Exercise 5

NTNU

TDT4165 fall 2018

All multiple choice answers should also contain a very brief explanation of why you chose the answer you did.

1 Grammar

Task 1

Given the grammar G:

```
<S> ::= <A> | <B>
<A> ::= a <A> b | a <C> b | ϵ
<B> ::= a <B> b b | ϵ
<C> ::= c <C> c | ϵ
```

(Where ϵ is the empty token.)

Which of the following strings are not valid in L(G)?

a) abb

b) aaccbb

c) caabbc

d) aabbbb

Task 2

What is the most specific class in the Chomsky hierarchy that this formal grammar belongs to?

```
<S> ::= <Z> | <Y>
<Z> ::= z <Z> y | z <Y> y | ϵ
<Y> ::= z <Y> y x  | ϵ
```

(Where ϵ is the empty token.)

a) context-sensitive

b) regular

c) unrestricted

d) context-free

Task 3

An interpreter...

a) reads a sequence of characters and outputs a sequence of tokens.

b) translates a sequence of characters into a sequence of low-level instructions that can be executed on a machine.

c) reads a sequence of tokens and outputs an abstract syntax tree.

d) reads program code input as text and evaluates and prints the result of executing the code

e) traverses the syntax tree and generates low-level instructions for a real machine or an abstract machine.

Task 4

A parser...

a) reads a sequence of characters and outputs a sequence of tokens.

b) translates a sequence of characters into a sequence of low-level instructions that can be executed on a machine.

c) reads a sequence of tokens and outputs an abstract syntax tree.

d) reads code input as text and evaluates and prints the result of executing the code

e) traverses the syntax tree and generates low-level instructions for a real machine or an abstract machine

Task 5

Given the grammar

```
<expression>    ::= <integer>
                  | ( <expression> <operator>
                      <expression> )
<operator>      ::= + | - | * | /
<integer>       ::= <nonzero digit> { <digit> }
<digit>         ::= 0 | <nonzero digit>
<nonzero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

How many parse trees can be generated for `1+2*3` with this grammar?

a) 1

b) 2

c) 3

d) 4

2 Scope and namespaces

Task 6

A namespace. . .

a) is a problem that develops if the same identifier is used to refer to two values in a region of code where both bindings are reachable

b) is the section of code where a binding is reachable

c) is a name that can be bound to an entity, e.g. types, values or functions

d) is an environment or abstract container that holds the set of bindings available at a given point in a program

Task 7

A scope. . .

a) is a problem that develops if the same identifier is used to refer to two values in a region of code where both bindings are reachable

b) is the section of code where a binding is reachable

c) is a name that can be bound to an entity, e.g. types, values or functions

d) is an environment or abstract container that holds the set of bindings available at a given point in a program

Task 8

```
a = 5

func :: Show a => a -> String
func a = show a
```

What is most correct about the code above?

a) the identifier **a**, which is bound to 5, has a global scope

b) the code will not compile because of a name collision

c) the identifier **a** is not in the namespace inside the function **func**

d) the scope of the identifier **a**, which is bound to 5, is the first two lines of the program

Task 9

In **letter.hs**, three letter types are defined: Large, Medium and Small. How is it possible for them to have the same names in their record fields? Hint: Use **:set -ddump-simpl** in GHCi, load the file and locate the desugared **changeStatus** in the dump.

3 Exception handling

Partial functions

A **partial function** is a function that is undefined for some of its valid input. A function that returns a valid output for all possible input, is called a total function.

```
--partial function
(!!) :: [a] -> Int -> a
(!!) (x:xs) n
    | n == 0    = x
    | otherwise = (!!) xs

--total function
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Our index function will fail if the list is empty, or if we try to retrieve an element on an index larger than the list size. **map**, on the other hand, will always return a valid output.

Task 10

```
(!!) :: [a] -> Int -> a
(!!) [] _ = error "Index too large"
(!!) (x:xs) n
  | n < 0     = error "Negative index"
  | n == 0    = x
  | otherwise = (!!) xs (n-1)
```

The edited index function above...

a) is a total function

b) is a partial function

c) is neither

Task 11

```
(!!) :: [a] -> Int -> Maybe a
(!!) [] _ = Nothing
(!!) (x:xs) n
  | n < 0     = Nothing
  | n == 0    = Just x
  | otherwise = (!!) xs (n-1)
```

The function above...

a) is a total function

b) is a partial function

c) is neither

Task 12

```
undefined :: a
undefined = error "This value is undefined"
```

The value above...

a) is a total function

b) is a partial function

c) is neither

Task 13

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n
    | n < 0     = 0
    | otherwise = n * factorial (n-1)
```

The function above...

a) is a total function

b) is a partial function

c) is neither

Task 14

We can use the type **Either** to return one of two types from a function.

```
data Either a b = Left a | Right b deriving (Show)

factorial :: Integer -> Either a Integer
factorial n
    | n < 0      = Left $ error "Factorial of negative
        number"
    | otherwise  = Right $ fact' n
                     where
                   fact' :: Integer -> Integer
                   fact' 0 = 1
                   fact' n = n * fact' (n-1)
```

What is true about the two functions above?

a) **factorial** is a partial function, because it contains a partial function **fact'**

b) **factorial** is a total function, because it always returns a valid output and does not cause a runtime error

c) **factorial** is a partial function, because it returns an error, which is equivalent to **undefined**

d) **fact'** is not a partial function

Task 15

The type system can also be used to create more specific types, making sure that we only pass the values that we want to our functions. In **natfib.hs** we have created a natural number type, ensuring that our function **fib** only receives positive integers. **Provide at least one positive and negative property of this implementation.**

```
--usage
> toInteger . fib $ fromInteger 6
8
```

For the curious student (not curriculum): Languages like Agda, Coq and Idris support a concept called *dependent types*, which allows us to create extremely expressive types. Examples include *a couple of integers where the first is larger than the second* and *a list that is not infinite* (a vector). Examples in Idris can be found here: `http://docs.idris-lang.org/en/latest/tutorial/typesfuns.html#dependent-types`

Task 16

In exercise 4, we used a type TokErr that circumvents the type system. Remove this Token type and rewrite the functions, using Maybe, Either or both. Maybe and Either are in Prelude and imported by default. Make sure none of your functions are partial.
Hint: Try to minimize the amount of code affected. Functions like **fromJust** (a partial function!) from Data.Maybe and **rights/lefts** from Data.Either could help you with this. These functions need to be imported explicitly.