INFOB3TC - Final Exam

David van Balen, Lawrence Chonavel

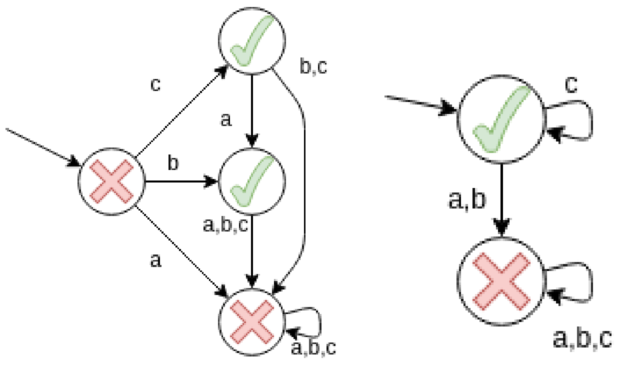
1st February, 13:30 - 16:00 (extra time: 16:30)

Preliminaries

- Write your name and student number on every page you hand in.
- The total amount of points is 85.
- Try to give concise answers, and write legibly.
- Please answer all questions in English.
- You may use any notation, theories and lemmas covered in the slides or lecture notes.
- The appendices have additional information and examples on a few of the languages we use in the exam.

1 (15p total) Finite State Machines

Consider the following two DFAs, which match some regexps r1 and r2 respectively:



1.1 (2p) Draw an NFAε that matches the regexp r1r2

1.2 (8p) Draw a DFA that matches the regexp r1r2

1.3 (5p) Draw a DFA that matches the regexp r3

r_3 = `((a|b)+)(c?)(d*)`

2 (10p total) Fold and Algebra for a Binary Search Tree

Suppose we have the following definition of a tree, which holds values of two different types:

```
data Tree a b = Node (Tree a b) a (Tree a b)
              | Leaf b
```

This tree type can be used to implement a `Map` (also known as a `Dictionary`):

```
type Map key value = Tree key (key,value)
```

The idea is that the leaves store `(key,value)` pairs, and `lookup` is implemented using binary search. In order to make the lookup efficient, we impose an order on the tree: For each `Node left x right`, the `left` subtree should contain the `(key,value)` pairs where `key <= x`, and the `right` subtree should contain the `(key,value)` pairs where `key > x`.

Here is an example:

```
-- A list of (key,value) pairs
list = [(10,a),(2,b),(5,c)]

-- A corresponding binary search tree
dict = Node
        (Node
          (Leaf (2,b))
          3
          (Leaf (5,c)))
        5
        (Leaf (10,a))
```

2.1 (5p) Write the algebra type and fold function for the `Tree` type

2.2 (5p) Write an algebra to implement `lookup`

Your `lookup` function should return `Just` a value matched by the key, if one exists, and otherwise return `Nothing`.

The `lookupAlgebra` should fit in the following template:

```
lookup :: (Ord k, Eq k) => k -> Map k v -> Maybe v
lookup key dict = foldTree lookupAlgebra dict key
```

Hint: `(foldTree lookupAlgebra dict) :: k -> Maybe v`

3 (20p total) Compiler passes

Consider a compiler with a nanopass architecture. One of its optimization passes eliminates loops that have a fixed iteration size. For example, it would convert the following for-loop

```
for(int i = 0; i<5; i++)
  f(i);
```

into this equivalent code:

```
f(0); f(1); f(2); f(3); f(4);
```

N.B. This pass only recognizes a very specific pattern: `int i = 0; i < n; i++` where n is a literal number.

3.1 (10p) Optimization

3.1.1 When is the pass *safe* (i.e. semantics-preserving)?

3.1.2 When & how might the pass *improve* the code?

3.1.3 When & how might the pass *degrade* the code?

3.2 (10p) Nanopasses

Consider the loop-elimination optimization pass described above.

You have been tasked with adding this optimization pass to an existing compiler.

The existing compiler has the following passes:

- Parsing
- Loops to Jumps
- Loop-Invariant Code Motion
- Unroll all loops exactly 32 times
- Type-checking

For each of the passes already in the compiler…

a. State whether it should come *before* or *after* the loop-elimination pass (or if it doesn't matter), in order to improve the code

b. Explain your answer to (a)

4 (10p total) Compiler Checks

Consider the following language:

```
Stmt ::= int Var; Stmt
       | Var = Func ( Var? ) ; Stmt
       | return Var ;
       | if (0 ≤ Var) { Stmt } else { Stmt } ; Stmt
       | ε
```

```
type Func = String
data Stmt = Decli Var Stmt
          | Asign Var Func (Maybe Var) Stmt
          | Retrn Var
          | IfNat Var Stmt Stmt Stmt
          | Empty

data StmtAlgebra r = SAlg
  { decli :: Var -> r -> r
  , asign :: Var -> Func -> Maybe Var -> r -> r
  , retrn :: Var -> r
  , ifnat :: Var -> r -> r -> r -> r
  , empty :: r
  }

foldStmt :: StmtAlgebra r -> Stmt -> r
```

