

Marks : 16 **CSCI131**  
**Spring Session 2015**

Due date: October 30th

**Assignment 5**  
**Compilers**  
**8 marks**

*Complete the [associated exercises](#) before attempting the assignment*

### **Aim**

This assignment and its associated exercises provide an understanding of compilers and some limited experience in building a compiler using tools such as bison and flex.

### **Objectives**

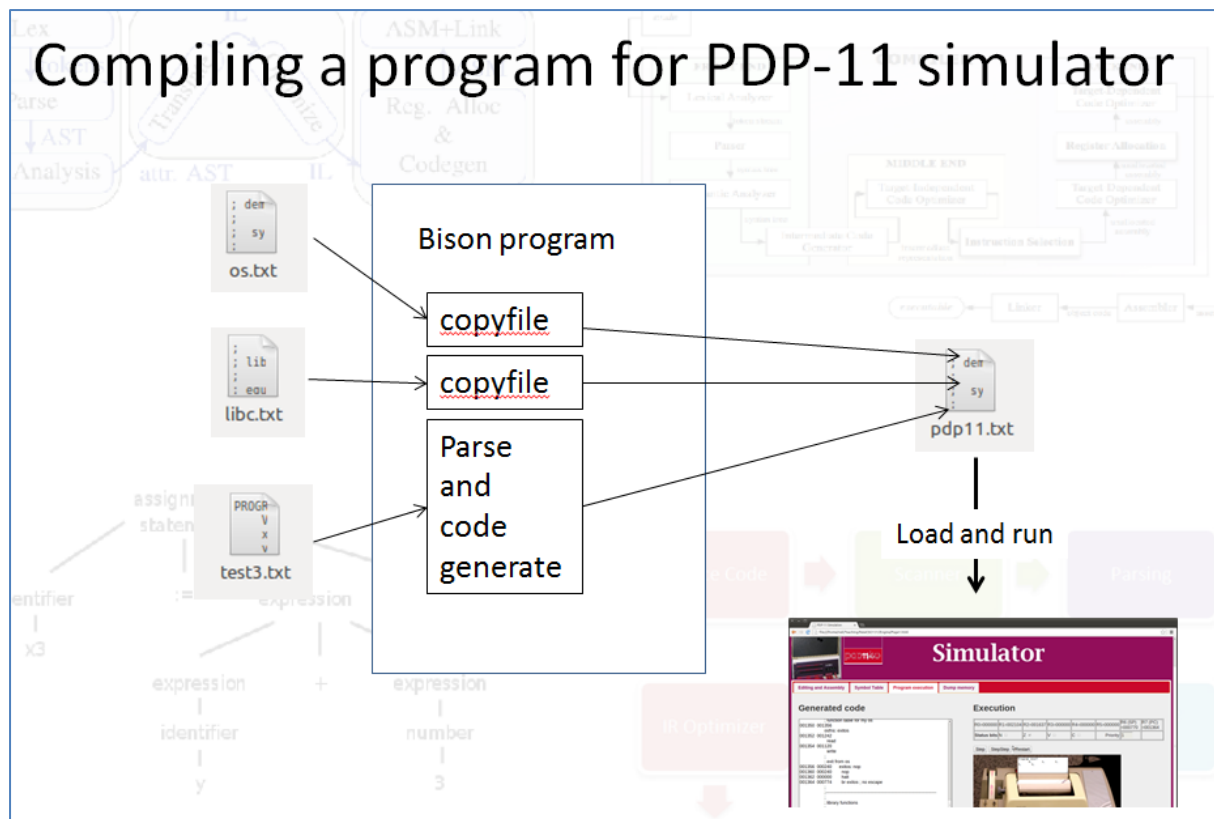
On completion of this assignment and its associated exercise, you will be able to:

- Explain the workings of a compiler
- Create applications that exploit a scanner generated using flex
- Write grammar rules for bison
- Write an interpreter that uses a parser generated using bison
- Write a simple compiler that generates executable code.

