The parameter is the name of the function and it returns its label for integration into the ARM code. In case the function hasn't been declared, an error is thrown.

- **string intToStr (int):** In this function, integers are converted into strings using stringstreams.

- **void codeGen (string instruct, int dest, string operand1, string operand2):** This is the main function of the language processor. The instruction op-code is passed to the function as well as the destination register and the two operands. It then outputs the correct ARM code to an external file.

In addition to these functions, a system of flags is used to detect whether the parser is inside a 'if'/'for'/'while' loop or inside a function. These flags modify the op-codes sent to the code generator accordingly.

Registers management is minimalistic. A simple counter is used to increment the register address each time a new variable is declared. This could have been improved using either graph colouring or a LRU algorithm.

5) Challenges

Several challenges arose whilst doing the language processor. My first version was an extremely optimised one, where most of the calculations were done at compiling-time. Thus, statements such as A := 1; B := A+1; would result in the following ARM code: MOV R0, #1; MOV R1, #2.

Unfortunately, it became very difficult to continue on this path beyond the 'for' loops. Therefore, I started again, this time reading the pascal statements and literally translating them into ARM code. This resulted in inefficient but working code. Loops were easy to implement as were functions.

The arithmetic part of my language processor was something I would have liked to look into further, as multiplications aren't implemented correctly. Due to timing limitations, arrays and pointers weren't included in the final build.

The symbol tables are also something that I would have liked to improve. In addition to storing the variable's name and register location, it would have been useful to store its value as well.

More optimisations could have been done, such as using a "token.h" class and implementing the language processor using an Object-Orientated approach. This would also make error handling much simpler.

Finally, the generated ARM code couldn't be tested on ARMulator as the software was unable to run on my Laptop. However, due to the basic complexity of the programs, it was possible to determine of a code was correct or not by tracing through it.