

# CS1006 – P1 - mini\_compiler

Set by Saleem Bhatti, University of St Andrews, Mon 26 Jan 2015

*originally created by John Thomson*

**Deadline: 2100 on 12 Feb 2015**

## Specification

---

(to be read in conjunction with the notes from the lecture)

LOGO is an interpreted language, in which an on-screen cursor called a 'Turtle' can be moved around a screen, leaving lines in its wake, and thus creating drawings.

PostScript is a stack-based language commonly used as a page description language on printers.

Write a LOGO->PostScript compiler. The compiler should be able to correctly translate LOGO programs into PostScript, which is interpretable by GhostScript, or a compliant printer. You are supplied with three sample LOGO programs, each drawing a different *fractal*. You are also supplied with a small starter code bundle which you may use. You may change this in any way you like.

NB: Some of you may be aware of standard tools like Lex and Yacc. Please do not use such tools in your practical. Write your own lexer and parser.

The grammar supplied in the practical should be considered definitive for LOGO.

## Lexical Analysis:

---

Your lexical analyser should convert your LOGO program into a sequence of Tokens. Suggested tokens should include tokens for each keyword in the language, for each operator, for brackets and a number and an identifier token (all the *terminals*).

## Parser:

---

The parser should accept the stream of tokens generated by your lexer and parse them appropriately into a suitable data structure. The parser should generally deal with all the *non-terminals*.

The parser should ensure the *syntax* of the LOGO code is correct.

## Code Generation:

---

Generate appropriate code in PostScript based on your parsed program and output it to a file.

## Deliverables

---

As explained in the lecture, the deliverable for this project comprises two parts:

1. Your source code, including a README file with any execution instructions.
2. Your report (a PDF file).

Your source code should be well structured and well commented.

Your code submission must work on the Linux lab workstations.

Your report should give a detailed account of your solution. It should contain a justification of all of the design decisions you made and some sample output of your program.

The following describes three levels of deliverable, intended as a guide to the mark you can expect.

### **Basic Deliverable**

---

An implementation of the lexer and the parser which parses anything written in the LOGO language, and a report.

### **Intermediate Deliverable**

---

In addition to the Basic Deliverable, code generation should be implemented and your compiler should produce correct code. Exit gracefully when syntactically incorrect LOGO code is received.

### **Advanced Deliverable**

---

In addition to the Intermediate Deliverable, implement an error reporting system which produces useful error messages for the programmer when the LOGO program input is not syntactically correct. Allow your program to recover from these errors and continue to parse the code until the end if possible

Try to write your own LOGO code! Any other interesting fractals?

### **Marking**

---

Up to 10 for Basic.

Up to 17 for Intermediate.

Up to 20 for Advanced.

Please note that the final mark you get will be based on the overall quality of your submission - attempting intermediate and/or advanced features is not a guarantee of a higher mark.

### **Administrative information**

This assignment is worth 25% of your overall coursework mark for CS1006.

Normal lateness penalties apply:

<https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/lateness-penalties>

You are reminded to follow Good Academic Practice:

[https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/Good\\_Academic\\_Practice](https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/Good_Academic_Practice)

Marking will follow the University's 20-point scale:

[https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/Mark\\_Descriptors](https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/Mark_Descriptors)

## PostScript Data Sheet

PostScript is a FILO stack-based language. PostScript commands use values on the stack. To push a value onto the stack, simply enter a value on its own:

PS:	Stack status after command:
2	{2}
5	{2, 5}
35	{2, 5, 35}

Operators with a result – pop two values, and push the result:

add	{2, 40}
30	{2, 40, 30}
sub	{2, 10}

Boolean operators – pop two values.

5	{2, 10, 5}
gt	{2}

Remember, boolean operators are only used in conditional statements.

You should also use one register (temporary storage) called Arg. This should be used to represent the argument of the procedure. It is initially set to 0 to represent to void value given to MAIN.

You can push the value of Arg onto the stack (Arg contains 10 here):

Arg	{2, 10}
add	{12}

You can pop the bottom value from the stack and copy it to Arg:

```
/Arg exch def      {}      (Arg = 12)
```

Conditionals can be expressed as:

```
boolean_op
{
  TRUE_STATEMENTS
}{
  FALSE_STATEMENTS
} ifelse
```

i.e.:

```
20          {20}
30          {20, 30}
eq          {}
{
TRUE_STATEMENTS
}{
FALSE_STATEMENTS
} ifelse
```

Procedure definitions are handled as follows:

```
/MyProc {
PROCEDURE CONTENTS
} def
```

Procedure calls are as follows:

MyProc

You must set up the procedure by setting the value of Arg as the argument before calling. *N.B.: Do not use more than one register. Use it only for this purpose. Use the stack for everything else.*

Valid Postscript files start with a set *prologue* and end with a set *epilogue*. These are provided for you.

List of PostScript operators:

add – add.  
sub – subtract  
mul – multiply  
div – divide  
eq – is equal to  
ne – is not equal to  
ge – greater than or equal to  
gt – greater than  
le – less than or equal to  
lt – less than.

Procedures for you in Postscript are provided (they are in the prologue) which perform Forward, Left and Right. You must call them as required using 'Forward', 'Left' or 'Right'.

# LOGO Grammar

```
prog ::=
  { proc } ;

proc:
  "PROC" ident '(' ident ')' stmts ;

stmt ::=
  "FORWARD" expr
  "LEFT" expr
  "RIGHT" expr
  "IF" expr "THEN" stmts "ELSE" stmts "ENDIF"
  ident '(' expr ')' ; //procedure call!

stmts ::=
  { stmt } ;

expr ::=
  primary-expr
  binary-expr ;

primary-expr ::=
  num
  ident ;

binary-expr ::=
  expr op expr ;

op ::=
  '+' | '-' | '*' | '/' |
  '==' | '!=' | '>' | '<' | '<=' | '>=' ;

ident ::=
  letter { letter | digit } ;

letter ::=
  "a"..."z" | "A"..."Z" ;

num ::=
  "0"
  nonzerodigit { digit } ;

digit ::=
  "0"..."9" ;

nonzerodigit ::=
  "1"..."9" ;
```

num and ident are not technically terminals here, but you can treat them as such for simplicity. Do not make classes for individual digits and letters.