Session 11: Course summary and some key points

COMP2221: Functional programming

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Reminder

Exam I

Exam assesses

- *knowledge* and *comprehension*: how do things work in Haskell, why do they work, ...
- *application*: what does some code do; can you write code to solve problem X...
- *evaluation*: what are the concepts; what properties does some solution have...

Remarks

- Practice via problem sheets (will cover programming knowledge)
- Types are important: *always write types in code*
- Theory, methodology, concepts from lectures are also relevant
- Please use exact terminology (definitions)

Normally exams at Durham are "closed book".

This year, "open book".

closed book exams: some fraction of marks "bookwork".

open book exams: → no bookwork.
type questions.

→ consequence: a little more applied and synthesis of concepts from course.

2hr paper with 24 hr turnaround.

→ I think 2 hrs work is sufficient to do well.

If could be nice if your code answers are typed (but no requirement).

Exam II

By its nature, cannot be exhaustive.

Past and model papers a good guide. Broadly they cover these types of questions:

Have your editor open and write the code in that.

- Can you write/read (short) Haskell functions? Type annotations, class constraints, pattern matching, guard expressions, conditionals.
- Can you use list-based functions from the standard library? `head`, `tail`, `length`, `map`, comprehensions, ...
- Can you explain/define key terms? Types of polymorphism, currying, side effects, higher order functions, ...
- Can you explain/describe differences in different programming paradigms? Functional/imperative, pure/impure (side effects/side effect free), lazy evaluation lazy/strict, ...
- Can you implement/describe simple type class interfaces and their utility? Properties and requirements of the builtin type classes we covered `Num`, `Ord`, `Functor`, ...

Topics

- Functional vs. imperative, syntax
- Builtin types, function types
- Syntax: conditional expressions, guard equations, pattern matching
- Polymorphism: parametric ("generic functions") vs. method overloading/subclassing. Class constraints

  ```
  (+) :: Num a => a -> a -> a
  ```
- Lists and pattern matching, list comprehensions.
- Recursion classification, writing recursive functions
- Maps and folds, higher order functions, `foldr`, `foldl`
- User-defined data types `data`
- More type classes `Functor` (mappable things), `Foldable`
- Evaluation strategies, lazy evaluation

Concepts

We covered various concepts and structured ideas for programs and types.

Can you explain, or describe, how these might help with (or hinder) writing correct programs?

Some common errors

- Recursion without a base case

```
reverse' :: [a] -> [a]
-- Missing equation for empty list
reverse' (x:xs) = (reverse' xs) ++ [x]
```

- Incorrect syntax when pattern matching lists

```
reverse' :: [a] -> [a]
reverse' [] = []
-- Not a valid pattern, use (x:xs)
reverse' [x:xs] = ...
```

Editor will catch this

- Patterns or guard equations in wrong order

```
sign :: Num a => a -> Int
sign a | a == 0    = 0
       | otherwise = -1
       -- This case never reached
       | a > 0     = 1
```

```
not' :: Bool -> Bool
not' _     = False
-- Never reached, _ matches everything
not' False = True
```

GHC will complain here if you make code.

Type classes

- Basic type classes `Eq`, `Num`, `Ord`, ...capture simple properties of types. Used to provide interfaces
- More complex properties are also captured by a sequence of type classes. We saw `Functor` for mappable types and `Foldable` for foldable types.
- Important when implementing instances that the methods you implement obey the required rules (e.g. Functor laws).

$\Rightarrow$ often done by showing (proving) that your implementation obeys them.

Relevant past papers

Via DUO-library past papers.

2020	Q1 and Q2
2019	Q2 (the single Haskell question)
2018	Q1 (c–e, g) (not (a), (b), (f))
2017	Q1 and Q2. These are mostly programming questions that should be doable if you have looked at the practicals
2016	Q1 (a, c, e, g, h), Q2 (a, b, d, e)

