General Computer Science II (320201) Spring 2007

Michael Kohlhase
Jacobs University Bremen

April 8, 2013

Contents

Assignment 1: Graphs and Trees
(Given Feb. 7., Due Feb. 14.)

Conjecture 1 *1. Let $G = \langle V, E\rangle$ be a directed graph. Then,*

$$\sum_{i=1}^{\#(V)} indeg(v_i) = \sum_{i=1}^{\#(V)} outdeg(v_i) = \#(E)$$

2. If G is undirected, we have

$$\sum_{i=1}^{\#(V)} deg(v_i) = 2 \cdot \#(E)$$

25pt

Problem 1.1 (Degrees in an Undirected Graph)

Prove or refute the conjecture above

Note: For undirected graphs, we introduce the notation deg with $\deg(v) = \mathrm{indeg}(v) = \mathrm{outdeg}(v)$ for each node.

Solution:

Proof: by induction over $m = \#(E)$

P.1.1 $m = 0$ **(base case)**: For graphs that only consist of isolated nodes, both assertions hold trivially. □

P.1.2 $m \to m + 1$ **(induction step)**:

P.1.2.1 If we remove an arbitrary edge $e \in E$ from G, we obtain $G\backslash\{e\}$

P.1.2.2 $G\backslash\{e\}$ is a directed (or undirected, resp.) graph with m edges.

P.1.2.3.1 **directed graph**: By removing one edge, we have decreased the sum of in-degrees as well as the sum of out-degrees by one. □

P.1.2.3.2 **undirected graph**: By removing one edge $e = \langle u, v\rangle$, we have decreased the degree of u as well as the degree of v by one and thus the sum of degrees by two. □

□

□

Problem 1.2 (Node Connectivity Relation is an Equivalence Relation) 15pt

Let $G = \langle V, E \rangle$ be an undirected graph and the relation C be defined as

$$C := \{\langle u, v \rangle \,|\, \text{there is a path from } u \text{ to } v\}$$

Prove or refute that C is an equivalence relation.

Solution:

Proof:

P.1.1 reflexivity: From every node, there is a zero-length path to itself. □

P.1.2 symmetry: If there is a path from a node u to a node v, just reverse it. □

P.1.3 transitivity: If there is a path from u to v and a path from v to w, we can reach w from u via v. □

□

Problem 1.3 **(Parse Tree)** 30pt

Given the data type prop for formulae

```
datatype prop = tru | fals (* true and false *)
              | por of prop * prop (* disjunction *)
              | pand of prop * prop (* conjunction *)
              | pimpl of prop * prop (* implication *)
              | piff of prop * prop (* biconditional *)
              | pnot of prop (* negation *)
              | var of int (* variables *)
```

Write an SML function that computes the parse tree for a formula. The output format should be

- a list of integers for the set of vertices,
- a list of pairs of integers for the set of edges,
- and for the labeling function a list of pairs where the first component is an integer and the second a string (the label).

Solution:

```
datatype prop = tru | fals (* true and false *)
| por of prop * prop (* disjunction *)
| pand of prop * prop (* conjunction *)
| pimpl of prop * prop (* implication *)
| piff of prop * prop (* biconditional *)
| pnot of prop (* negation *)
| var of int (* variables *)
```

The output is a triple (vertices, edges, labeling_list) where

- vertices is simply an integer (so the vertices are represented as integers from 1 to that number)
- edges is a list of pairs of integers that are the vertices between which there are edges
- labeling_list: a list of (vertex:integer, label:string) pairs that will be used to make a labeling function which takes the index of a vertex and returns its label, e.g. a string "impl" Input: A pair (root, p)
- root is a name for the root node (an integer), see the function maketree
- p a variable of datatype prop in which a boolean expression is stored

```
fun totree(mroot, pnot(be)) = let val (verout, edges, lblpairs) = totree(mroot+1, be)
                                                          in (verout, [mroot, mroot+1] :: edges, (mroot, "−") ::
                                                          end |
      totree(mroot, tru) = (mroot, nil, [(mroot, "T")])|
      totree(mroot, fals) = (mroot, nil, [(mroot, "F")])|
        totree(mroot, var v) = (mroot, nil, [(mroot, Int.toString(v))])|
        totree(mroot, two_var) = let val ax = fn (por(be1, be2)) => ("OR", be1, be2) |
                                                                            (pand(be1, be2)) =>
                                                                            (pimpl(be1, be2)) =>
                                                                            (piff(be1, be2)) => ("
                                              val (lbl, be1, be2) = ax two_var
                                              val (verout1, edges1, lblpairs1) = totree(mroot+
                                              val (verout2, edges2, lblpairs2) = totree(verout1
                                        in (verout2, [[mroot, mroot+1],[mroot, verout1+1]] @ ec
                                        end
```

Now we have an optional wrapper function that

- eliminates the need for the index of the root (default value is 1)
- converts the labeling_list into labeling function that takes a vertex and returns its label

```
local
        fun findString(num, hd::tl) = let
                val (a,b) = hd
        in
                if a = num then b
                else findString(num, tl)
        end

in
        fun maketree(t) = let
                val (lastnode, edges, lblpairs) = totree(1, t)
        in (lastnode, edges, fn num => findString(num, lblpairs))
        end
end
```

Finallly: an example of how the program can be tested:

```
totree(1, por(por(piff(pnot(var 4), fals), pimpl(tru, pand(var 2, var 3))), var 9));
val it =
(12,[[1,2],[1,12],[2,3],[2,7],[3,4],[3,6],[4,5],[7,8],[7,9],[9,10],[9,11]],
[(1,"OR"),(2,"OR"),(3,"<=>"),(4,"−"),(5,"4"),(6,"F"),(7,"=>"),(8,"T"),
(9,"AND"),(10,"2"),(11,"3"),(12,"9")])
: int * int list list * (int * string) list *)
```

Assignment 2: Combinatorial Circuits
(Given Feb. 14., Due Feb. 21.)

15pt

Problem 2.4 (Number of Paths in Balanced Binary Tree)
Let $p(n)$ be the number of different paths in a fully balanced binary tree of depth n. Find a recursive equation for $p(n)$.

