Programming Languages 3: Questions and Answers: April/May 2009
Duration: 90 minutes.
Rubric: Answer all four questions.

1. (a) Pure functional languages (such as Haskell) are highly expressive, despite lacking the assignments and loops of imperative (and object-oriented) languages. Identify and briefly explain *three* features of functional languages that account for their expressive power.

[Notes + insight; any three of the following points are sufficient]

Recursive functions are effective for expressing repetitive computations.

Pattern-matching is effective for defining functions case-by-case.

Higher-order functions are effective for expressing computational patterns.

Polymorphic functions and types are effective for expressing computations that are (largely) independent of the types of the data involved.

[3]

(b) A *dictionary* (of the kind used by a spell-checker) is a set of words.

(i) Define a Haskell type suitable for implementing a dictionary.

[Unseen problem]

```
    type Dict = [String]
```

[1]

(ii) Write a Haskell function, `lookup w d`, that yields true if and only if dictionary `d` contains word `w`.

[Similar to seen problem]

```
    lookup :: String -> Dict -> Bool
    lookup w [] = False
    lookup w (w':ws) =
      if w == w'
      then True
      else lookup w ws
```

[4]

(iii) Write a Haskell function, `add w d`, that yields the dictionary obtained by adding word `w` to dictionary `d` (or yields `d` if it already contains `w`).

[Similar to seen problem]

```
add :: String -> Dict -> Dict
add w ws =
  if lookup w ws
  then ws
  else w:ws
```

[3]

Explicitly declare the type of each function in (ii) and (iii).

(*Note:* A simple implementation using linear search is acceptable.)

(c) Consider the following Haskell type definition:

```
data Tree a = NULL | NODE a (Tree a) (Tree a)
-- A value of type Tree a is a binary tree whose nodes contain elements of type a.
```

(i) Write a Haskell function, `depth t`, that yields the depth of tree `t`. (*Note*: a tree with a single node has depth 0; an empty tree has depth –1.)

[Similar to seen problem]

```
depth :: Tree a -> Int
depth NULL = -1
depth (NODE x t1 t2) =
  1 + max (depth t1) (depth t2)
```

[3]

(ii) Write a Haskell function, `postorder t`, that yields a list of all elements of tree `t`, using post-order traversal.

[Similar to seen problem]

```
postorder :: Tree a -> [a]
postorder NULL = []
postorder (NODE x t1 t2) =
  postorder t1 ++ postorder t2 ++ [x]
```

[3]

(iii) Write a Haskell function, `mirror t`, that yields the mirror-image of tree `t` (i.e., the tree obtained from `t` by swapping every pair of subtrees).

[Unseen problem]

```
mirror :: Tree a -> Tree a
mirror NULL = NULL
mirror (NODE x t1 t2) =
  NODE x (mirror t2) (mirror t1)
```

[3]

Explicitly declare the type of each function in (ii) and (iii).

2. The following is part of the BNF grammar of a hypothetical programming language:

exp ::= *pexp*
| *exp* + *exp*
| *exp* - *exp*

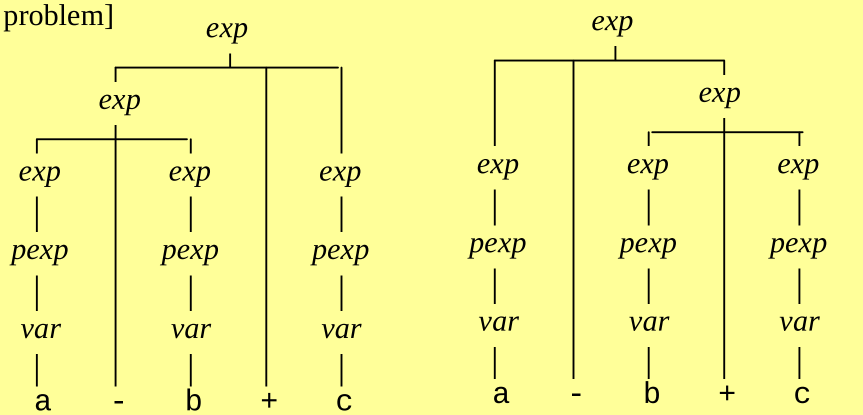
pexp ::= *var*
| (*exp*)

var ::= a | b | c | d | …

(Here *exp* is an expression; *pexp* is a primary expression; *var* is a variable.)

