88

Quiz 1 CS 381, Spring 2014

Name: Sean Penney

1. Expression Evaluation

Consider the lists `xs = [1,2,3]`, `ys = [4,5]`, and `zs = [6]` as well as the nested list `ll = [xs,ys,zs]`. Evaluate the following expressions. Write "*Error*" if the expression contains a type error or produces a runtime error.
Note: An expression `f x*g y` is parsed as `(f x)*(g y)`. (This is true for any binary operation.)

(a) `head ys:tail xs`

[4,2,3]

(b) `map head ll`

[1,4,6]

(c) `(head . tail) ll`

[4,6]

(d) `(tail . head) ll`

[2,3]

(e) `map tail (tail ll)`

[6]

(f) `reverse (tail ll)`

[zs, ys]

(g) `map reverse ll`

[[3,2,1], [5,4], [6]]

2. Function Definitions

Consider the function `maxL :: [Int] -> Int` to compute the largest element of a list of non-negative integers. (The largest element of an empty list is defined to be `0`.)

(a) Give a Haskell function definition for `maxL`. You can make use of the binary function `max :: Int -> Int -> Int` that computes the larger of two integers.

maxL :: [Int] → Int
maxL [] = 0
maxL (x:xs) = max (x, maxL xs)

(b) Using `maxL`, give an expression that computes the largest element of a nested list of non-negative integers `ll` (of type `[[Int]]`), as defined in question 1.

-6
map maxL (maxL ll)
maxL (map maxL ll)

3. Understanding Functions

Explain in simple words what the following functions compute.

(a) What does the function `f` compute?

```
f :: [a] -> [a]
f xs = f' xs []

f' []     ys = ys
f' (x:xs) ys = f' xs (x:ys)
```

Reverses the list xs

(b) What does the function `g` compute?

```
g []     ys = ys
g (x:xs) ys = x:g xs ys
```

The first list (xs) with the second list (ys) appended

4. Pattern Matching & Recursion

Consider the following two Haskell functions.

```
f [y,x] = y              g (x:y:xs) = y:g (x:xs)
f (x:xs) = f xs          g _ = []
```

Evaluate the following expressions.

(a) `f [1,2,3,4,5]`

4

(b) `g [1,2,3,4,5]`

2 3 4 5 1

-6

[2,3,4,5] -2 (list)

75

Quiz 2

CS 381, Spring 2014

Name: Sean Penney

1. Grammars, Languages & Syntax Trees

(a) Describe in your own words precisely what sentences can be derived from N with the following grammar.

$$N \;::=\; 0N1 \mid \epsilon$$ or empty

any amount of 0s followed by the same amount of 1s

-3

√ (b) Consider the following grammar.

$$P \;::=\; aPa \mid bPb \mid \epsilon$$

For each of the following sentence, determine whether or not they can be derived from P.

	Yes	No
aa	☑	☐
aabb	☐	☑
baab	☑	☐
abab	☐	☑
bbabb	☐	☑
bbaabb	☑	☐

(c) Judge whether each of the following statements is true in general. (*Note:* "A ⇒ B" means that given A , we know B.)

-2

	Yes	No
Derivation ⇒ syntax tree	☑	☐
Syntax tree ⇒ derivation	☐	☑
Nonterminal N ⇒ N's children in a syntax tree	☐	☑
Terminal a ⇒ a's parent in a syntax tree	☑	☒

2. Grammars and Data Types

Consider the following grammar for a language to control a robot that can move around in the plane. The nonterminal $i \in Int$ ranges over integers, the nonterminal a represents robot actions, and the nonterminal d ranges over directions.

```
a  ::= move forward by i
     | turn left
     | turn d
     | a; a
d  ::= north | east | by i degrees
```

Represent the grammar by Haskell data type definitions, and represent a program that moves forward 5 units and then turns 60 degrees as a value built with constructors of these data types.

data A = Move (i) - not defined
| Turn left
| Turn d
| seq [A]

data d = North | East | (i) ← constructor

data move = Int

-15

Seq [Move 5, Turn (60)]

3. Abstract Syntax

Consider the following grammar for describing meeting times (*mtg*) that can be given by either time values or intervals. The nonterminal i represents integers.

```
time ::= i:i
mtg  ::= at time | from time to time
```

Assume that we have defined the abstract syntax for *time* by the following data type definition.

```
data Time = HM Int Int
```

Which of the following abstract syntax definitions for *mtg* are *not correct* and why? *Note:* The following definitions *may* rely on the data type definition for Time, but they don't have to.

(a) `data Mtg = At Time | From Time Time`

Correct

-5

(b) `data Mtg = Time | From Time Time`

not Correct

doesn't specify 'At' before Time

(c) `data Mtg = At Time | From Time To Time`

not Correct.

doesn't need 'To'

(d) `data Mtg = From Time Time | At Time`

Correct

(e) `data Mtg = From Time | At Time Time`

not Correct.

From should have 2 times, At should have 1

(f) `data Mtg = At Int Int | From Time Time`

not Correct.