Solution: Base case: $p(0) = 0$ and $p(1) = 2$. Recursive rest for $n > 1$:

$$p(n) := n * 2^n + p(n-1)$$

Problem 2.5 (Combinational Circuit for Logical Equivalence) 15pt

Logical equivalence can be expressed by the Boolean function

$$f\colon \{0,1\}^2 \to \{0,1\}; \langle i_1, i_2\rangle \mapsto (\overline{i_1} + i_2) * (i_1 + \overline{i_2})$$

Draw the corresponding combinational circuit and write down its labeled graph $G = \langle V, E, f_g\rangle$ in explicit math notation.

Problem 2.6 (Binary Comparator) 15pt

Design a combinational circuit from AND, OR, and NOT gates that takes two n-bit binary numbers and returns 1 if they are equal and zero otherwise. What is the depth of your circuit? Modify the first circuit to a circuit that returns 1 if one n-bit binary number is greater than the other.

Problem 2.7 (Is implication universal?) 30pt

Imagine a logical gate IMPL that computes the logical implication $a \Rightarrow b$. Prove or refute whether the set $S = \{\text{IMPL}\}$ is *universal*, considering the following two cases:

1. combinational circuits without constants
2. combinational circuits with constants

If the set turns out to be *not* universal in either of the cases, add *one* appropriate non-universal gate $G \in \{\text{AND}, \text{OR}, \text{NOT}\}$ to S, and prove that the set $S' = \{\text{IMPL}, G\}$ is universal.

Note: A set of boolean function is called *universal* (also called "functionally complete"), if *any* boolean function can be expressed in terms of the functions from that set. {NAND} is an example from the lecture.

Solution:

Proof:

P.1.1 combinational circuits without constants:

P.1.1.1 $S = \{\text{IMPL}\}$ is not universal, as the NOT gate cannot be constructed from IMPL gates only, because:

- $a \Rightarrow a = 1$
- $1 \Rightarrow a = a$
- $a \Rightarrow 1 = 1$
- ...and all other combinations reduce to the above.

P.1.1.2 If we choose $G = \text{NOT}$, we can construct NOR from the elements S because of $a \downarrow b = \neg(\neg a \Rightarrow b)$. As we know that {NOR} is universal, $\{\text{IMPL}, G\}$ is universal, too. □

P.1.2 combinational circuits with constants:

P.1.2.1 The NOT gate can be constructed using IMPL and a constant input of 0, because $a \Rightarrow 0 = \neg a$.

P.1.2.2 Now we can argue as in the first case. □

□

Assignment 3: Combinatorial Circuits, Arithmetics

(Given Feb. 21., Due Feb. 28.)

40pt

Problem 3.8 (A Counter with Minimal Changes)

If you imagine a counter for four-bit binary numbers that counts up from 0000 to 1111 and then from 0000 again in constant time intervals, there are steps where several outputs change at one, e.g. from 0111 to 1000. For some applications it is not desirable to have multiple outputs change their value at once; it is better to have exactly one output change.

Design the combinational circuit of an encoder with four inputs $a_3 \ldots a_0$ (which can be the four outputs of a binary counter). In each counting step, only one bit of the output $c_3 \ldots c_0$ of the encoder should change its value.

1. Suggest an appropriate code by giving a truth table that maps each of the successive states $a_3 \ldots a_0$ of the binary counter to one code value $c_3 \ldots c_0$.

2. Implement your encoder as a combinational circuit.

Solution: There is more than one solution, but the most intuitive one leads to the following code, called a "Gray code" (see Wikipedia):

```
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000
```

The conversion function from ordinary binary code to Gray code requires a right-shift operation (which can here be implemented by rewiring the inputs properly) and exclusive or: $G = B \oplus SHR(B)$

Problem 3.9 (Sign-and-Magnitude Adder) 40pt

Recall the naïve *sign and magnitude* representation for n-bit integers: If the sign bit is 0, the number is positive, else negative. The other $n-1$ bits represent the absolute value of the number.

1. Describe how to add two equally-signed n-bit numbers (simple).
2. Describe how to add two n-bit numbers numbers with different sign bits (a bit more tricky).
3. Draw a combinational circuit of a 4-bit sign and magnitude adder (one sign bit, three data bits). You may use the 1-bit full adder/subtractor (with one input that selects whether to add or to subtract) known from the lecture, an n-bit multiplexer that selects one of two n-bit numbers, as well as an n-bit comparator that computes the function $f\colon \{0,1\}^2 \to \{0,1\}$ defined as follows:

   $$f(a,b) := \begin{cases} 1 & \text{if } a \leq b \\ ,0 & \text{else} \end{cases}$$

   Be sure to explain the layout of your circuit.
4. How can an over-/underflow be detected at the outputs? In which cases can an over-/underflow occur?

Solution:

1. Add the absolute values and keep the sign
2. Choose the greater of the absolute values of the two numbers ($|a|$), subtract the absolute value of the other number ($|b|$) from it and take the sign of a.
3. (Wire the above.)
4. An over-/underflow can occur when two equally-signed numbers are added. It can be detected by checking the last carry-out.

Problem 3.10 (State After an Arithmetic Operation) 20pt

After arithmetic operations, it is important to know the following about the result:

Zero The result is zero / not zero

Negative The result is positive / negative

Overflow The operation has been executed correctly / has led to an over-/underflow

Explain in detail — but without a formal proof — how the above conditions (here, we handle over- and underflow as one case) can be derived from the values of edges in a combinational circuit after performing an addition of two two's complement numbers. Give examples, if appropriate.

Solution:

Zero Simplest solution: feed all outputs of the adder/subtractor into a NOR gate (and check whether the 0 is not the result of an operation that caused an overflow!)

Negative Check the sign bit and whether there was no overflow

Overflow Check whether the carry-in and carry-out of the sign bit are identical (see lecture)

Assignment 4: Sequential Logic Circuits, Memory, Register Machine

(Given Feb. 28., Due Mar. 7.)