(a) Show that the expression "a-b+c" is *ambiguous* by drawing its two syntax trees.

[Unseen problem]

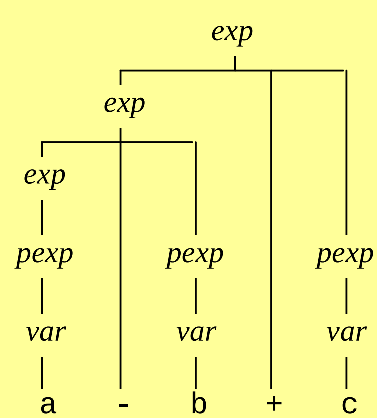


[5]

(b) Show how to modify the grammar to eliminate the ambiguity, in such a way that expressions associate to the left. For example, "a-b+c" should be interpreted like "(a-b)+c". Illustrate your answer by drawing the unique syntax tree of "a-b+c".

[Unseen problem]

exp ::= *pexp*
| *exp* + *pexp*
| *exp* - *pexp*



[5]

3. **(a)** What is an *interpretive compiler*? Why are interpretive compilers useful?

[Notes]

An interpretive compiler consists of:
(i) a compiler that generates virtual machine code, and
(ii) an interpreter for virtual machine code.

An interpretive compiler is useful because [2 of these reasons suffice]:
it reduces the effort of building a compiler;
it speeds up compilation (good for program development);
it makes the compiler portable.

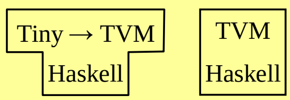
[4]

The remainder of this question is about the Tiny interpretive compiler presented and used in the *Programming Languages 3* course.

(b) Draw tombstone diagrams representing the components of the Tiny interpretive compiler (expressed in Haskell).

[Similar to seen problem]

The interpretive compiler consists of a Tiny → TVM translator plus a TVM interpreter (where TVM = Tiny Virtual Machine code):



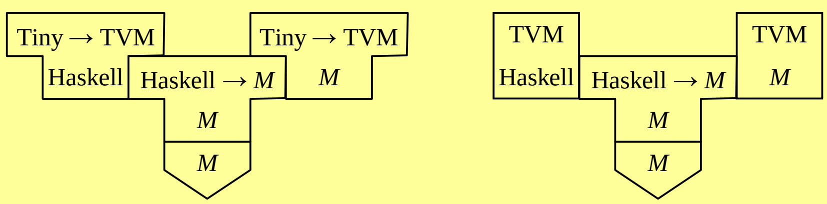
[2]

(c) Draw tombstone diagrams showing how to install the Tiny interpretive compiler on machine *M*. (Assume that a Haskell → *M* compiler is available.)

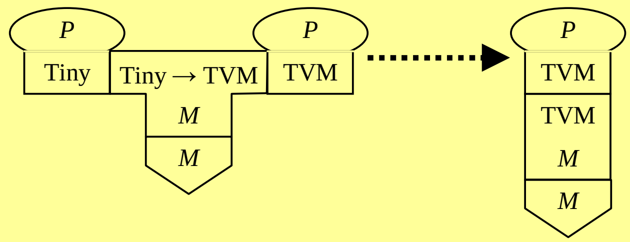
Also show how the interpretive compiler would be used to compile and run a Tiny program *P*.

[Similar to seen problem]

To install the interpretive compiler:



To compile and run a Tiny program *P*:



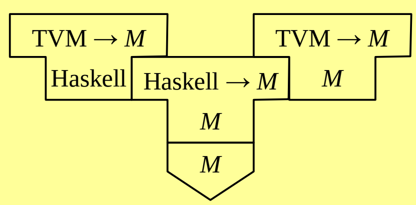
[4]

(d) Suppose now that you are required to build a compiler that will translate Tiny to *M* machine code. Using tombstone diagrams, show how you would build your compiler using the Tiny interpretive compiler components, adding just one new component.

