INFOB3TC - Final Exam

David van Balen, Lawrence Chonavel

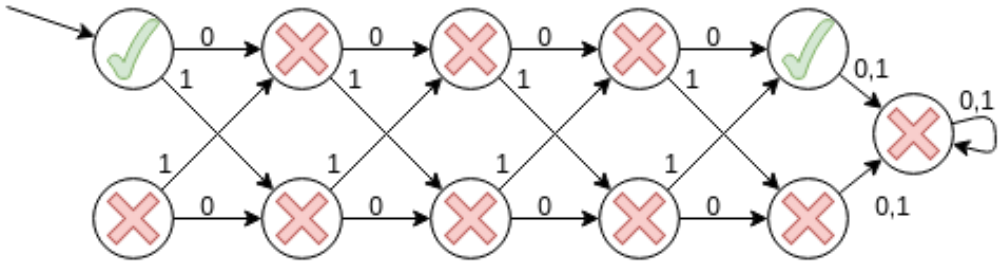
30 January 2025 13:30 - 16:00 (extra time: 16:30)

Preliminaries

- Write your name and student number on every page you hand in.
- The total amount of points is 80.
- 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 DFA, which matches some regexp r1:



1.1 (2p) Draw an NFAε that matches the regexp r1+

1.2 (8p) Draw a DFA that matches the regexp r1+

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

`(ab|ba)+`

2 (5p) Fold and Algebra for State

Suppose we have the following definition of a State datatype, which encodes a stateful computation:

```
data State s a = Get (s -> State s a)
               | Put s (State s a)
               | Return a
```

For example, the following value encodes a state computation that increments the state by one and then returns its new value:

```
incr :: State Int Int
incr = Get (\i -> let j = i+1
                  in Put j (Return j))
```

Write the algebra type and fold function for this State datatype:

```
data StateAlg ... = ...

foldTree :: ...
foldTree ... = ...
```

3 (20p total) Compiler passes

Consider a compiler with a nanopass architecture. One of its optimization passes is called “loop unswitching”. This pass splits a loop that contains a conditional, into a conditional containing two loops.

For example, it might convert the following for-loop

```
for(int i = 0; i < n; i++){
  foo1
  if (condition) {
    foo2
  } else {
    foo3
  }
  foo4
}
```

into the following code:

```
if (condition) {
  for(int i = 0; i < n; i++){
    foo1
    foo2
    foo4
  }
} else {
  for(int i = 0; i < n; i++){
    foo1
    foo3
    foo4
  }
}
```

3.1 (10p) Optimization

3.1.1 When is the pass *safe* (i.e. semantics-preserving), and when is it not?

3.1.2 When & how might the pass be *beneficial* (i.e. good)?

3.1.3 When & how might the pass *detrimental* (i.e. bad)?

3.2 (10p) Nanopasses

Consider the loop unswitching optimization pass described above.

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

The existing compiler has the following passes (here in alphabetical order):

- Converting loops to conditional jumps
- Loop fusion
- Parsing
- Scope-checking
- Strength reduction (a peephole optimization that replaces specific instances of expensive operations with cheap ones, e.g. "division by 2" to a bitshift)

For each of the passes already in the compiler…

a. State whether it should come *before* or *after* the loop-elimination pass, or that it doesn't matter, or that it would be best to run it both before and after.

b. Explain your answer to (a)

4 (10p) Validation

Consider the following language

```
Type ::= bool | float | int

Stmt ::= ε
     |   Type Var = Exp; Stmt
     |   while (Exp) { Stmt } Stmt

Exp ::= Int
    |   Float
    |   round(Exp)
    |   Exp < Exp
    |   Exp + Exp
    |   undefined
    |   Var = Exp
    |   Var
    |  ( Exp )
```

```haskell
data Type = Bool | Float | Int
data Stmt = Empty
          | Decl Type Var Exp Stmt   -- exp must have type given
          | While Exp Stmt Stmt      -- exp must have type bool
data Exp
    = ILit Int       -- has type int
    | FLit Float     -- has type float
    | Round Exp      -- has type int, arg must have type float
    | Less Exp Exp   -- has type int, args must have type int
    | Plus Exp Exp   -- has same type as args, args must have type both int or both float
    | Undefined      -- has any type
    | Assign Var Exp-- has same type as var, var type must match exp type
    | Variable Var   -- has type of arg

data StmtExpAlgebra s e = SEAlg
    {   empty   :: s
    ,   decl    :: Type -> Var -> e -> s -> s
    ,   while   :: e -> s -> s -> s
    ,   ilit    :: Int   -> e
    ,   flit    :: Float -> e
    ,   round   :: e -> e
    ,   plus    :: e -> e -> e
    ,   less    :: e -> e -> e
    ,   undefined :: e
    ,   assign    :: Var -> e -> e
    ,   variable  :: Var -> e
    }

foldStmt :: StmtExpAlgebra s e -> Stmt -> s
```