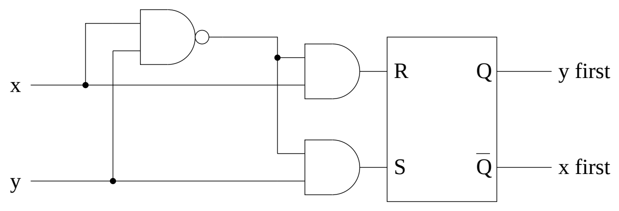
30pt

Problem 4.11 (Event Detection with RS Flipflops)
Using RS flipflops, you can detect events.

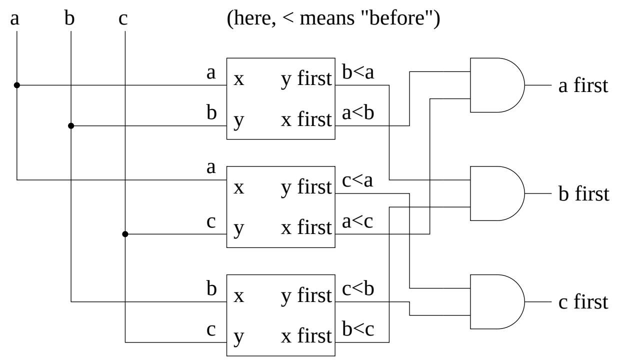
1. Design a sequential logic circuit (draw a graph) with two inputs and two outputs that detects, which out of *two* events occurred first. Use the RS flipflop and elementary gates (AND, OR, NOT, ...). Assume that, initially, all inputs are 0 and the RS flipflop(s) are holding a 0. If input I_i, where $i \in \{1, 2\}$, changes its value to 1, output O_i should change its value to 1, and all other outputs should yield 0. The outputs must not change any more when the second input changes to 1.
2. Combine several (how many?) of the circuits from step 1 to a similar event detector for three events.

Note: You need not handle the case of two inputs simultaneously changing to 1.

Solution: Circuit that checks which out of two events (x or y) occurred first:



Three of those combined to a circuit that checks which out of three events (a, b, or c) occurred first:



Problem 4.12 (3-bit Address Decoder) 10pt

Design a three-bit address decoder. Draw your circuit as a graph.

Problem 4.13 (Poking Zeros) 20pt

Given are $n \geq 1$ (n is stored in $P(0)$) integers stored in an array $P(10) \ldots P(9+n)$; some of them are zeros. Write an assembler program that overwrites all zeros in the array with a sum calculated in the following way: If $P(k) = 0, 10 \leq k \leq 9+n$, then the new value of $P(k)$ should be the sum of all values from those array fields $P(10) \ldots P(k-1)$ that were not zero *before* the execution of your program.

Problem 4.14 (A Shift Program) 15pt

Write an assembler program that shifts the values of only the first n cells to its upper neighbor, where n is the content of the accumulator; i.e. if $P(k) = a_k$ for $k = 1 \ldots n$ is the state before the execution of the program then it must be $P(k+1) = a_k$ for $k = 1 \ldots n$ afterwards and the program must terminate.

Solution:

P	instruction	comment
0	`MOVE` ACC $IN1$	$IN1 := ACC$
1	`LOADIN` 1 0	$ACC := P(IN1)$
2	`STOREIN` 1 1	$P(IN1+1) := ACC$
3	`MOVE` $IN1$ ACC	$ACC := IN1$
4	`SUBI` 1	$ACC := ACC - 1$
5	`JUMP`$_=$ > 1	if $ACC > 0$ goto step 0
6	`STOP` 0	Stop

Problem 4.15 (A Recursive Function) 25pt

Write an assembler program that computes the following recursive function:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(i) &= 4 \cdot f(i-2) + (-1)^i \cdot f(i-1), i \geq 2 \end{aligned}$$

Assume that the input i is given in cell 0. Store the result in cell 3. If you need additional assembler instructions for which we know circuits from the lecture (e.g. left/right shifting n-bit numbers, or inverting them, i.e. computing the bit-wise NOT), you may give them descriptive names like *shiftr* or *inv*. But be sure to use only circuits that implement functions $f\colon \mathbb{B}^n \to \mathbb{B}^n$!

Assignment 5: Register Machine, Virtual Machine
(Given March 7., Due March 14.)

60pt

Problem 5.16 (Simulating a Register Machine)
Write an SML function regma (register machine) that simulates the simple register machine we discussed in class. To represent the program and data store, you should use SML vectors as described in `http://www.standardml.org/Basis/vector.html`. In a nutshell, Vector.sub(arr,i) returns the i^{th} element of the vector arr and Vector.update(arr,i,x) returns the vector arr, except that the i^{th} element is replaced by x. Finally (useful for testing) Vector.fromList makes a vector from a list.

So the the data store should be of type int vector and the program store is of type (instruction $*$ int) vector, where instruction is defined by the following type

```
datatype instruction =
          load | store | add | sub | loadi | addi | subi |
          loadin1 | loadin2 | storein1 | storein2 |
          moveaccin1 | moveaccin2 | movein1acc | movein2acc | movein1in2 | movein2in1 |
          jump | jumpeq | jumpne | jumpless | jumpleq | jumpgeq | jumpmore |
          nop | stop
```

regma should take as input a data store data and a program store prog, and regma(prog,data) should return the value of the accumulator register, when the program encounters a stop instruction.

Solution: We first write an auxiliary function that takes care of the current instruction by a large case statement. Let us start out with the load/store instructions:

```
\scriptsize
fun arg(n,p) = let val (_,a) = Vector.sub(n,p) in a end
fun doinst ((load,i),prog,data,acc,pc,in1,in2) =
       doinst(Vector.sub(pc+1,prog),prog,data,arg(pc,prog),pc+1,in1,in2)
  | doinst((store,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,Vector.update(data,acc,i),pc+1,in1,in2)
  | doinst ((loadi,i),prog,data,acc,pc,in1,in2) =
       doinst(Vector.sub(pc+1,prog),prog,data,i,pc+1,in1,in2)
  | doinst((loadin1,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,Vector.sub(data,in1+i),pc+1,in1,in2)
  | doinst((loadin2,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,Vector.sub(data,in2+i),pc+1,in1,in2)
  | doinst((storein1,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,Vector.update(data,acc,in1+i),pc+1,in1,in2)
  | doinst((storein2,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,Vector.update(data,acc,in2+i),pc+1,in1,in2)
```

