Programming Languages 3: Questions and Answers: April/May 2014

Duration: 90 minutes.
Rubric: Answer all three questions. Total 60 marks.

1. (*Syntax*)

Box 1 shows parts of the EBNF grammar of the programming language Fun.

Suppose that Fun is to be extended with arrays. All arrays are to be 1-dimensional, and indexed from 0 upwards. The following program illustrates the required extension:

```
# sum(v) returns the sum of all components of v.
func sum (int[] v):
     int s = 0
     int i = 0
     while i < length(v):
          s = s + v[i]
          i = i + 1
     .
     return s
.

# main() reads a year and write the number of days.
proc main ():
     int year = read()
     int[] size = [31,28,31,30,31,30,31,31,30,31,30,31]
     int feb = 1
     if year/4*4 == year:
          size[feb] = size[feb] + 1 .
     write(sum(size))
.
```

A variable `v` of type '`int[]`' is an array of integers. The construct '`v[i]`' uses the value of `i` to index the array `v`. An expression such as '`[31,28,…,31]`' creates an array.

Modify the grammar to allow for the required extension.

[10]

[Unseen problem]

Grammar with additions emphasized:

prog	=	…	
var-decl	=	…	
type	=	*prim-type*	
	|	***prim-type* '[' ']'**	[2]
prim-type	=	'`bool`'	
	|	'`int`'	

com = *id* ‘=’ *expr*
| ***id* ‘[’ *expr* ‘]’ ‘=’ *expr*** [3]
| ‘if’ *expr* ‘:’ *seq-com* ‘.’
| …

seq-com = …

expr = …

sec-expr = …

prim-expr = ‘false’
| ‘true’
| *num*
| *id*
| ***id* ‘[’ *expr* ‘]’** [2]
| **‘[’ *expr* (‘,’ *expr*) * ‘]’** [3]
| ‘(’ *expr* ‘)’
| …

…

[Restricting the array indexing construct, such that the index is a literal or identifier, will lose 1 mark.]

[Restricting the array creation expression, such that the components are all literals, will be acceptable.]

[Restricting the array creation expression, such that it can occur only in a var-decl, will be acceptable.]

prog = *var-decl* * *proc-decl* + *eof*

var-decl = *type* *id* ‘=’ *expr*

type = ‘bool’
| ‘int’

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

seq-com = *com* *

expr = *sec-expr* ((‘==’ | ‘<’ | ‘>’) *sec-expr*) ?

sec-expr = *prim-expr* ((‘+’ | ‘-’ | ‘*’ | ‘/’) *prim-expr*) *

prim-expr = ‘false’
| ‘true’
| *num*
| *id*
| ‘(’ *expr* ‘)’
| …

…

Box 1 Parts of the EBNF grammar of Fun.
(Here *prog* is a program, *var-decl* is a variable declaration,
com is a command, *seq-com* is a sequential command,

expr is an expression, *prim-expr* is a primary expression,
id is an identifier, and *num* is a numeral.)

2. (*Concepts*)

(a) What is meant by the *lifetime* of a variable?

What is the lifetime of:

(i) a global variable?

(ii) a local variable?

(iii) a heap variable?

[6]

[Notes]

The lifetime of a variable is the time interval between its creation and its destruction. [1.5]

(i) The lifetime of a global variable is the program's entire run-time. [1.5]

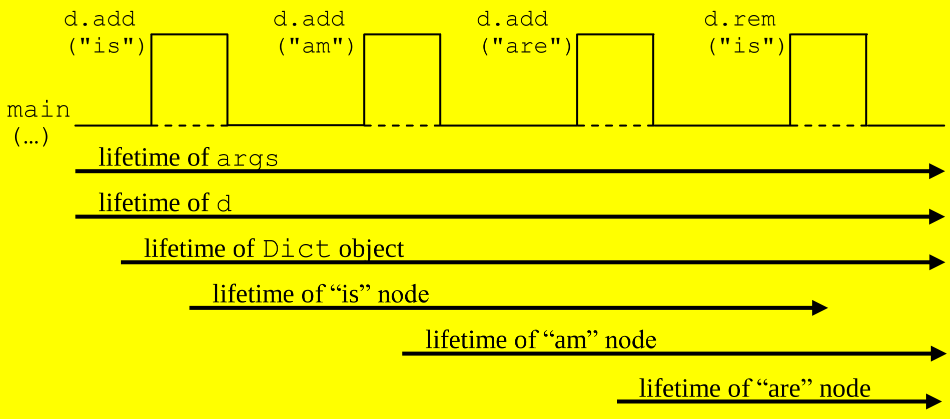
(ii) The lifetime of a local variable is an activation of the block in which it is declared. [1.5]

(iii) The lifetime of a heap variable starts when it is created by an allocator; and finishes when it is destroyed by a deallocator, when it becomes unreachable, or when the program halts. [1.5]

(b) Consider the Java program outlined in Box 2. Draw a diagram showing the lifetimes of all global and heap variables created by this program.

[6]

[Unseen problem]



[1 for each variable]

(c) Briefly explain the general concept of *encapsulation* in programming languages. Why is encapsulation an important concept?

