# Programming Languages 3

# Tutorial Exercises (2013-14)

You should attempt these tutorial exercises *before* the PL3 tutorial session in the week concerned. Then you can contribute to the discussion during the tutorial session.

Sample solutions will be posted at the PL3 Moodle site later in the course. *Attempt each exercise before consulting the sample solution.*

## Exercises 1 (Syntax)

**1A.** *(Regular expressions)*

In Cobol, an identifier consists of letters, digits, and hyphens (‘`-`’), but always starts with a letter. Hyphens may not occur consecutively, nor at the end of an identifier. Thus ‘`setup`’, ‘`set-up`’, and ‘`s-et-up`’ are well-formed identifiers, but ‘`set--up`’, ‘`-setup`’, and ‘`setup-`’ are ill-formed.

(a) Write an RE that specifies the syntax of Cobol identifiers.

(b) Write an EBNF grammar of Cobol identifiers.

**1B.** *(Regular expressions)*

The Unix utility program `egrep` provides a pattern-matching notation, which is summarized in the course notes (slide 1-16).

(a) Show how each of the following `egrep` patterns could be expressed in standard RE notation:

```
b[aeiou]t    b.t    be*t    b[aeiou]*t    b(a|e|i)+t
```

(b) Write and test `egrep` commands to do the following with a given file:

(i) Find every line that contains ‘`<H1>`’, ‘`<H2>`’, …, or ‘`<H9>`’.

(ii) Find every line that contains a sequence of one or more lower-case letters enclosed between ‘`{`’ and ‘`}`’.

(iii) Find every line that contains a sequence of zero or more characters enclosed between ‘`{`’ and ‘`}`’.

(iv) Find every line that contains ‘`Mr`’, ‘`Ms`’, ‘`Mrs`’, or ‘`Miss`’.

(v) Find every line that contains ‘`b`’ followed by one or more consecutive occurrences of ‘`an`’ followed by ‘`a`’.

**1C.** *(BNF)*

Consider the mini-English BNF grammar in the course notes (slide 1-22). That grammar generates some sentences, such as ‘`I sees the cat .`’, that would be ill-formed in the English language because the subject does not “agree” with the verb.

(a) Modify the grammar to ensure that a 1st person subject agrees with a 1st person verb, and a 3rd person subject agrees with a 3rd person verb.

(b) Add 2nd person subjects and verbs to the grammar.

(c) *(Optional)* Add plurals to the grammar.

**1D.** *(Phrase structure)*

Calc’s BNF grammar includes the following production rules:

```
expr  =  prim
      |  expr ‘+’ prim
      |  expr ‘-’ prim
      |  expr ‘*’ prim
prim  =  num
      |  id
      |  ‘(’ expr ‘)’
```

(a) Draw the syntax trees of the following expressions:

```
x+y*z    x*y+z
```

(b) Note that all operators have equal priority. Modify the grammar so that ‘`*`’ has higher priority than ‘`+`’ and ‘`-`’. For example, ‘`x+y*z`’ should be evaluated like ‘`x+(y*z)`’, but ‘`x*y+z`’ should be evaluated like ‘`(x*y)+z`’.

**1E.** *(Ambiguity)*

Suppose that a programming language’s BNF grammar includes the following production rules:

```
        com  =  id '=' expr ';'
             |  'while' '(' expr ')' com
             |  com com
```

(a) Show that this grammar is ambiguous.

(b) How does the Fun grammar avoid this ambiguity?

(c) How does the Java grammar avoid this ambiguity?

**1F.** *(EBNF)*

Consider a programming language whose EBNF grammar includes the following production rules:

```
        prog  =  decl+ com+
        com   =  id '=' expr ';'
              |  'if' '(' expr ')' com ( 'else' com )?
              |  'while' '(' expr ')' com
              |  '{' com+ '}'
        decl  =  type id
        type  =  'bool' | 'int'
```

Suppose that this programming language is to be extended with the following constructs. Extend the grammar accordingly.

(a) A for-command is to have the form illustrated by:

```
for (i = 1 .. n) {
  write i; write i*i;
}
```

Allow any expressions to the left and right of '`..`'. Allow any command after '`)`'.

(b) A case-command is to have the form illustrated by:

```
case (m-n) {
  when 1 : x = m;
  when 2 : x = n;
  otherwise : x = 0;
}
```

The case-command may contain one or more '`when`' limb, each labeled by a numeral. The last limb may optionally be followed by an '`otherwise`' limb. Allow any command after each '`:`' in each limb.

(c) A procedure call command is to have the form illustrated by:

```
p ();
q (b, n-1);
```

The procedure call may have any number of actual parameters. An actual parameter is an expression.

(d) A procedure declaration is to have the form illustrated by:

```
procedure p () {
  x = 0;
}
procedure q (bool c, int i) {
  x = 2*i;
  if (c)
    x = i-1;
  else
    x = i;
}
```

The procedure declaration may have any number of formal parameters. The procedure's body may be any command.