Then come the cases for the computation instructions, where we just make use of the SML computation facilities.

```
\scriptsize
  | doinst((add,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc + Vector.sub(data,i),pc+1,in1,in2)
  | doinst((Vector.sub,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc − Vector.sub(data,i),pc+1,in1,in2)
  | doinst((add,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc + i,pc+1,in1,in2)
  | doinst((Vector.sub,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc − i,pc+1,in1,in2)
```

The register move instructions are rather boring:

```
    \scriptsize
  | doinst(moveaccin1,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,acc,in2)
  | doinst(moveaccin2,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,in1,acc)
  | doinst(movein1acc,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,in1,pc+1,in1,in2)
  | doinst(movein2acc,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,in2,pc+1,in1,in2)
  | doinst(movein1in2,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,in1,in1)
  | doinst(movein2in1,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,in2,in2)
```

The jump instructions can be mapped to conditional expressions in SML using the SML comparisons

```
    \scriptsize
  | doinst((jump,i),prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+i,prog),prog,data,acc,pc+i,in1,in2)
  | doinst((jumpeq,i),prog,data,acc,pc,in1,in2) =
        if (acc = 0)
        then doinst(Vector.sub(pc+i,prog),prog,data,acc,pc+i,in1,in2)
        else doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,in1,in2)
  ...
  | doinst((jumpmore,i),prog,data,acc,pc,in1,in2) =
        if (acc > 0)
        then doinst(Vector.sub(pc+i,prog),prog,data,acc,pc+i,in1,in2)
        else doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,in1,in2)
```

For the skip instruction we do nothing, except increment the program counter and supply the next instruction:

```
    \scriptsize
      | doinst(nop,prog,data,acc,pc,in1,in2) =
      doinst(Vector.sub(pc+1,prog),prog,data,acc,pc+1,in1,in2)
```

Finally, the stop instruction just returns the value of the accumulator.

```
    \scriptsize
      | doinst(stop,prog,data,acc,pc,in1,in2) = acc
```

With this giant case distinction, the function regma is very simple, we just have to use doinst with suitable initial values.

\scriptsize

```
fun regma (prog,data) = doinst(Vector.sub(0,prog),prog,data,0,1,0,0)
```

Problem 5.17 (Testing the Simulation of the Register Machine) 10pt

Test your SML simulation regma of the register machine by implementing the assembler programs from last week's assignment on it, i.e. by rewriting them as SML data structures as stated in the regma problem assignment.

Problem 5.18 (Even Odd Test) 15pt

Assume the data stack initialized with `con` n for some natural number n. Write a $\mathcal{L}$(`VM`) program that returns on top of the stack 0 if n is even and 1 otherwise. Furthermore, draw the evolution of the stack during the execution of your program for $n = 4$ and $n = 5$.

Problem 5.19 (Fibonacci Numbers) 15pt

Assume the data stack initialized with `con` n for some natural number n. Write a $\mathcal{L}$(`VM`) program that computes the n^{th} Fibonacci number and returns it on the top of the stack.

Solution:

con n The requested fibonacci number
con 0 con 1 The 0th **and** the 1st fibonacci number
con 0 peek 0 leq cjp 5 If $\RMdatastore{0}$ <= 0
peek 1 halt return the current fibonacci number
peek 2 **else** save the next fibonacci number ...
peek 1 peek 2 add poke 2 ... compute the number after next **and** save at $\RMdatastore{2}$
poke 1 ... make the next number current ...
con 1 peek 0 sub poke 0 ... decrease n by 1 ...
jp −28 ... **and** jump back the the beginning

Assignment 6: Virtual Machine
(Given March 14., Due March 21.)

20pt

Problem 6.20 (Environment of the Virtual Machine)
Implement the environment for the virtual machine in SML based on the following signature introduced in class:

```
type a env
exception Unbound of id
val empty : a env
val insert : id * a * a env -> a env
val lookup : id * a env -> a (* Unbound *)
```

Solution:

```
type id = string (* identifier *)
type index = int
type env = id->index

exception Unbound of id

fun empty _ = ~1; (* a kind of empty function *)

fun insert(k:id, v:index, env:env) =
    fn key => if key = k then v
                        else env(key)

fun lookup(k:id, env:env) =
        if env(k) = ~1 then raise Unbound k
        else env(k);
```

Problem 6.21 (New Statements for SW) 30pt

Extend the Simple While Language SW by a repeat−until and a for loop, two variants of the while loop. Consider both the abstract syntax data type sta and the compileS function.

The concrete syntax of a repeat−until loop looks as follows:

```
repeat
  (* statements *)
until (* conditional−expression *);
```

... where the statements in the body are repeated until the conditional expression evaluates to true (i.e. a number $\neq 0$).

The concrete syntax of a for loop looks as follows:

```
for (* counting−variable *) := (* start−expr *) to (* end−expr *) do
  (* statements *)
end;

(* Example:
var i;
var n := 10;
var sum := 0;
for i := 1 to n do
  sum := sum + i;
end; *)
```

Assume that a counting variable, e.g. named i, has been declared before. In the first run of the loop, i is initialized to the value of *start-expr*. After each run of the loop, i is increased by one. In the last run of the loop, i has the value of what *end-expr* evaluated to *before* the loop started; then the execution of the loop stops.

Problem 6.22 (Moving a parameter from `proc` to `call`) 25pt

In the realization of `call`, we write the number of arguments in the frame by taking the parameter a of the corresponding `proc` instruction (referred to as 'stealing' in the comments). An alternative is to have a `call` that takes two arguments: the address of the procedure and the number of its arguments. Clearly, `proc` will have only one argument (the length of the procedure) in that implementation. Realize such instructions in `ASM` by modifying the current implementation.

Assignment 7: Turing Machines
(Given April 12, Due April 19)

25pt

Problem 7.23 (Turing Machine Evaluation)
What does the following Turing machine do?