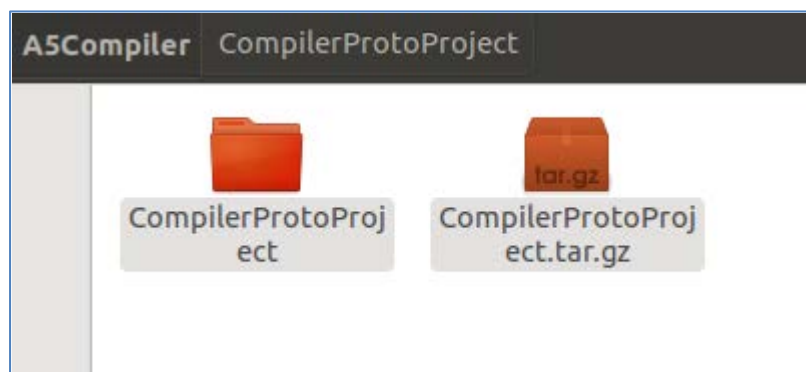
### **Task context – the PDP-11 compiler from the lecture notes**

**You are to extend the compiler system presented in the lectures.**

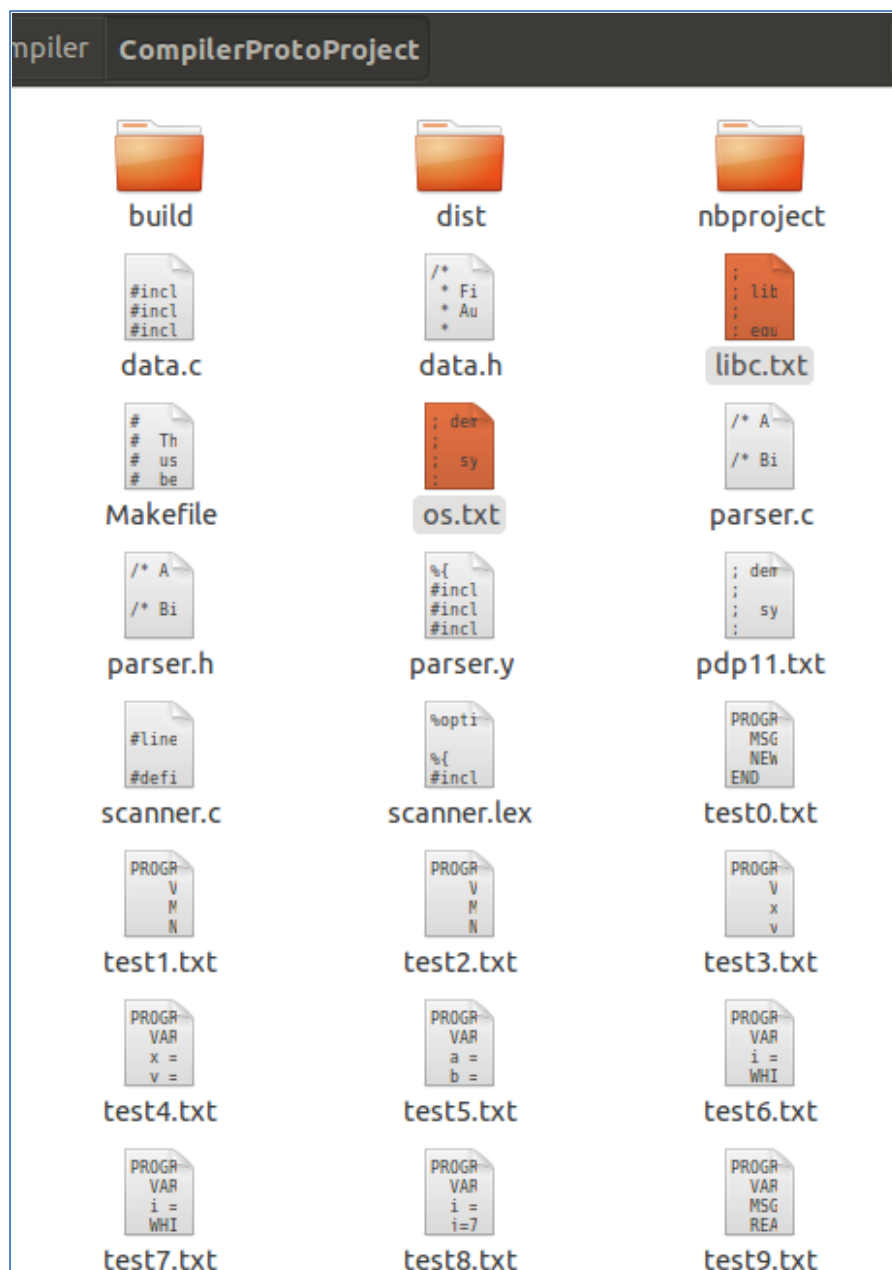
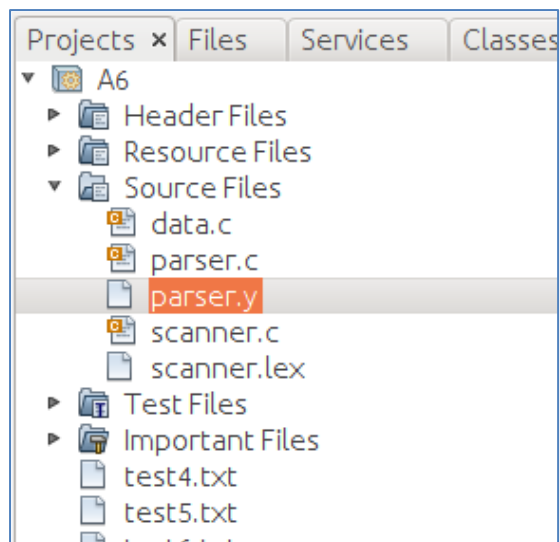
This system generates PDP-11 code from source code in a simplified quasi-Basic language. The generated code is in effect linked with a “libc” library that provides functions for reading and writing numbers (in both decimal and octal formats), and runs with “os” code. The “os” uses interrupt driven I/O; it supplies just two supervisor calls – readline and writeline.



