Programming Languages 3: Questions and Answers: April/May 2013
Duration: 90 minutes.
Rubric: Answer all three questions. Total 60 marks.

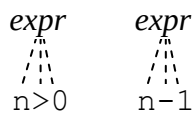
1. (*Syntax*)

Box 1 shows part of the BNF grammar of the programming language Fun. It shows the syntax of commands and sequential commands. It omits the syntax of expressions (not needed in this question).

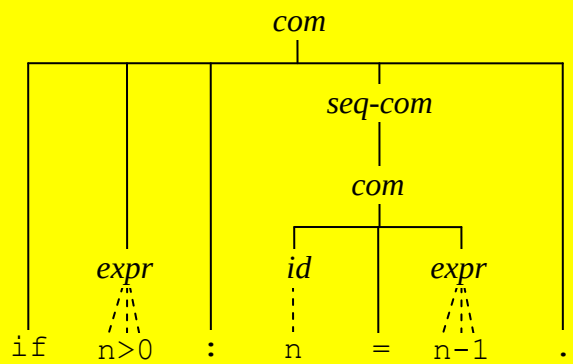
(a) Show the syntax tree of the following Fun command:

```
if n>0 : n = n-1 .
```

You may assume that `n>0` and `n-1` are expressions. Your syntax tree should show these expressions in outline:



[Unseen problem]



[4]

(b) Suppose that Fun is to be extended with a switch-command with multiple cases. For example, the following switch-command contains two cases:

```
switch n :
   case 1: a = 1  b = 2
   case 3: c = 3
.
```

and the following switch-command contains one case and a default:

```
switch n :
   case 1: a = 1
   default: x = 0  y = 0
.
```

In general, the switch-command may contain any number of cases and at most one default. Each case contains a numeral and a sequential command. The default (if any) comes last, and contains a sequential command.

Modify the grammar to allow for switch-commands. You may use either BNF or EBNF.

[Unseen problem]

com = …
| ‘`switch`’ *expr* ‘`:`’ *switch-body* ‘`.`’

switch-body =
| ‘`default`’ *seq-com*
| ‘`case`’ *num* ‘`:`’ *seq-com* *switch-body*

or (in EBNF):

switch-body = (‘`case`’ *num* ‘`:`’ *seq-com*)* (‘`default`’ ‘`:`’ *seq-com*)?

[This answer permits 0 or more cases in a case-body. An answer permitting 1 or more cases is also acceptable.]

[6]
[*total 10*]

com = *id* ‘=’ *expr*
| ‘`if`’ *expr* ‘`:`’ *seq-com* ‘`.`’
| …

seq-com = *com*
| *seq-com* *com*

Box 1 Part of the BNF grammar of programming language Fun (*com* is a command, *seq-com* is a sequential command, *expr* is an expression, *id* is an identifier, *num* is a numeral).

2. (*Concepts*)

(a) Java is a statically-typed language, whereas Python is a dynamically-typed language. Carefully explain the difference between *static typing* and *dynamic typing*.

What are the advantages of static typing, and what are the advantages of dynamic typing?

Illustrate your answer using the following Java method:

```
static int min (int m, int n) {
    if (m < n)
        return m;
    else
        return n;
}
```

and the following Python function:

```
def min (m, n):
    if m < n:
        return m
    else:
        return n
```

[Notes + unseen example]

Static typing: Each variable, function, etc., has a declared type (or a type that can be inferred by the compiler). All operations are type-checked by the compiler. E.g., the Java method `min` is guaranteed to be called with a pair of `int` arguments, and is guaranteed to return an `int` result.

Dynamic typing: Only values have fixed types. A value of any type can be assigned/bound to a variable. Operations are type-checked at run-time. E.g., the Pythin function `min` could be called with any pair of arguments; it will fail if these arguments cannot be compared with '>', otherwise it will return one or other of these arguments.

Advantages of static typing: The program can be verified type-safe by the compiler. There is no need for run-time type-checking instructions or type tags. E.g., the Java method `min` accepts only a pair of integers, and is guaranteed to return an integer without fail.

Advantages of dynamic typing: It is possible to write some programs that are inexpressible in a statically-typed language. E.g., the Pythin `min` accepts any pair of comparable values (though it will fail at run-time if given a pair of incomparable values).

[2+2+1+1]

(b) In what circumstances would the Python function of part (a) fail at run-time? In what circumstances would the Java method of part (a) fail at run-time?

[Unseen problem]

The Python function would fail if passed a pair of arguments that cannot be compared using '<'.

The Java method would never fail (at least not due to a type-error).

[2]

(c) Carefully explain the concept of *encapsulation* in a programming language. Compare and contrast the support (or lack of support) for this concept in Java and in Python.

[Notes]

Encapsulation refers to a program-unit that declares several components, some of which are private (hidden from application code).

The components of a Java class can be individually declared as public, private, or protected. The compiler prevents application code from accessing private or protected components.

The components of a Python module are not encapsulated, but there is a convention that components whose names start with '_' are intended to be private. This convention is not enforced by the compiler.

[6]

(d) Suppose that you are required to design a program unit that maintains a single queue (first-in-first-out sequence) of strings. Application code must be able to add an element to the rear of the queue, to remove the element at the front of the queue, and to test whether the queue is empty. If possible, application code should be prevented from accessing the queue in irregular ways.