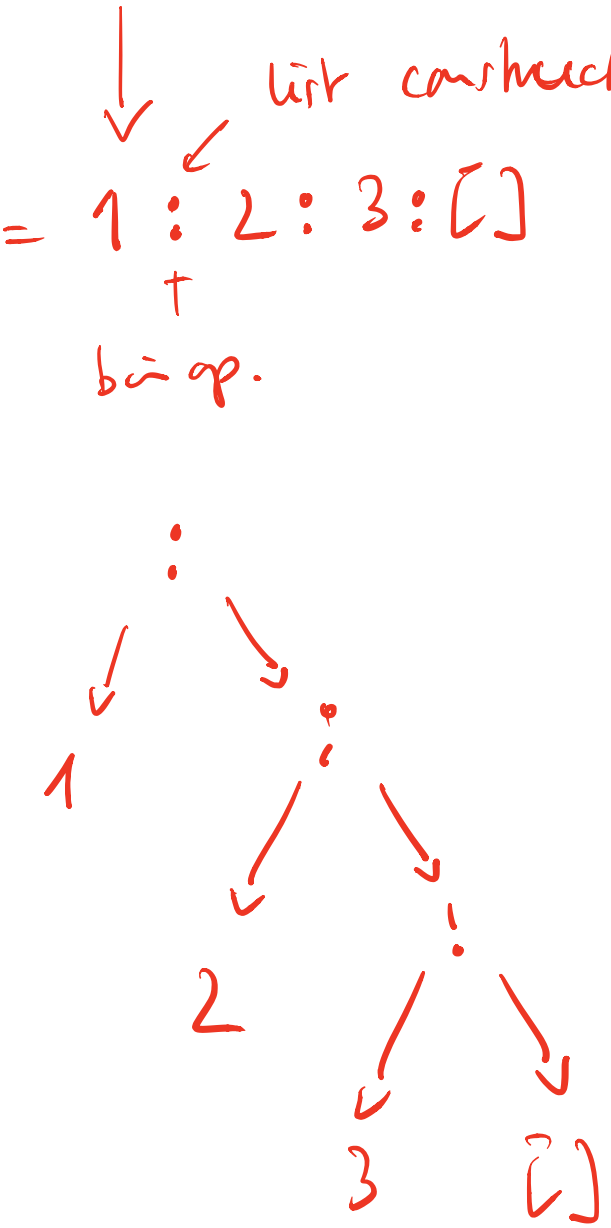
Note that compared to previous years, this year there are fewer bookwork questions (see mock paper for some examples).

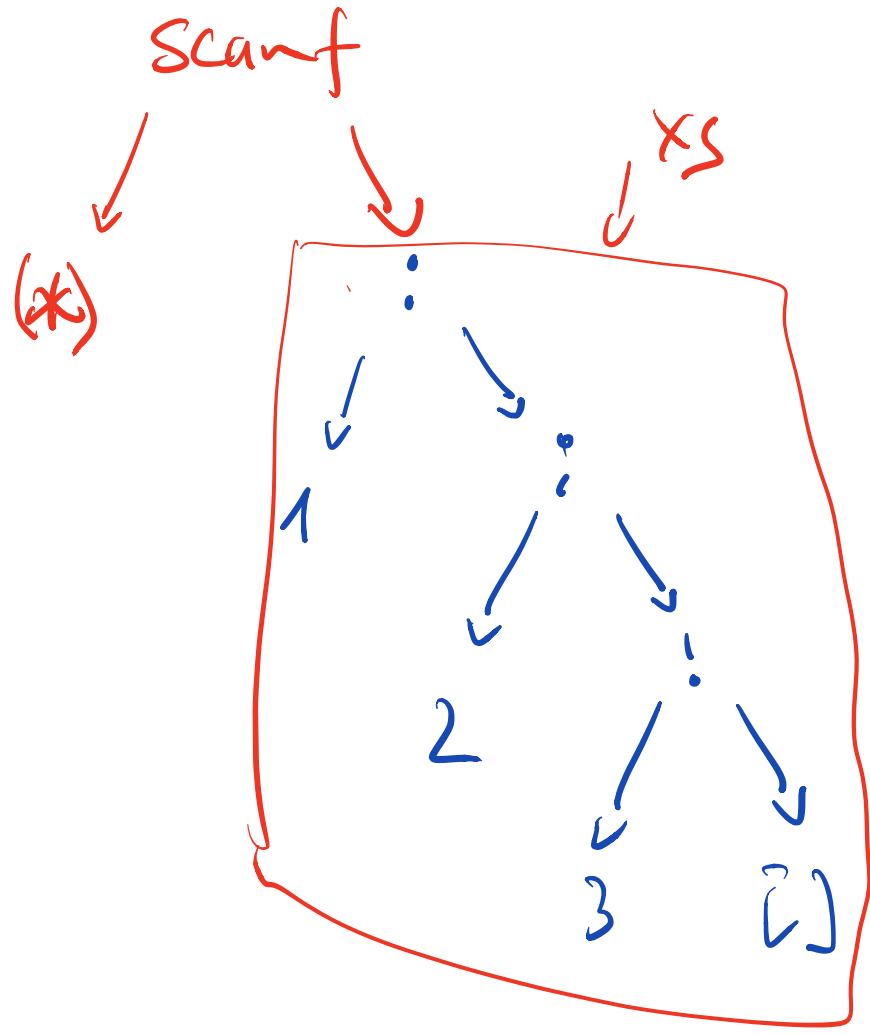
Code in your exam answers.