- Its input is a word $w \in \{0,1\}^*$, surrounded by hash marks as delimiters; i.e. the overall alphabet is $\{0, 1, \#\}$.
- The states are $\{s_0, s_1\}$, with an initial state of s_0. Initially, the head is on the first character of w.
- Admissible moves are L (*left*), R (*right*), and N (*none*).
- The transition function is defined as follows:

Old	Read	Write	Move	New
s_0	0	1	N	s_0
s_0	1	1	R	s_0
s_0	#	1	N	s_1
s_1	0	0	L	s_1
s_1	1	1	L	s_1
s_1	#		(halt)	

Simulate the machine on the input 1010. Specify the behaviour of the machine, interpreted as

1. a function on strings, for example: "$\{0,1\}^* \ni w \mapsto 0^{|w|}1^m$, where $m < |w|$ is the number of 1's in w"
2. a function on binary-encoded integers, for example: "$\mathbb{N} \ni w \mapsto \log_2 w + 3w$"

where the final word on the tape is considered as the return value of the function. Try to be as formal as possible.

Solution: The input 1010 yields the output 11111.

1. $\{0,1\}^* \ni w \mapsto 1^{|w|+1}$
2. $\mathbb{N} \ni w \mapsto 2^{\lceil \log_2 w \rceil + 1} - 1$

Problem 7.24 (Palindrome Detector Turing Machine) 25pt

Create a Turing machine that detects whether the word $w \in \{0,1\}^*$ on the tape is a palindrome, i.e. $w = w^R$, where w^R is w reversed. If so, leave a 1 at the final position of the head, which can be anywhere on the tape; otherwise leave a 0. Assume that, initially, the head is over the first character of w, and that w is surrounded by hash marks as delimiters, i.e. you have the alphabet $\{0,1,\#\}$.

Note: Admissible moves are L (*left*), R (*right*), and N (*none*) with the obvious meaning.

Solution: Idea: Define states that encode one memorized character, as well as the information whether the beginning or the end of the word is currently being examined. In each step, we compare the first character of the word to the last one, erase them and proceed.

1. Read a character at one end of the word.
   - If it is a digit, memorize whether it was 0 or 1 by using an appropriate state.
   - If it is a hash mark, write a 1 and halt.
2. Overwrite the character with a hash mark.
3. Walk to the other end of the word, i.e. move until a hash mark is read and then move one cell back to the beginning/end of the word (using the "motion" information encoded in the current state).
4. - If the character under the head does not match the memorized one, write a 0 and halt.
   - If it does match, overwrite it with a hash mark, move one cell further to the new beginning/end of the word and continue from the beginning.

Problem 7.25 (Number of Steps of a Turing Machine) 30pt

Let $s_{\max}(n)$ be the maximum number of steps that an n-state Turing machine with the alphabet $\{0, 1\}$ can take on an empty tape, halting in the end. Is the function $s_{\max}$ computable? Give a proof or a refutation.

Note: From the lecture, we know that it is impossible to implement a function will_halt(program, input). Assume the following corollary, known as the "halting problem on the empty tape", as given: It is even impossible to write a Turing machine (or an equivalent function will_halt_empty(program), resp.) that tells whether an arbitrary Turing machine halts on an *empty* tape.

Solution:

Proof: by contradiction

P.1 $s_{\max}(n)$ is the maximum number of steps a halting n-state Turing machine can take on an empty tape.

P.2 Any n-state machine that runs for more than $s_{\max}(n)$ steps must be non-halting.

P.3 Using $s_{\max}$, one could now implement will_halt_empty(TM) as follows:

- Let n be the number of states of TM.
- Compute $m := s_{\max}(n)$.
- Simulate at most m steps of TM.
- If TM has not halted so far, return "yes"; otherwise, return "no".

P.4 This contradicts the non-computability of will_halt_empty. □

Assignment 8: Problems and Searching

(Given April 18, Due April 25)

20pt

Problem 8.26 (The Dog/Chicken/Grain Problem)
A farmer wants to cross a river with a dog, a chicken, and a sack of grain. He has a boat which can hold himself and either of these three items. He must avoid that either dog and chicken or chicken and grain are together alone on one river bank, since otherwise something gets eaten.

1. Represent the farmer's problem of crossing the river without losing his goods as a search problem.
2. Draw a sufficiently large portion of the search tree induced by this problem to exhibit a solution.
3. Discuss three search strategies and their advantages and disadvantages in this scenario.

Solution: We present the states as a pair $\langle S,T\rangle$ of sets, where S is the set of items of the on the start bank and T that of items on the target bank. The initial state is $\langle\{f,d,c,g\},\emptyset\rangle$ and the goal state is $\langle\emptyset,\{f,d,c,g\}\rangle$. The actions are represented as $cross(A)$, where A is the item taken over $back(A)$ the action of taking A back.

$$\begin{array}{l}
\langle\{f,d,c,g\},\emptyset\rangle\longrightarrow^{c(c)}\langle\{d,g\},\{f,c\}\rangle\longrightarrow^{b}\langle\{f,d,g\},\{c\}\rangle\longrightarrow^{c(d)}\\
\langle\{g\},\{f,d,c\}\rangle\longrightarrow^{b(c)}\langle\{f,c,g\},\{d\}\rangle\longrightarrow^{c(g)}\langle\{c\},\{f,d,g\}\rangle\longrightarrow^{b}\\
\langle\{f,c\},\{d,g\}\rangle\longrightarrow^{c(c)}\langle\emptyset,\{f,d,c,g\}\rangle
\end{array}$$

Problem 8.27: Describe a state space in which iterative deepening search performs much worse than depth-first search (for example $O(n^2)$ vs. $O(n)$). 10pt

Solution: Depth-First search strategy performs great if the solutions are dense. Consider an abstract situation where we have many possible solutions and they are roughly at the same depth. Furthermore let the solution with minimal depth be at a high depth level.

To make the situation more concrete consider a search problem with the following parameters:

- $b = 100$
- $d = 33$
- $m = 36$

Assume furthermore that the solutions are very dense (most leaves represent a possible solution). Here depth-first will find a solution extremely fast while iterative-deepening will take much more time to reach the optimal solution at depth 33.