Outline such a program unit (i) in Java, and (ii) in Python. For simplicity, assume that the elements are strings. Show declarations of variables, functions, etc., but do not implement them.

How effective is each program unit in preventing application code from accessing the queue in irregular ways?

[Unseen problem]

(i) Java class:

```
class Queue {
   private String[] elems = new String[…];
   private int length = 0, front = 0, rear = 0;
   public void add (String x) {…}
   public String remove () {…}
```

```
    public boolean isEmpty () {…}
}
```

Application code cannot access `elems` or `length`, so will continue to work even if the class is modified.

(ii)Python module:

```
_elems = [ ]
_length = 0
_front = 0
_rear = 0
def  add (x):
   …
def remove ():
   …
def isEmpty ():
   …
```

Application code can access `_elems` and `_length`, but then might no longer work if the module were modified.

[3+3]

[*total 20*]

3. (*Implementation*)

(a) Explain the difference between a *compiler* and an *interpreter*.

[Notes]

A compiler translates source code into lower-level object code.

An interpreter executes source code, one 'instruction' at a time.

[2]

(b) Explain the role of the *syntactic analysis*, *contextual analysis*, and *code generation* phases of a compiler. How do these phases typically communicate with each other?

[Notes]

Syntactic analysis: lexing and parsing the source code, building an AST.

Contextual analysis: scope checking and type checking, using the AST.

Code generation: address allocation and code selection, using the AST.

[4]

The rest of this question concerns the ANTLR compiler generator, which you have seen being used to generate a Fun → SVM compiler.

(c) Box 3a shows parts of a *grammar file*. Explain carefully what ANTLR does with this grammar file.

[Seen example]

ANTLR uses the grammar file to generate a lexer and a parser, which are Java classes named `FunLexer` and `FunParser`.

`FunLexer` is generated from the lexical rules in the grammar file, i.e., those defining `IF`, `ID`, `ASSN`, `COLON`, etc. When run, the lexer will take a Fun source file and translate it to a token stream.

`FunParser` is generated from the context-free rules in the grammar file, i.e., those defining `com`, `seqcom`, etc. It is a modified form of recursive-descent parser that contains a parsing method for each nonterminal, i.e., `com()`, `seqcom()`, etc. When run, the parser will accept a token stream and build an AST, in accordance with the tree-building operations following '`->`' in the grammar file.

[6]

(d) Box 3b shows parts of a *tree grammar file*. Explain carefully what ANTLR does with this tree grammar file.

[Seen example]

ANTLR uses the tree grammar file to generate a code generator, which is a Java class named `FunEncoder`.

`FunEncoder` is generated from the tree patterns and actions in the tree grammar file. It is a depth-first left-to-right tree walker. When run, it pattern-matches the AST and performs the actions '{…}' associated with each pattern. These actions perform address allocation and SVM code selection, including (where necessary) patching of forward jumps.

[8]

(e) Suppose that the Fun language is to be extended with a repeat-command such as the following:

```
repeat :
   p = p*2
   g = g+1
until p < n
```

The syntax should allow any sequential command between '`:`' and '`until`'. The semantics should be to execute the sequential command repeatedly until the expression following '`until`' yields true. (The expression is to be evaluated *after* the sequential command is executed.)

Show how the files of Box 3a and 3b should be modified to achieve this extension.

[Unseen problem]

Grammar file with additions emphasised:

```
grammar Fun

…

com
     : ID ASSN expr                  -> ^(ASSN ID expr)
     | IF expr COLON seqcom          -> ^(IF expr seqcom)
     | …
     | REPEAT COLON seqcom UNTIL expr
                                     -> ^(REPEAT seqcom expr)
     ;

seqcom
     : com*                          -> ^(SEQ com*)
     ;

…

REPEAT : 'repeat' ;
UNTIL  : 'until' ;
IF     : 'if' ;
ID     : LETTER+ ;
ASSN   : '=' ;
COLON  : ':' ;
…
```

Tree grammar file with additions emphasised:

```
tree grammar FunEncoder

…

com
     : ^(ASSN ID expr)       { … }
     | ^(IF expr             { … }
         com)                { … }
     | ^(SEQ com*)
     | …
     | ^(REPEAT              { let c be the address of the next instruction
                             }
         com expr)           { emit an instruction 'JUMPF c'
                             }
     ;

…
```

[4+6]
[*total 30*]

```
grammar Fun
…
com
     : ID ASSN expr                -> ^(ASSN ID expr)
     | IF expr COLON seqcom        -> ^(IF expr seqcom)
     | …
     ;
seqcom
     : com*                        -> ^(SEQ com*)
     ;
…
IF     : 'if' ;
ID     : LETTER+ ;
ASSN   : '=' ;
COLON  : ':' ;
…
```

Box 3a Outline of an ANTLR grammar file.

```
tree grammar FunEncoder
…
com
     : ^(ASSN ID expr)      { let d be the address of variable ID
                              emit an instruction 'STORE d'
                            }
     | ^(IF expr            { let c1 be the address of the next instruction
                              emit an instruction 'JUMPF 0'
                            }
         com)               { let c2 be the address of the next instruction
                              patch c2 into the instruction at address c1
                            }
     | ^(SEQ com*)
     | …
     ;
…
```

Box 3b Outline of an ANTLR tree grammar file.
(For clarity, actions are expressed in English rather than Java.)