4.1 (10p) Implement `checkScope`

Users of the language are complaining that it's too `bash`-like. In particular, they don't like how "out of scope" errors can occur at run-time. For example, the following program would crash at run-time:

```
int input;
input = get_input();
if (0 ≤ input) {x = sqrt(input);}
          else {x = not_a_number();}
return x;
```

To satisfy the users, you are tasked with writing a *scope checker*, to move "out of scope" errors from run-time to compile-time. Implement a scope-checking algebra `scopeAlg` (in pseudo-Haskell), which finds the free variables in a statement:

```
type Env = [Var]
checkScope :: Stmt -> Env -> Valid [Var] ()
checkScope = foldStmt scopeAlg
```

Hint: recall from the lectures that

```
data Valid e r = Err e | OK r
instance Functor     (Valid es)  -- provides (<$>), (<$), fmap
instance Applicative (Valid [e]) -- provides (<*>), (<*), (*>), pure
instance Monad       (Valid es)  -- provides (>>=), return, guard
```

5 (30p total) Regular, Context-Free, or Neither

Consider the following three languages:

L1 The language of the names of all employees at the UU

L_1 = `{ David van Balen, Lawrence Chonavel, ...}`

L2 The language of matching parentheses (aka LISP).

Each opening or closing bracket must have a unique match, and we also have the character 'a'.

L_2 = `{ w | w ∈ {a,(,)}*, matchingParens(w)}`

(see sec. 6 for examples)

L3 The language of *well-formed markdown code blocks*,

$$
\begin{aligned}
L_3 = \{\ & \texttt{`}^m s \texttt{`}^m \\
& \mid m \in \mathbb{N} \land m > 0 \\
& \land s \in \mathbb{A}^+ \\
& \land \mathrm{head}(s) \neq \texttt{`} \land \mathrm{last}(s) \neq \texttt{`} \\
& \land \neg\ (s \text{ contains the substring } \texttt{`}^m)\ \}
\end{aligned}
$$

Where $\mathbb{A}$ is the alphabet of ASCII characters

(see sec. 7 for examples)

5.1 (5p) Prove that L1 is Regular

5.2 (5p) Prove that L2 is Context-Free

5.3 (10p) Prove that L2 is not Regular

Hint:

```
  ∀ L where L is regular,
    ∃ n ∈ ℕ, such that
      ∀ xyz ∈ L where |y| ≥ n,
        ∃ uvw where |v| > 0, such that
          ∀ i ∈ ℕ, xuv^i wz ∈ L
```

5.4 (10p) Prove that L3 is not Context-Free.

Prove that `L` is **not** context free, by completing the following (unfinished) proof, writing a proof for each case:

```
We prove that L is not context-free, using the pumping lemma for context-
free languages:

  ∀ context-free L,
    ∃ n ∈ ℕ,
      ∀ z ∈ L with |z| ⩾ n,
        ∃ u,v,w,x,y where z = uvwxy ∧ |vx| > 0 ∧ |vwx| ⩽ n,
          ∀ i ≥ 0, uv^i wx^i y ∈ L

Let n be a natural number.
We choose the word z = `3n_`2n_`3n, which is in L and longer than n.

Let z=uvwxy be a splitting of our word, where |vx| > 0 and |vwx| ⩽ n.

Now consider which part of `3n_`2n_`3n
the strings v and x come from.

Case 1: v and x both fit completely into the middle `2n

Case 2: v or x contains an _

Case 3: v or x overlaps with one of the `3n parts
```

6 Appendix: Extra information about the language of *matching parentheses*

The alphabet of *matching parentheses* consists of the characters ‘a’, ‘(’, and ‘)’. Each opening bracket is matched with exactly one closing bracket that follows it, and each closing bracket is matched with exactly one opening bracket that precedes it.

6.1 These are words with matching parentheses:

✅ a

✅ ()a()

✅ ε

✅ a(((a))a)(a())

6.2 These are not words with matching parentheses:

❌ (

❌)a(

❌ ((a)

7 Appendix: Extra information about the language of *well-formed markdown code blocks*

The language of *well-formed markdown code blocks* consists of arbitrary code (i.e. a non-empty ASCII string), enclosed in some positive number n of backtick characters `, with the additional restriction that the code may not contain any n-or-longer sequences of backtick characters `

7.1 These are well-formed markdown code blocks:

✅ ``int main (void)``

✅ `````````` <- trust me, I counted these -> ``````````

✅ `````Look, a code block: ``int main (void)`` (inside a `code` block!)`````

✅ ``Did you know that `LaTeX quotes' use backticks?``

7.2 These are not well-formed markdown code blocks:

❌ ```` I have 4 opening backticks, but only 3 closing backticks ```

❌ ` I have 1 opening backtick, and 5 closing backticks ````

❌ ``I have too many backticks: ```` in the code!``

❌ `I also have `too many` backticks inside the code!`