Users of the language are complaining that it's too Haskell-like. In particular, they don't like how their programs crash at run-time if they contain `undefined`.

For example, the following program would crash at run-time:

```
int i = 0;
float acc = 0.0;
while (i < undefined + round(undefined)) {
  bool x = undefined;
  int dummy = (i = i + 1);
  float dummy = (acc = acc + (undefined + 2.0));
}
```

To satisfy the users, you are tasked with writing an *interaction mode*, to move `undefined` errors from run-time to compile-time.

Implement an `undefined`-finding algebra `listUndefined` (in pseudo-Haskell), which lists the type of every `undefined` expression in a program:

```haskell
listUndefined :: StmtExpAlgebra [Type] [Type]
```

You may assume that input programs are well-scoped and well-typed. Your algebra should list holes in the order that they occur in the program.

For example, your algebra should produce this output for the program shown above:

```haskell
foldStmt listUndefined exampleProgram ==
  [ Int
  , Float
  , Bool
  , Float
  ]
```

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

Consider the following three languages:

L1 The language of comma-separated numbers. (see sec. 6 for examples)

L2 The language of arithmetical expressions.

```
L2 = { w | w ∈ {0,1,2,3,4,5,6,7,8,9,+,*,(,)}
         , w has matching parentheses and operators are given numerical arguments }
```

(see sec. 7 for examples)

L3 The language of rectangular comma-separated matrices of whole numbers.

```
L3 = { ((\d* ,)ᵐ \n)ⁿ | m > 0, n > 0 }
```

It is important here that each line has the same length. (see sec. 8 for examples)

5.1 (5p) Prove that L1 is Regular

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

5.3 (20p) Choose one: Either prove that L2 is not Regular, or that L3 is not Context-Free.

If you use a different version of the pumping lemma than covered in the course, please specify which version you use. In case you choose to prove that L3 is not context-free, we provide the pumping lemma for context-free languages for you:

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

6 Appendix: Extra information about the language of *comma-separated numbers*

The language of *comma-separated numbers* consists of numbers and commas.

6.1 These are well-formed rows:

✅

✅ `0`

✅ `1,234,5`

✅ `67,89,012345,6,7,8`

6.2 These are not well-formed rows:

❌ `,0,123`

❌ `45,67,`

❌ `,89,`

7 Appendix: Extra information about the language of *well-formed arithmetical expressions*

The language of *well-formed arithmetical expressions* consists of numbers, parentheses, multiplications and additions. The guideline is that a dumb calculator should be able to assign a numerical value to the expression: paretheses should match, operators should have two arguments, and the expression can't be empty.

7.1 These are well-formed arithmetical expressions:

✅ `0`

✅ `((1+2))*3`

✅ `((((4+5)*(6*7*8))))`

7.2 These are not well-formed arithmetical expressions:

❌

❌ `+0`

❌ `1(+2)`

❌ `3*+4`

❌ `5+((6*7)`

8 Appendix: Extra information about the language of *rectangular matrices*

The language of *rectangular, comma-separated matrices of numbers* consists of numbers, commas, and newlines. The rectangular property means that each line has the same amount of numbers, but each number can have any positive amount of digits. Each line should end in a comma followed by a newline character (\n).

8.1 These are well-formed matrices:

✅ `0,\n`

✅ `1,23,45,\n67,8,9,\n`

✅ `7,89,\n01,2,\n345,6,\n`

8.2 These are not well-formed matrices:

❌

❌ `0\n`

❌ `1,`

❌ `1,,2,\n`

❌ `2,3,4,5\n6,\n`

❌ `7,\n,8,\n`