You will start by downloading the code of the compiler system described in lectures (it is [CompilerProtoProject.tar.gz](http://CompilerProtoProject.tar.gz) in the Resources folder for the subject). This is the version that handles simple arithmetic expressions and while loops.



The project contains the parser.y and scanner.lex files that are used to generate the compiler. The folder also contains the files libc.txt and os.txt that make up the “systems library” and “operating system” for this simplified example. There are also a number of example programs that can be compiled, with the compiled code being run on the simulator that was used in Task Group 2.



The lex file is similar to those that you have already worked with:

```

16
17
18 PROGRAM      return PROGRAM;
19 END          return END;
20 VAR          return DECLARE;
21 ,           return COMMA;
22 ;           return SEMICOLON;
23 "+"         return PLUS;
24 "-"         return MINUS;
25 "*"         return MULTIPLY;
26 "/"         return DIVIDE;
27 "("         return LPAR;
28 ")"         return RPAR;
29 "="         return ASSIGN;
30 "<"         return LESS;
31 ">="        return GE;
32 WHILE        return WHILE;
33 DO           return DO;
34 ENDWHILE     return ENDWHILE;

```

The grammar rules for the parser are presented in the lecture notes. Most are similar to those that you have already seen in the exercise, but things like the WHILE construct are a little more complex.

```

statement:
| NEWL SEMICOLON { fprintf(output, "call newline\n"); }
| MSG QSTRING SEMICOLON { int ival = addstring($2); fprintf(output, "writeln\nmsg%d\n", ival); }
| READ IDENTIFIER SEMICOLON { fprintf(output, "call readint\nmov r0,%s\n", $2); }
| PRINT expression SEMICOLON { fprintf(output, "mov (sp)+,r0\ncall printint\n"); }
| READOCI IDENTIFIER SEMICOLON { fprintf(output, "call readoct\nmov r0,%s\n", $2); }
| PRINTOCI expression SEMICOLON { fprintf(output, "mov (sp)+,r0\ncall printoct\n"); }
| IDENTIFIER ASSIGN expression SEMICOLON { fprintf(output, "mov (sp)+,%s\n", $1); free($1); }
| WHILE { fprintf(output, "wh%d: ", lblcount); nestingstack[nestinglevel++] = lblcount; }
| comparison { fprintf(output, "tst r0\nbge wh%d\n", lblcount); lblcount++; }
| DO { int lbl; nestinglevel--; lbl = nestingstack[nestinglevel]; fprintf(output, "br wh%d\nwh%d: ", lbl, lbl); }
| statements ENDWHILE SEMICOLON

```

The language construct is:

```

WHILE
  comparison
  DO
  statements ENDWHILE SEMICOLON

```

With something like this, code has to be emitted at several points. When the parser meets the WHILE token, it will have to set a label.

```

WHILE { fprintf(output, "wh%d: ", lblcount); nestingstack[nestinglevel++] = lblcount; }

```

It must then handle the comparison which will produce some output instructions.

```
WHILE {
    comparison
DO {
```

When it reaches the “DO” token, it must complete the code that allows for a jump to the end of the block if the comparison condition is not satisfied.

```
DO { fprintf(output, "tst r0\nbeg ewh%d\n", lblcount); lblcount++; }
```

The statements must then be compiled. Finally, at the ENDWHILE, the parser must generate the code for a jump back to the start of the WHILE construct.

```
statements ENDWHILE SEMICOLON { int lbl; nestinglevel--; lbl = nestingstack(nestinglevel); fprintf(output, "br wh%d\nnewb%d: "
```

It also has to keep track of labels in a stack (this is necessary as one can have nested WHILE loops).