Problem 8.28 (Implementing Search) 50pt

Implement the depth-first and breadth-first search algorithms in SML. The functions depthFirst and breadthFirst take three arguments that make up the problem description:

1. the initial state
2. a function next that given a state x in the state tree returns at set of pairs (action,state): the next states (i.e. the child nodes in the search tree) together with the actions that reach them.
3. a predicate (i.e. a function that returns a Boolean value) goal that returns true if a state is a goal state and false else.

the result of the functions should be the goal state together with a list of actions that reaches the goal state from the initial state.

Solution: We will follow the hint and write a simple function first and later extend it to the full case.

```
exception search_exhausted
fun depthFirst next goal x =
    let fun dfs [] = raise search_exhausted
          | dfs (state::rest) = if goal(state) then state
                                else dfs (next state @ rest)
     in dfs [x] end;
\smlout{val depthFirst = fn : ('a −> 'a list) −> ('a −> bool) −> 'a −> 'a}

fun breadthFirst next goal x =
    let fun bfs [] = raise search_exhausted
          | bfs (state::rest) = if goal(state) then state
          else bfs (rest @ next state)
     in bfs [x] end;
\smlout{val breadthFirst = fn : ('a −> 'a list) −> ('a −> bool) −> 'a −> 'a}
```

Note that the programs only differ in the order of the arguments in the recursive call of the local function. Now, we extend the functions to deal with actions. Here we add the plans how to get to the fringe node to the states in the argument of the local function. Thus we need to add the current plan to the actions in the recursive call.

```
fun depthFirst next goal x =
    let fun dfs [] = raise search_exhausted
          | dfs ((plan,state)::rest) =
             if goal(state) then plan
             else dfs ((map (fn ((act,st)) => (act::plan,st)) (next state)) @ rest)
     in rev(dfs [(nil,x)]) end;
\smlout{val depthFirst = fn : ('a −> ('b * 'a) list) −> ('a −> bool) −> 'a −> 'b list}
fun breadthFirst next goal x =
    let fun bfs [] = raise search_exhausted
          | bfs ((plan,state)::rest) =
             if goal(state) then plan
             else bfs (rest @ (map (fn ((act,st)) => (act::plan,st)) (next state)))
```

```
      in rev(bfs [(nil,x)]) end;
\smlout{val breadthFirst = fn : ('a -> ('b * 'a) list) -> ('a -> bool) -> 'a -> 'b list}
```

Problem 8.29 (Repeated States) 20pt

Extend the functions in ?? with a check for repeated states. Compare the run-times with the naive versions.

Solution: To check for repeated states, we simply add another argument visited to the local function and check whether states have been visited before.

```
fun member q [] = false | member q(x::xs) = (q=x) orelse (member q xs);
fun depthFirst next goal x =
    let fun dfs([],_) = raise search_exhausted
          | dfs((plan,state)::rest,visited) =
              if member(state,visited) then dfs(rest,visited)
              else if goal(state) then plan
              else dfs((map (fn ((act,st)) => (act::plan,st))
                (next state)) @ rest,state::visited)
     in rev(dfs([(nil,x)],nil)) end;
\smlout{val depthFirst = fn : ('a -> ('b * 'a) list) ->
  ('a -> bool) -> 'a -> 'b list}
```

Assignment 9: Problems and Searching
(Given April 28, Due May 9)

20pt

Problem 9.30 (Actions with Negative Costs)
Suppose that actions can have arbitrary large negative costs.

1. Explain why this possibility would force any optimal algorithm to explore the entire state space.

2. Does it help if we insist that step costs must be greater than or equal than to some negative constant c? Justify your answer.

Problem 9.31 (A Trip Through Romania) 20pt

Represent the Romanian map we talked about in class in a concrete next function. Search with the procedures from Problem 8.28 a trip from Arad to Bucharest. Compare the solution paths and run times.

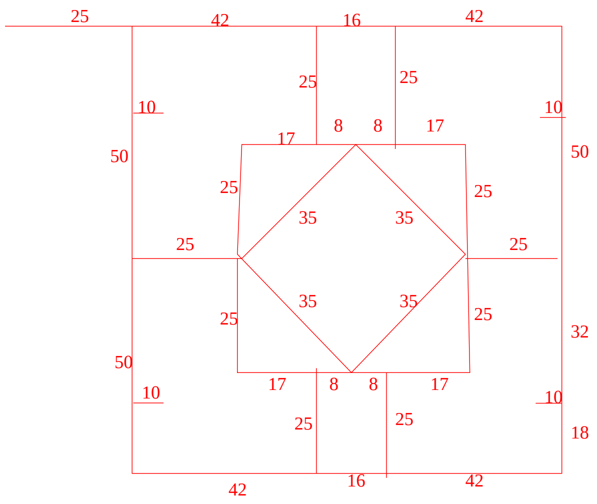
Solution:

```
datatype State = Arad | Zerind | Oradea | Timisoara | Sibiu | Lugoj | Mehadia |
                 RimnicuVilcea | Fagaras | Pitesti | Craiova | Dobreta | Giurgiu |
                 Bucharest | Urziceni | Hirsova | Eforie | Vaslui | Iasi | Neamt;

datatype Actions = goAra | goZer | goOra | goTim | goSib | goLug | goMeh | goRim | goFag | goPit |
                   goCra | goDob | goGiu | goBuc | goUrz | goHir | goEfo | goVas | goIas | goNea;

fun next Arad = [(goZer, Zerind), (goSib, Sibiu), (goTim, Timisoara)] |
    next Timisoara = [(goAra, Arad), (goLug, Lugoj)] |
    next Zerind = [(goAra, Arad), (goOra, Oradea)] |
    next Oradea = [(goZer, Zerind), (goSib, Sibiu)] |
    next Sibiu = [(goAra, Arad), (goOra, Oradea), (goRim, RimnicuVilcea), (goFag, Fagaras)] |
    next Lugoj = [(goTim, Timisoara), (goMeh, Mehadia)] |
    next Mehadia = [(goLug, Lugoj), (goDob, Dobreta)] |
    next Dobreta = [(goMeh, Mehadia), (goCra, Craiova)] |
    next Craiova = [(goDob, Dobreta), (goPit, Pitesti), (goRim, RimnicuVilcea)] |
    next RimnicuVilcea = [(goCra, Craiova), (goPit, Pitesti), (goSib, Sibiu)] |
    next Pitesti = [(goRim, RimnicuVilcea), (goCra, Craiova), (goBuc, Bucharest)] |
    next Fagaras = [(goSib, Sibiu), (goBuc, Bucharest)] |
    next Bucharest = [(goPit, Pitesti), (goFag, Fagaras), (goGiu, Giurgiu), (goUrz, Urziceni)] |
    next Giurgiu = [(goBuc, Bucharest)] |
    next Urziceni = [(goBuc, Bucharest), (goHir, Hirsova), (goVas, Vaslui)] |
    next Hirsova = [(goEfo, Eforie), (goUrz, Urziceni)] |
    next Eforie = [(goHir, Hirsova)] |
    next Vaslui = [(goUrz, Urziceni), (goIas, Iasi)] |
    next Iasi = [(goVas, Vaslui), (goNea, Neamt)] |
    next Neamt = [(goIas, Iasi)];

fun goal Sibiu = true |
    goal _ = false;

val RESdepth = depthFirst(Giurgiu, next, goal);
val RESbreadth = breadthFirst(Giurgiu, next, goal);

(* The result is:
val RESdepth = (Bucharest,[goZer,goOra,goSib,goRim,goCra,goPit,goBuc])
: State * Actions list
val RESbreadth = (Bucharest,[goSib,goFag,goBuc]) : State * Actions list

We know that BFS always finds the optimal solution, and in our case
this is actually so. We go to Bucharest in 3 actions, while DFS has found a path with 7 actions.
Depending of where a solution it, BFS may prove to be faster than DFS. However, DFS will use less
memory in literally all cases. Our test case is too simple to notice any difference *)
```