Should use Time constructor after A

-25

Quiz 3 CS 381, Spring 2014

Name: Sean Penney

1. Expression Language

Consider the following abstract syntax for a variation of the expression language.
The constant `Zero` denotes `0`, and the operation `Succ` computes successors. The operation `Min` returns the smaller of its two arguments, and `IfZero` returns the value of the second expression if the value of the first expression is `0`. Otherwise, it returns the value of the third expression.

```
data Expr = Zero
          | Succ Expr
          | Sum [Expr]
          | IfZero Expr Expr Expr
```

Define the semantics of the expression language as a function `sem` of the following type.

```
sem :: Expr -> Int
```

sem (zero) = 0
sem (Succ x) = 1+ sem x
base case sem (Sum x:xs) = sem x + sem (Sum xs)
sem(IfZero x y z) | x == 0 = sem(~~Succ~~ y)
| x /= 0 = sem(~~Succ~~ z)

-2

-4

2. Time Language

Consider the following abstract syntax for describing times within a 24-hour day. The semantics of any construct of this language is given by the minutes since midnight. For example, the time 8:13am is represented by the semantic value 493.
The constructors `Midnight` and `Noon` denote two constant time values (in minutes). In contrast, the constructor `PM` denotes a time value at noon or in the afternoon, depending on its `Int` argument that represent an hour value. Finally, an expression `Before m t` denotes the time `m` minutes before time `t`. (Note: You can assume that `m` is small enough that it doesn't lead to negative values. You can also assume that the argument for `PM` is between 1 and 12.)

```
type Minutes = Int

data Time = Midnight
          | Noon
          | PM Int
          | Before Minutes Time
```

Define the semantics of the time language as a function `sem` of the following type.

```
sem :: Time -> Minutes
```

sem (Time midnight) = 0
sem(Time Noon) = 12·60
sem (PM x) = 12·60 + x·60
sem (Before m t) = (sem t) − m

3. Move Language

Consider the abstract syntax of a language for describing movements on a two-dimensional grid. The meaning of `JumpTo p` is to immediately go to the position indicated by `p`, regardless of the current position. In contrast, the command `UpBy i` moves from the current position up by `i` units. The operation `Right` yields the position given by 1 unit to the right of the current position. Finally, `Seq m m'` first performs the move `m` and then the move `m'`.

```
type Pos = (Int,Int)

data Move = JumpTo Pos
          | UpBy Int
          | Right
          | Seq Move Move
```

Define the semantics of the language `Move` as a function `sem` that hase the following type.

```
sem :: Move -> Pos -> Pos
```

sem ((Jumpto (x,y)) (x2, y2) = (x, y)
sem ((UpBY x) (x2, y2) = (x2, y2+x) -1
sem ((Right) (x, y)) = (x+1, y)
sem ((Seq a b) (x,y)) = sem a (x,y) . sem b (x,y)

-4

Quiz 4 — CS 381, Spring 2014

Name: Sean Penney

1. Static and Dynamic Typing

Determine the type for each of the following expressions under static and dynamic typing. Note that even and odd are functions of type `Int -> Bool`.

- Assume *strong typing*.
- Make *optimistic type assumptions* for the variable x that make the expression as *type correct* as possible. Mention the type of x if an assumption is needed.

(a) `if False then "Hello" else x`

-6
- *Static:* type error
- *Dynamic:* Int
- *Type of* x: Int

String

✓ (b) `x/(x-x)`

- *Static:* Int
- *Dynamic:* ~~type~~ Error
- *Type of* x: Int

(c) `if x>5 then even else odd`

-2
- *Static:* Bool — Int -> Bool
- *Dynamic:* Bool
- *Type of* x: Int

(d) `tail [x>5,even x]`

-2
- *Static:* [Bool]
- *Dynamic:* [Bool]
- *Type of* x: Int

(e) `if x==[] then x else x+1`

-2
- *Static:* type Error
- *Dynamic:* [~~Int~~] or error
- *Type of* x: [~~Int~~]

(f) `if x then x:[3,4] else [x]`

-6
- *Static:* [Int] error
- *Dynamic:* [Int] error or [Bool]
- *Type of* x: [Int] Bool

2. Properties of Type Systems

Which of the following statements are true?

(a) A language in which a string can be treated as a list of characters is weakly typed.
(b) A language that has only a type `Byte` is weakly typed.
(c) Static typing is less precise than dynamic typing.
(d) Type-correct programs cannot cause runtime errors.
(e) Dynamic typing finds more errors than static typing.

-12

3. Type Checking

Consider the following abstract syntax for a language for non-nested integer lists. With `N` we represent integer constants. The constant `Empty` denotes an empty list, and the operation `Cons` adds an integer (given as the first argument) to a list. We can extract the first element of a list using `Head`, and the operation `Length` represents a function to compute the length of a list.

```
data Expr = N Int | Empty | Cons Expr Expr
          | Head Expr| Length Expr

data Type = Int | List | Error
```

(a) Defince a function `tc :: Expr -> Type` that implements a type checker for this language.

```
tc :: Expr -> Type
tc (N i) = Int
tc(Empty) = Error List -2
tc(Cons x y) = List
tc (Head x:xs) = tc x
tc (Length x) = Int
```

Need to check types of parameters

-16

(b) Check all the expressions that are *not* type correct.