The handling of comparisons involves generation of code using PDP-11 instructions such as ble (branch on less than or equal):

```
comparison: LPAR expression LESS expression RPAB {
    fprintf(output, "clr r0\n");
    fprintf(output, "cmp (sp)+, (sp)+\n");
    fprintf(output, "ble cmp%d\n", comparecount);
    fprintf(output, "inc r0\n");
    fprintf(output, "cmp%d: ", comparecount);
    comparecount++;
}
```

(The code generated for a comparison is devised to set r0; r0 is set to zero if the comparison failed, or to 1 if it was satisfied. This r0 value is then tested in the code used to determine whether to jump past the while inner block.)

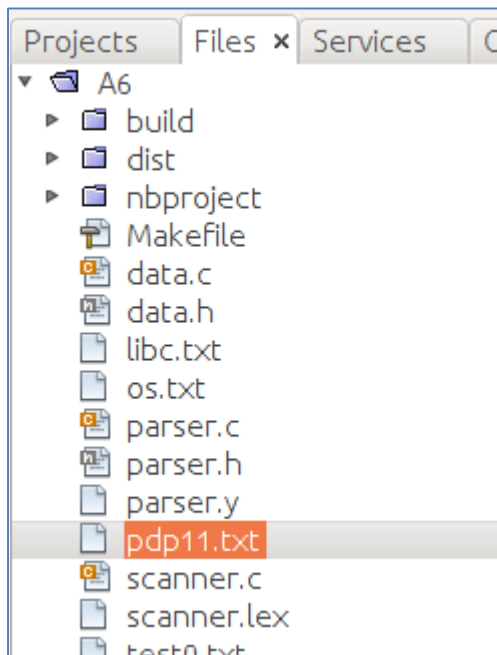
The code should work. It should generate a compiler that can handle the simple programming language and generate working PDP-11 executables.

Try it. Build, and then run with one of the supplied quasi-Basic programs, e.g. test0.txt:

```
test0.txt x test1.txt x
PROGRAM demo;
  MSG "It works - and about time too!";
  NEWL;
END
```

```
Output x
A6 (Build, Run) x A6 (Run) x
> Enter name of file with program to be interpreted : test0.txt
```

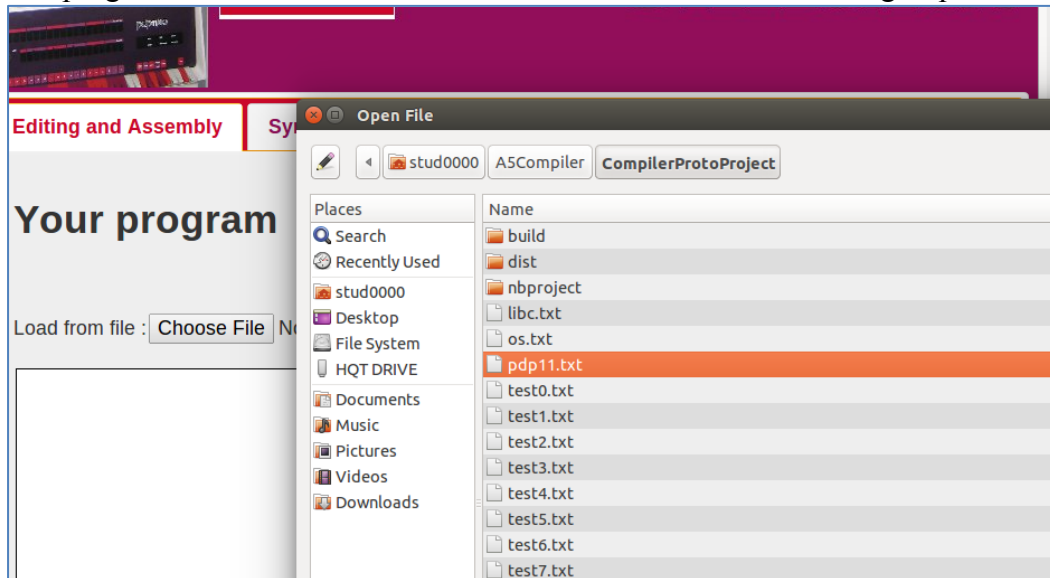
Compilation should have generated the file pdp11.txt with the generated code, the libc.txt code, and the osx.txt code.