Problem 9.32 (A^* **search on Jacobs campus)** 40pt

Implement the A^* search algorithm in SML and test it on the problem of walking from the main gate to the entrance of Research 3 with linear distance as heuristic. The length of line segments are annotated in the map below.

No function signature is provided, instead at the end of your program call your function so that it prints the actions needed to reach the entrance and the associated cost.



Solution:

```
val it = (["E","E","S","E","SE","E","S","W"],202) (* The states here are directions e.g. SE means Southeast. *)

fun coor 1 = (0,0) | coor 2 = (25,0) | coor 3 = (67,0) | coor 4 = (83,0) | coor 5 = (125,0) |
    coor 6 = (25,18) | coor 7 = (35,18) | coor 8 = (115,18) | coor 9 = (125,18) | coor 10 = (50,25) |
        coor 11 = (67,25) | coor 12 = (75,25) | coor 13 = (83,25) | coor 14 = (100,25) | coor 15 = (25,50) |
        coor 16 = (50,50) | coor 17 = (100,50) | coor 18 = (125,50) | coor 19 = (50,75) | coor 20 = (67,75) |
        coor 21 = (75,75) | coor 22 = (83,75) | coor 23 = (100,75) | coor 24 = (25,82) | coor 25 = (35,82) |
        coor 26 = (115,82) | coor 27 = (125,82) | coor 28 = (25,100) | coor 29 = (67,100) | coor 30 = (83,100) |
        coor 31 = (125,100);

val edges = [(1,2), (2,3), (3,4), (4,5), (6,7), (8,9), (10,11), (11,12), (12,13), (13,14), (15,16), (17,18),
                        (19,20), (20,21), (21,22), (22,23), (24,25), (26,27), (28,29), (29,30), (30,31),
                        (2,6), (6,15), (15,24), (24,28), (10,16), (16,19), (3,11), (20, 29), (4,13), (22,30),
                        (14,17), (17,23),
                        (5,9), (9,18), (18,27), (27,31),
                        (12,16), (16,21), (21,17), (17,12) ];

fun heuristic(n,m) = let
 val (x1,y1) = coor(n);
 val (x2,y2) = coor(m);
```

```
in Real.round(Math.sqrt(Real.fromInt((x1−x2)∗(x1−x2)+(y1−y2)∗(y1−y2))))
end;

fun next(n) = let
        fun successors(_,nil) = nil |
           successors(n,(a,b)::tl) = if n=a then b::successors(n,tl)
                                                        else if n=b then a::successors(n,tl)
                                                              else successors(n,tl);
        fun hlist(_, nil) = nil |
              hlist(n, hd::tl) = heuristic(n, hd) :: hlist(n, tl);
        fun tie(nil,nil) = nil |
              tie(h1::t1, h2::t2) = (h1,h2) :: tie(t1,t2);
        val succ = successors(n,edges)
        val cost = hlist(n, succ)
in
        tie(succ,cost)
end;

exception NoSolution;

(∗ASearch takes and initial node, next function and goal node and returns
the optimal path between initial and goal node ∗)

fun AStarSearch(initial, next, goal) = let
        fun putCheapestInFront(hd::tl, nil) = putCheapestInFront(tl,[hd]) |
              putCheapestInFront(nil, x) = x |
              putCheapestInFront((a,b,c,d)::tl1, (xa,xb,xc,xd)::tl2 ) =
                    if c < xc then putCheapestInFront(tl1, (a,b,c,d)::((xa,xb,xc,xd)::tl2))
                    else putCheapestInFront(tl1, ((xa,xb,xc,xd)::tl2)@[(a,b,c,d)]);
        fun addActionsCosts(_,_,nil) = nil |
              addActionsCosts(pcost, pactions, (node, cost)::tl) =
                    ( node, pcost + cost, pcost + cost + heuristic(node, goal), pactions@[node] ) ::
                    addActionsCosts(pcost, pactions, tl);
        fun asearch(nil) = raise NoSolution |
              asearch((node, pathcost, totalcost, actions)::rfringe) =
                    if node = goal then actions
                    else let
                          val expansion = next(node); (∗ next(20) = [(19,17), (21,8), (29,25)] ∗)
                          val newFringeEl = addActionsCosts(pathcost, actions, expansion);
                    in
                          asearch(putCheapestInFront(newFringeEl@rfringe, nil))
                    end
in
        asearch([(initial, 0, heuristic(initial, goal), [])])
end

(∗ The nodes are labeled starting from the upper−left corner of the map to right/down direction ∗)
val result = AStarSearch(1, next, 26);
```

Problem 9.33 (A variant of A^*) 20pt

Imagine an algorithm B^* that uses the evaluation function $f(n) = g(n) \cdot h(n)$, where $g(n)$ is the path cost to the current node n, and $h(n)$ is a heuristic function. Is this algorithm better or worse than A^*? Explain your findings. What does $h(n)$ represent?