- ☐ `Cons (N 1) Empty`
- ☐ `Cons (Length Empty) Empty`
- ☐ `Head Empty`
- ☑ `Cons Empty (Cons (N 1) Empty)`

-1

-47

100

Name: Sean Penney

1. Call-by-Value/Reference/Value-Result

Illustrate the evolution of the runtime stack from line 3 up until after the assignment on line 8 under different parameter passing schemes.

```
1 { int x := 2;
2   int z := 0;
3   int f(int y) {
4     x := y+1;
5     y := x+1;
6     return (x+y);
7   };
8   z := f(x);
9 }
```

(a) Show the runtime stack evolution under call-by-value.

[f:{}, z:0, x:2]
[y:2, f:{}, z:0, x:2]
[y:2, f:{}, z:0, x:3]
[y:4, f:{}, z:0, x:3]
[res:7, y:4 f:{}, z:0, x:3]
[f:{}, z:7, x:3]

(b) Show the runtime stack under call-by-reference.

[f:{}, z:0, x:2]
[y→x, f:{}, z:0, x:2]
[y→x, f:{}, z:0, x:3]
[y→x, f:{}, z:0, x:4]
y→x [res:8, f:{}, z:0, x:4]
[f:{}, z:8, x:4]

(c) Show the runtime stack under call-by-value-result.

[f:{}, z:0, x:2]
[y:2, f:{}, z:0, x:2]
[y:2, f:{}, z:0, x:3]
[y:4, f:{}, z:0, x:3]
y:4 [res:7, f:{}, z:0, x:4]
[f:{}, z:7, x:4]

2. Call-by-Name vs. Call-by-Need

Illustrate the evolution of the runtime stack from line 2 up until after the assignment on line 7 under different parameter passing schemes.

```
1 { int y;
2   y := 4;
3   { int f(int x) {
4       y := 2*x;
5       return (y+x);
6     };
7     y := f(y+3);
8   };
9 }
```

(a) Show the stack evolution under call-by-name.

[y:4]
[f:{}, y:4]
[x:y+3, f:{}, y:4]
[x:y+3, f:{}, y:14]
[res:31, f:{}, y:14]
[f:{}, y:31]

(b) Show the stack evolution under call-by-need.

[y:4]
[f:{}, y:4]
[x:y+3, f:{}, y:4]
[x:7, f:{}, y:14]
[res:21, f:{}, y:14]
[f:{}, y:21]

Quiz 7

CS 381, Spring 2014

(a)

Name: Sean Penney

1. Prolog Goals and Predicates

Consider the definition of the predicate `plays/2`.

```
plays(john,piano).    plays(jill,cello).
plays(john,cello).    plays(mike,piano).
```

Based on `plays/2`, answer the following questions.

(a) Write a goal that finds all people who play cello.

plays(x, cello)

(b) Define a predicate `duet/2` that finds two people that can play a duett with the same instrument.

duet(x,y) :- plays(x,i), plays(y,i),
x \= y

-4

(c) Define a predicate `talent/1` that yields true for people that can play more than one instrument.

talent(x) :- plays(x,a), plays(x,b), a\=b

2. Multiple Predicates

The following facts given by the predicate `answer/3` describe how different students have answered numbered multiple-choice questions. The predicate `key/2` shows the correct answer for each question.

```
answer(john,1,b). answer(mary,1,c).  key(1,c).
answer(john,2,a). answer(mary,3,b).  key(2,a).
answer(john,3,b). answer(tim,3,c).   key(3,b).
```

These predicates are the basis for the following questions.

(a) Define a predicate `correct/2` that determines for a given question number all the students who have answered that question correctly.

correct(x,s) :- answer(s,x,a), key(x,a)

(b) Define a predicate `discuss/2` that produces pairs of students who have answered a particular question differently.

cuss(x,y) :- answer(x,n,a), answer(y,n,b), a\=b

(c) How many answers will Prolog produce for the goal `discuss(X,Y)?`

3, assuming pairs are ~~not~~ duplicated

6

3. Recursion

The following predicate `conn/3` describes different forms of connections in a transportation network. For example, the fact `conn(2,b,4)` states that nodes 2 and 4 are connected via `b` (which could mean, for example, "bus" but that doesn't matter).

```
conn(1,a,2).    conn(2,a,3).    conn(3,b,2).
conn(1,b,3).    conn(2,b,4).
```

(a) Define a predicate `path/3` that finds paths in the transportation network labeled by one specific kind of connection. For example, `path(1,a,3)` should yield `true`.

Path(x,P,y) :- Conn(x,P,Z), Conn(Z,P,Y)

-5

(b) Using `path/3`, write a goal that finds all labels for which a path exists from node 1 to node 4.

Path(1,x,4)

(c) Write a goal that finds all nodes that can be reached from node 1 by only traversing edges with label a.

path(1,a,x)

(d) Write a goal that finds all nodes from which there is a path to node 3 with label a *and* with label b.

path(x,a,3), Path(x,b,3)