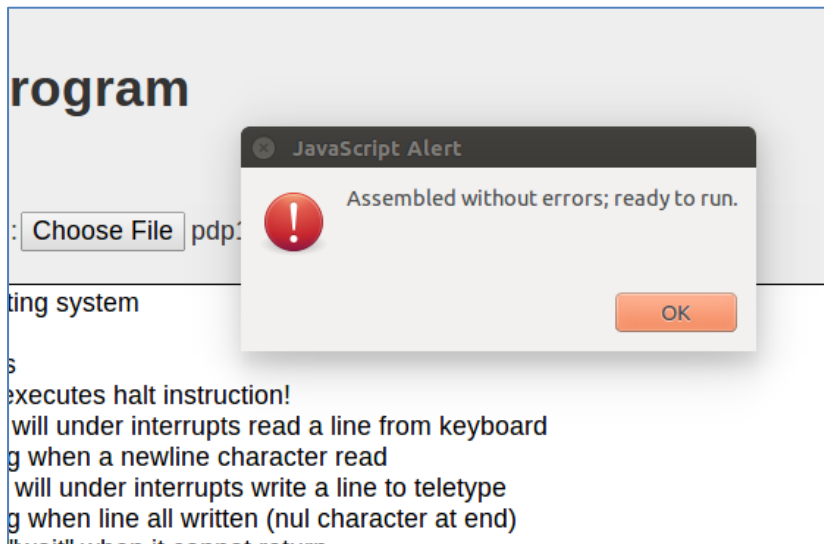
You can view the PDP-11 instructions generated from that source program (right at the end of pdp11.txt):

```
;-----  
.origin 2400  
application: writeline  
msg0  
call newline  
writeline  
msg1  
call newline  
exit  
msg0: .string "Program demo"  
msg1: .string "It works - and about time too!"  
.end osstart
```

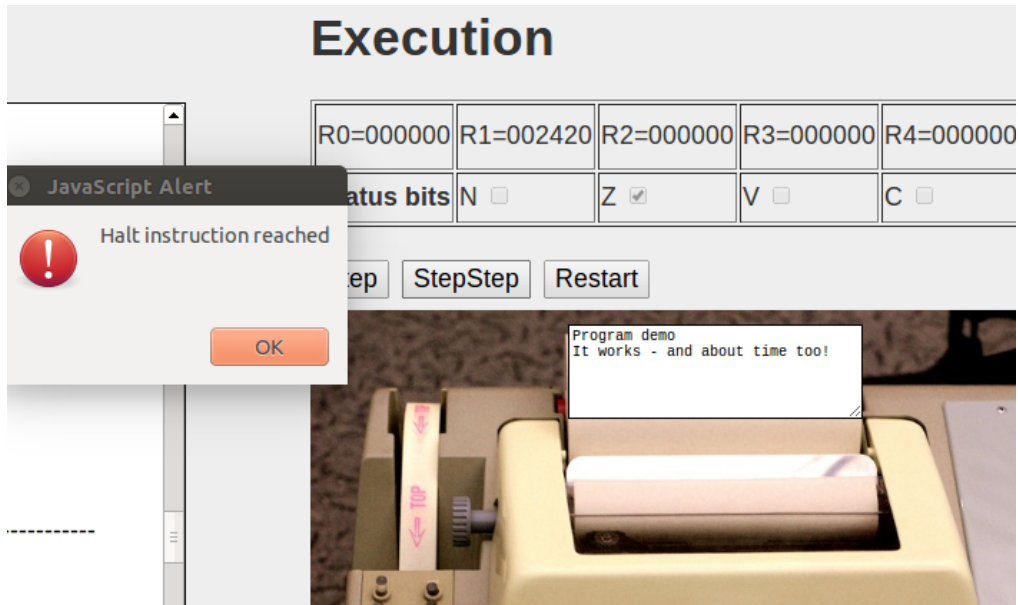
The program can be loaded into the simulator that was used in Task group 2:



The code should assemble:

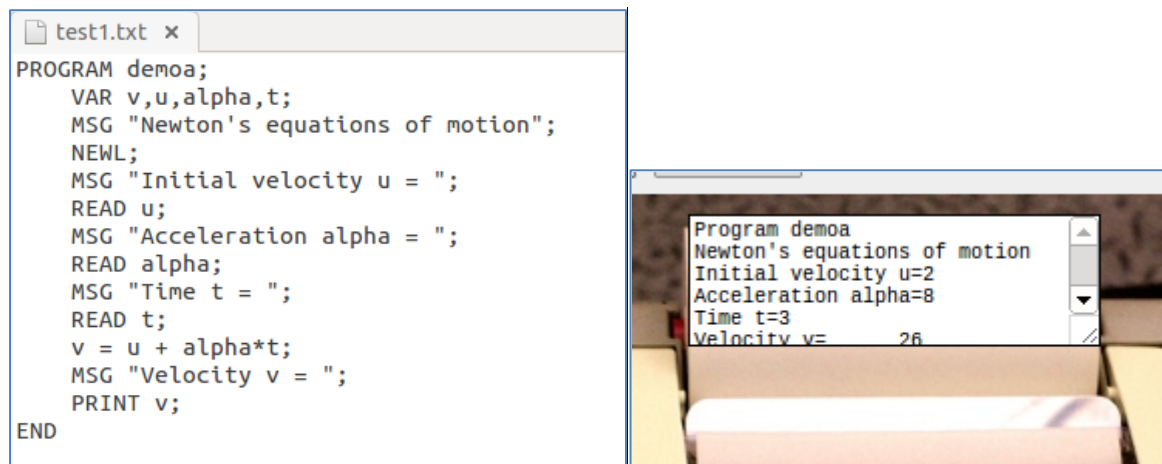


The code should run:



Try some of the other example programs as well:

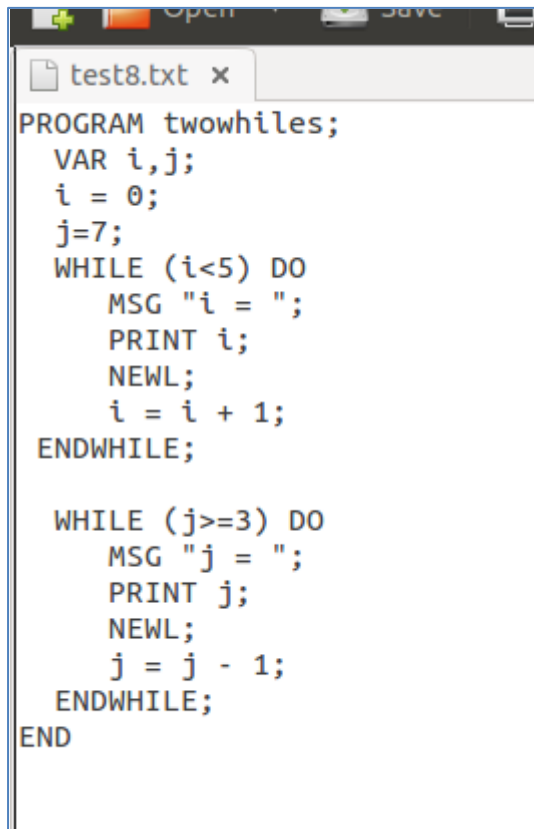
test1.txt



(Requires input values).

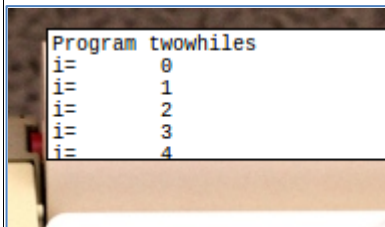


test8.txt



```
PROGRAM twowhiles;
  VAR i,j;
  i = 0;
  j=7;
  WHILE (i<5) DO
    MSG "i = ";
    PRINT i;
    NEWL;
    i = i + 1;
  ENDWHILE;

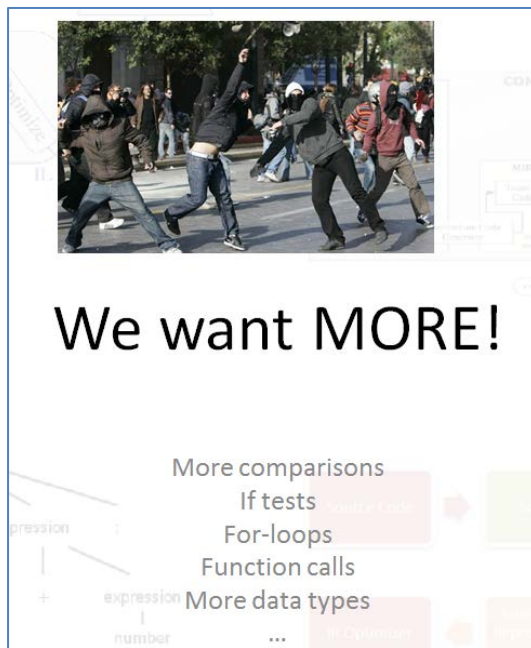
  WHILE (j>=3) DO
    MSG "j = ";
    PRINT j;
    NEWL;
    j = j - 1;
  ENDWHILE;
END
```



```
Program twowhiles
i= 0
i= 1
i= 2
i= 3
i= 4
```

(Have a look at the instruction sequence generated so as to gain an understanding of how code is generated for conditionals and for the overall WHILE construct with its labels and branch instructions.)