Solution: Comment by Andrei Aiordachioaie, to be formatted...

It's not that complicated and it leaves room for creativity to students. It would be nice to see how exactly they think :) As I see it, it's worse than A^* because heuristics h(n) needs extreme values for different nodes. When n is close to the root, h(n) needs to be very big to estimate a realistic distance to the goal, while if n is near the goal, the heuristic has to be very small. We will therefore need to work with floating-point numbers, which are a bit slower :)

Assignment 10: Prolog Programming
(Given May 12, Due May 21)

30pt

Problem 10.34 (Merging Dictionaries)

Consider a dictionary data structure represented as a list of key = value pairs. Implement a predicate dict_merge(dict1,dict2,merger) that merges one dictionary with another one, if a merger is possible without a contradiction, i. e. if there is no pair of two different values for one key. If, for example, the first dictionary is [topic=gencs, lecturer=kohlhase], it can be merged with a second dictionary [university=jacobs], yielding the dictionary [topic=gencs, lecturer=kohlhase, university=jacobs]. Merging with [lecturer=kohlhase, semester=2] should also be possible, as the value for the "lecturer" key is the same, but merging with [lecturer=kramer, university=jacobs] should fail, as there are two contradicting values for the "lecturer" key.

Note: You will need the built-in predicate select(Elem, List, Rest), which selects an element Elem from a List (as member(Elem, List) does) leaving a Rest. Use \+ to negate an expression, e. g. when checking whether an entry is not contained in a dictionary.

You can easily access the two components of a pair by pattern matching. The following predicate would, for example, match the second argument with the key of the given pair: key(K=V, K).

Solution:

```
% An empty dictionary can be merged with any other dictionary.
dict_merge([], Dict, Dict).

% As our dictionaries are lists and therefore unordered,
% we have to retrieve the Key=Value pair from the second dictionary
% using the built−in predicate select.

dict_merge([Key=Value|Rest1], Dict, [Key=Value|Rest3]) :−
    % if Key=Value exists in both dictionaries, ...
    select(Key=Value, Dict, Rest2),
    % ... merge the remainders of both.
    dict_merge(Rest1, Rest2, Rest3).

% If the key of the Key=Value pair does not exist in the second
% dictionary, we can add it to the result dictionary.

dict_merge([Key=Value|Rest1], Dict, [Key=Value|Rest2]) :−
    % if Key=something does not exist in the second dictionary, ...
    \+ member(Key=_, Dict),
    % retain this entry and merge the rest of the first
    % dictionary into the second one.
    dict_merge(Rest1, Dict, Rest2).
```

Test cases:

```
dict_merge([lect=kohl,top=gencs],[uni=jac],R).
% R = [lect=kohl, top=gencs, uni=jac]
% Yes
dict_merge([lect=kohl,top=gencs],[lect=kohl,top=gencs],R).
% R = [lect=kohl, top=gencs]
% Yes
dict_merge([lect=kohl,top=gencs],[uni=jac,lect=kram],R).
% No
```

(Source of this problem: Wilhelm Weisweber, `ProLog` – logisches Programmieren in der Praxis. International Thomson Publishing 1997. ISBN 3-8266-0174-2.)

Problem 10.35 (Derivation of Arithmetic Expressions) 40pt

Write a ProLog program that "computes" the derivative of an arithmetic expression; i. e. write facts and rules such that derivative(E1,x,E2) is satisfied if and only if the arithmetic expression E2 is the derivative of E1 with respect to x (where x is an atomic expression of course). The constructors of arithmetic expressions should be any integer or lower case letter as atomic expressions together with mul/2 and add/2 for building compound arithmetic expressions.

For instance, derivative(mul(x,add(x,2)),x,add(mul(1,add(x,2)),mul(x,add(1,0)))) should evaluate to true.

Note: You do not need to implement term rewriting, i. e. you need not be recognize *all* possible ways of writing expressions equivalent to E2.

Problem 10.36 (1+1=2) 20pt

Given the facts and rules:

nat(zero).
nat(s(X)):−nat(X).
add(X,zero,X).
add(X,s(Y),s(Z)):−add(X,Y,Z).

prove with the inference rules MP, $\wedge I$ and Subst, which `ProLog` uses, that informally speaking $1 + 1 = 2$.

Solution:

1. nat(zero) . . . from knowledge base
2. nat(zero)⇒nat(s(zero)) . . . Subst[zero/X] in nat(X)⇒nat(s(X))
3. nat(s(zero)) . . . MP on 1. and 2.
4. add(s(zero),zero,s(zero)) . . . Subst [s(zero)/X] in add(X,zero,X)
5. add(a(zero),zero,s(zero))⇒add(s(zero),s(zero),s(s(zero))) . . . Subst [s(zero)/X],[zero/Y],[s(zero)/Z] in add(X,Y,Z)⇒add(X,s(Y),s(Z))
6. add(s(zero),s(zero),s(s(zero))) . . . MP on 4. and 5.

Therefore, $1 + 1 = 2$ indeed :−)

Problem 10.37 (Relevance of Rule and Literal Order for Termination) 10pt

Consider the recursive definition of unary natural numbers from the lecture:

```
unat(o).
unat(s(X)) :− unat(X).
```

Consider a variant of this program with the two clauses swapped:

```
unat(s(X)) :− unat(X).
unat(o).
```

With both programs, first try to check whether a given argument (e. g. `pi` or `s(s(o))`) is a natural number, and then try to step-wise generate all natural numbers by entering the query `unat(X).` and making `ProLog` back-track and find the next result by pressing ;. Which difference do you notice, and why does it occur?