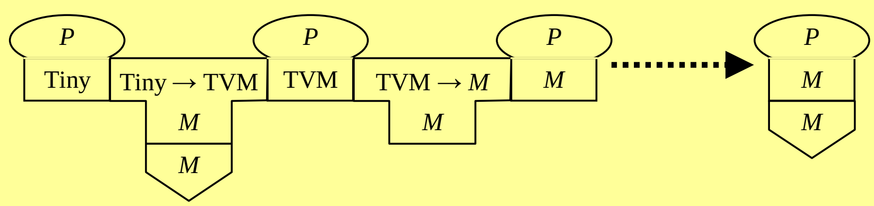
Also show how your compiler would be used to compile and run a Tiny program *P*.

[Similar to seen problem]

Write a TVM → *M* back-end in (say) Haskell, and compile it:



To compile and run a Tiny program *P*:



[10]

4. **(a)** Define *Cartesian products* (×), *disjoint unions* (+), and *mappings* (→). Briefly explain how each concept is relevant to the understanding of programming languages.

[Notes]

Cartesian products: $S \times T$ is a set of ordered pairs whose components are selected from S and T, respectively. This concept underlies records, structs, and tuples.

Disjoint unions: $S + T$ is a set of tagged values, each selected from either S or T. This concept underlies algebraic data types, variant records, and objects.

Mappings: $S \rightarrow T$ is the set of all possible mappings from S to T. This concept underlies arrays and functions.

[6]

(b) Using the concepts of part (a), write equations defining the set of values of each of the following Haskell types:

```
data Piece         = PAWN | KING
data Colour        = WHITE | BLACK
type ColouredPiece = (Colour, Piece)
data Square        = EMPTY | CONT ColouredPiece
type Board         = Int -> Square
```

[Unseen problem]

Piece	=	{*PAWN*, *KING*}
Colour	=	{*WHITE*, *BLACK*}
ColouredPiece	=	Colour × Piece
Square	=	*EMPTY* Void + *CONT* ColouredPiece
Board	=	Int → Square

[5]

(c) Using the concepts of part (a), write an equation defining the set of objects in a Java program that includes the following classes:

```
class Event {
   private Date date;
   private String description;
   …  // methods
}

class Appointment extends Event {
   private Time time;
   …  // methods
}

class Meeting extends Appointment {
   private int location;
   private String[] participants;
   …  // methods
}
```

(Assume that `Date` and `Time` are library classes.)

[Unseen problem]

Object = *Event* (Date × String)
+ *Appointment* (Date × String × Time)
+ *Meeting* (Date × String × Time × Int × (Int → String))
+ …

[5]

(d) Explain the difference between *statically-typed* and *dynamically-typed* programming languages.

[Notes]

In a statically-typed language, every variable and expression has a fixed type. All operands are type-checked at compile-time.

In a dynamically-typed language, values have fixed types, but variables and expressions do not. All operands are type-checked when computed at run-time.

[4]

(e) What are the advantages and disadvantages of static and dynamic typing? Illustrate your answer using the following Java method definition and method call:

```
static int max (int x, int y) {
   if (x > y)
   then return x;
   else return y;
}

int m = …;
int n = …;
int p = max(m, n);
```

and the following Python function definition and function call:

```
def max (x, y) :
   if x > y :
      return x
   else :
      return y

m = …
n = …
p = max(m, n);
```

(Java is statically-typed, whilst Python is dynamically-typed.)

[Notes + unseen example]

In the Java program, the method definition can be type-checked at compile-time, i.e., x : Int and y : Int , so x>y : Boolean, so the if-statement is well-typed; moreover, both return-statements are consistent with the function's type. The method call can also be type-checked at compile-time, i.e., m : Int and n : Int , so max(m,n) : Int . The compiler can guarantee that the program will never fail with a type error at run-time.

In the Python program, every operation must be type-checked at run-time. This is costly in terms of time (type checks) and space (type tags), and there is no guarantee of type safety. The advantage is that the function is more flexible: it can safely be applied to any pair of values that can be compared with “>”.

[10]