## Task



*We want more comparisons!*

*We want an 'if test'!*

OK – you extend the language and its compiler!

The comparisons should be fairly obvious. The version you downloaded dealt with “<” (less than), and “>=” (greater than or equal). What about the others?

You will need to handle “<=”, “==”, “!=”, “>”. All you need do is add some new tokens, define the flex code for recognizing these:

" ) "	<b>return</b> RPAR;
" = "	<b>return</b> ASSIGN;
" < "	<b>return</b> LESS;
" >= "	<b>return</b> GE;
" <= "	<b>return</b> LE;
" > "	<b>return</b> GT;
" == "	<b>return</b> EQ;
" != "	<b>return</b> NEQ;
WHILE	<b>return</b> WHILE;
DO	<b>return</b> DO;
ENDWHILE	<b>return</b> ENDWHILE;
IF	<b>return</b> IF;
THEN	<b>return</b> THEN;
ENDIF	<b>return</b> ENDIF;

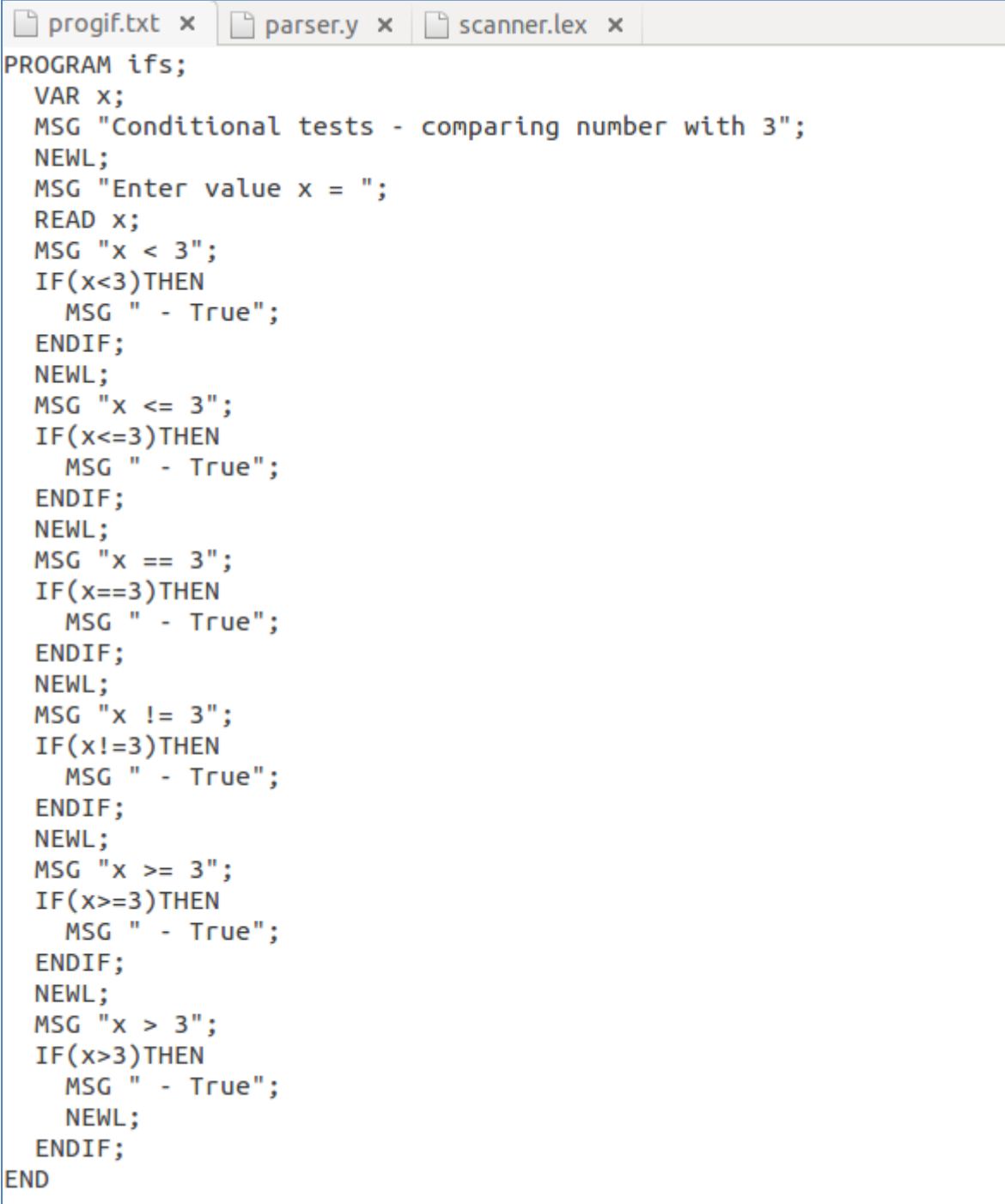
Then compose the grammar rules that generate appropriate instruction sequences that use the different comparison instructions implemented in the PDP-11. You shouldn't find that too hard and you should be able to test you compiler with WHILE constructs that use these additional comparisons.