[4]

[Notes]

Encapsulation makes it possible for some components of a program unit (module, package, or class) to be public whilst others are private. "Public" means visible to client code; "private" means visible only inside the program unit. [2]

This is important because it narrows the program unit's interface, and frees the implementer of the program unit to add or remove private components at will, without invalidating client code. [2]

(d) How is encapsulation supported by Java? Illustrate your answer by referring to the Java code of Box 2.

[4]

[Notes + seen problem]

Java supports encapsulation mainly by means of classes in which each component (variable/method) is specified as either public or protected or private. [2]

In the class of Box 2, the components `word` and `rest` are specified as private, whilst `Dict()`, `add()`, `rem()`, and `main()` are specified as public. [2]

```
public class Dict {
     // A Dict object is a dictionary.

     // A dictionary is represented by a sorted
     // linked list of words.
     private String word;
     private Dict rest;

     public Dict () { word = null; rest = null; }

     // add(w) adds word w to this dictionary.
     public void add (String w) {…}

     // rem(w) removes word w from this dictionary.
     public void rem (String w) {…}

     public static void main (String[] args) {
          Dict d = new Dict();
          d.add("is");
          d.add("am");
          d.add("are");
          d.rem("is");
     }

}
```

Box 2 Outline of a Java program.

3. (*Implementation*)

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

[3]

[Notes]

Syntactic analysis: lexing and parsing the source code, building an AST. [1]

Contextual analysis: scope checking and type checking, using the AST. [1]

Code generation: address allocation and code selection, using the AST. [1]

(b) Box 3a shows parts of an ANTLR grammar file. Explain in detail what ANTLR does with this grammar file.

[6]

[Seen example]

ANTLR uses this grammar file to generate a lexer and a parser, which are Java classes named `FunLexer` and `FunParser`. [2]

`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. [2]

`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. [2]

(c) Box 3b shows parts of an ANTLR tree grammar file. Explain in detail what ANTLR does with this tree grammar file.

[6]

[Seen example]

ANTLR uses this tree grammar file to generate a contextual analyser, which is a Java class named `FunChecker`. [2]

`FunChecker` 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 particular actions perform scope checking and type checking, using a type table. [4]

(d) Box 3c shows parts of an ANTLR tree grammar file. Explain in detail what ANTLR does with this tree grammar file.

[6]

[Seen example]

ANTLR uses this tree grammar file to generate a code generator, which is a Java class named `FunEncoder`. [2]

`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 particular actions perform address allocation and code selection, using an address table. [4]

(e) Suppose that the Fun language is to be extended with an additional assignment command such as the following:

```
s += a * b
```

This command should add the value of '`a*b`' to the value stored in the variable s. The syntax should allow an arbitrary expression to the right of '='.

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

[9]

[Unseen problem]

Grammar file with addition emphasized:

```
grammar Fun
…
com
      : ID ASSN expr              -> ^(ASSN ID expr)
      | ID PLUS ASSN expr         -> ^(PLUSASSN ID expr) [2]
      | …
      ;
…
ID      : LETTER+ ;
ASSN    : '=' ;
PLUS    : '+' ;
…
```

Tree grammar file with addition emphasized:

```
tree grammar FunChecker
…
```

```
com
      : ^(ASSN ID
           t2=expr)              { … }
      | ^(PLUSASSN ID
           t2=expr)              { lookup ID in the type table,
                                     and let its type be t1
                                   check that t1 and t2 are both INT
                                 }                                    [3]
      | …
      ;
expr                             returns [Type type]
      : ID                       { … }
      | ^(PLUS
           t1=expr
           t2=expr)              { … }
      | …
      ;
…
```

Tree grammar file with addition emphasized:

```
tree grammar FunEncoder
…
com
      : ^(ASSN ID
           expr)                 { … }
      | ^(PLUSASSN ID            { lookup ID in the address table,
                                     and let its address be d
                                   emit the instruction 'LOAD d'
                                 }
           expr)                 { emit the instruction 'ADD'
                                   emit the instruction 'STORE d'
                                 }                                    [4]
      | …
      ;
expr
      : ID                       { … }
      | ^(PLUS
           expr
           expr)                 { … }
      | …
      ;
…
```

```
grammar Fun
…
com
     : ID ASSN expr                    -> ^(ASSN ID expr)
     | …
     ;
…
ID     : LETTER+ ;
ASSN   : '=' ;
PLUS   : '+' ;
…
```

Box 3a Part of an ANTLR grammar file.

```
tree grammar FunChecker
…
com
     : ^(ASSN ID
          t2=expr)              {  lookup ID in the type table,
                                     and let its type be t1
                                   check that t1 is equivalent to t2
                                }
     | …
     ;
expr                            returns [Type type]
     : ID                       {  lookup ID in the type table,
                                     and let its type be t
                                   set $type to t
                                }
     | ^(PLUS
          t1=expr
          t2=expr)              {  check that t1 and t2 are both INT
                                   set $type to INT
                                }
     | …
     ;
…
```

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

```
tree grammar FunEncoder
…
com
     : ^(ASSN ID
          expr)               {  lookup ID in the address table,
                                    and let its address be d
                                 emit the instruction 'STORE d'
                              }
     | …
     ;
expr
     : ID                     {  lookup ID in the address table,
                                    and let its address be d
                                 emit the instruction 'LOAD d'
                              }
     | ^(PLUS
          expr
          expr)               {  emit the instruction 'ADD'
                              }
     | …
     ;
…
```

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