- Is a screenshot fine?
  Yes if it is readable.
- Ideally copy-pastable is best
  But whatever works for you.

xs = [1, 2, 3]

scanf (*) xs





Normal form:

→ expression graph has

- no cycles
- no reducible expressions (redexes)

No cycles:

[1, 2, 1, 2, 1, 2, ...

Not normal form.

No redexes

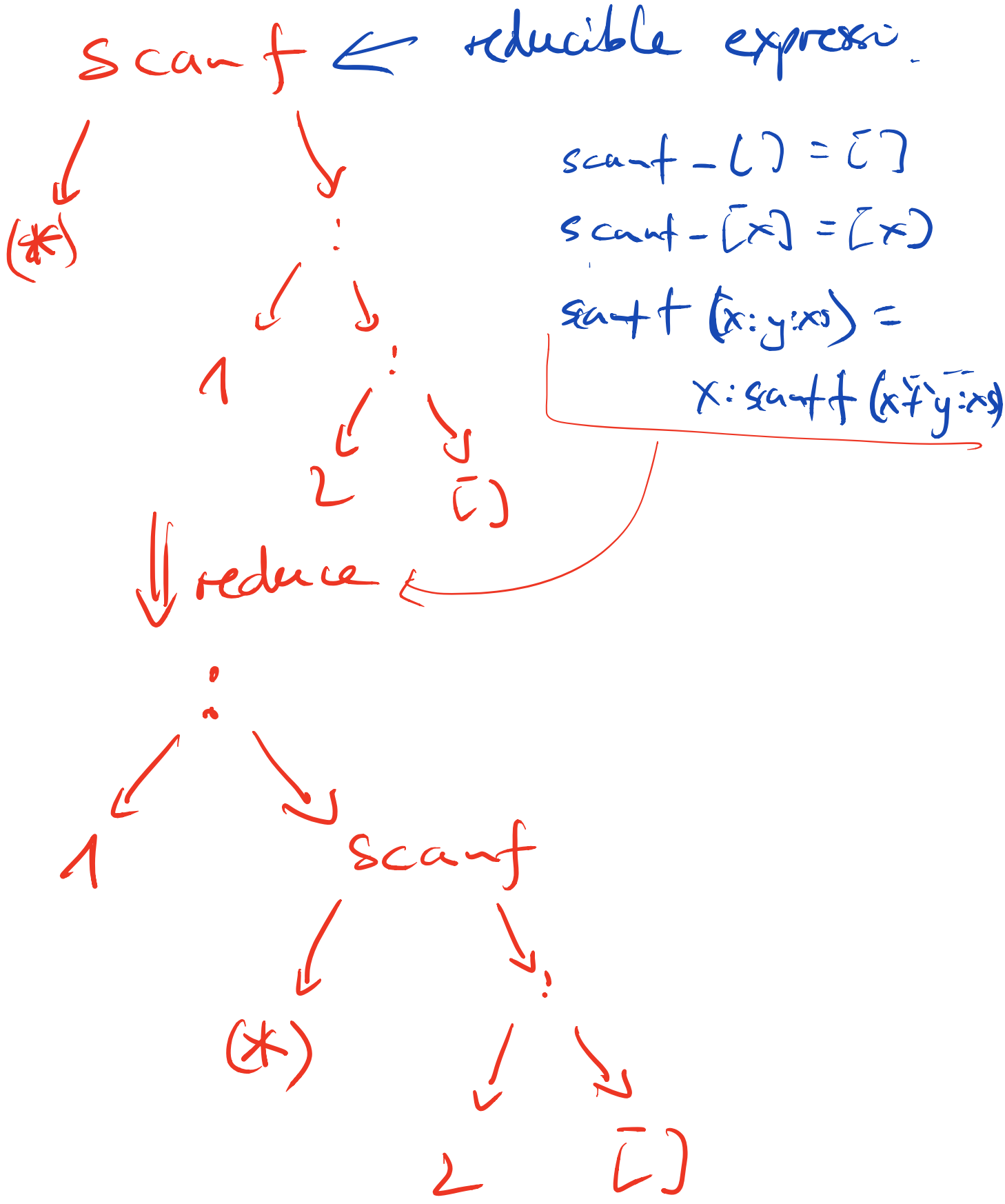
Graph contains no nodes which are functions.

function: introduces a substitution rule.

Data constructors like (:)

or True, Nothing,

do not introduce substitution rules


scanf ← reducible expressi.
scanf _ [] = []
scanf _ [x] = [x]
scanf (x:y:xs) =
x : scanf (x `f` y : xs)
(*)
:
1
:
2
[]
reduce
:
1
scanf
(*)
:
2
[]

Weak Head Normal form (WHNF)

→ Either graph is in normal form

→ Or, the head of graph (top node) is a data constructor.