The “if” test should take the form:

```
IF condition THEN
    statements
ENDIF;
```

(Don’t try for ELSE, or ELSIF etc.)

You want to be able to process programs such as the following:



The screenshot shows a text editor window with three tabs: 'progif.txt', 'parser.y', and 'scanner.lex'. The active tab is 'progif.txt', which contains the following code:

```
PROGRAM ifs;
VAR x;
MSG "Conditional tests - comparing number with 3";
NEWL;
MSG "Enter value x = ";
READ x;
MSG "x < 3";
IF(x<3)THEN
    MSG " - True";
ENDIF;
NEWL;
MSG "x <= 3";
IF(x<=3)THEN
    MSG " - True";
ENDIF;
NEWL;
MSG "x == 3";
IF(x==3)THEN
    MSG " - True";
ENDIF;
NEWL;
MSG "x != 3";
IF(x!=3)THEN
    MSG " - True";
ENDIF;
NEWL;
MSG "x >= 3";
IF(x>=3)THEN
    MSG " - True";
ENDIF;
NEWL;
MSG "x > 3";
IF(x>3)THEN
    MSG " - True";
    NEWL;
ENDIF;
END
```

(The hardest part of implementing “IF ... ENDIF;” will probably be keeping track of labels. You will need to exploit, and maybe extend the code that handles labels for WHILE

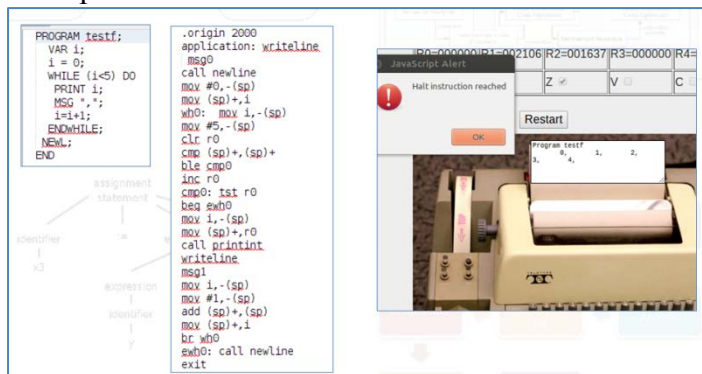
constructs. Of course, you should be able to handle IF constructs inside WHILE loops and WHILE loops inside IF constructs).

## Report

Your report should contain:

- scanner.lex
- parser.y
- examples of source code and generated PDP-11 instructions
- evidence that your generated PDP-11 code runs correctly.

Example:



(If pasting in screen shots of parser.y make sure that they are clear for the marker to read.)

## Submission

Prepare your report and convert to PDF format as the file A5.pdf.

Submission is done electronically via a program called `turnin` that runs on “banshee” – the main CS undergraduate machine. You must first transfer your A5.pdf file to your home directory on banshee (this is different from the home directory that you access on the Linux machines). You can transfer the file using a SSH file-transfer program. The Ubuntu OS allows you to open a file-browser connected to your banshee home directory – and you can simply drag your A5.pdf file across using the visual file browser.

For CSCI131, assignments are submitted electronically via the `turnin` system. For this assignment you submit your assignment via the command:

```
turnin -c csci131 -a 5 A5.pdf
```

This assignment will be due in the last week of session; consequently there will be NO late submissions.

The program `turnin` only works when you are logged in to the main banshee undergraduate server machine. From an Ubuntu workstation in the lab, you must open a terminal session on the local machine, and then login to banshee via `ssh` and run the `turnin` program.

## Marking

The assignment is worth 8 marks total.

- Appearance and structure of report: 1 mark
- Evidence for correct operation including suitable test programs that utilize the new “IF” language construct and the additional comparison tests: 2 marks
- Code and explanations of your implementation: 5 marks total