General Computer Science II (320201) Spring 2009

Michael Kohlhase
Jacobs University Bremen

April 8, 2013

Contents

Assignment 1: Recap and graph basics
(Given Feb.4., Due Feb. 11.)

10pt

Problem 1.1: Prove in the resolution calculus using derived rules:

$$\models A \wedge (B \vee C) \Rightarrow A \wedge B \vee A \wedge C$$

Solution: Clause Normal Form transformation

$$\frac{\frac{\frac{A \wedge (B \vee C) \Rightarrow A \wedge B \vee A \wedge C^{\mathsf{F}}}{A \wedge (B \vee C)^{\mathsf{T}}; A \wedge B \vee A \wedge C^{\mathsf{F}}}}{A^{\mathsf{T}}; B^{\mathsf{T}} \vee C^{\mathsf{T}}; A \wedge B^{\mathsf{F}}; A \wedge C^{\mathsf{F}}}}{A^{\mathsf{T}}; B^{\mathsf{T}} \vee C^{\mathsf{T}}; A^{\mathsf{F}} \vee B^{\mathsf{F}}; A^{\mathsf{F}} \vee C^{\mathsf{F}}}$$

Resolution Proof

1	A^{T}	initial
2	$B^{\mathsf{T}} \vee C^{\mathsf{T}}$	initial
3	$A^{\mathsf{F}} \vee B^{\mathsf{F}}$	initial
4	$A^{\mathsf{F}} \vee C^{\mathsf{F}}$	initial
5	B^{F}	with 1 and 3
6	C^{F}	with 1 and 4
7	C^{T}	with 2 and 5
8	$\square$	with 6 and 7

Problem 1.2 (Graph basics) 20pt

For each of the five directed graphs below do the following:

- State whether the graph is also a tree and explain why.
- Determine the depth of the graph.
- Write out in math notation a path from A to E if one exists and determine the path's length.

1. $G_1 := \langle \{A, B, C, D, E\}, \{\langle A, B\rangle, \langle A, C\rangle, \langle A, D\rangle, \langle D, E\rangle\}\rangle$
2. $G_2 := \langle \{A, B, C, D, E\}, \{\langle A, B\rangle, \langle B, C\rangle, \langle C, A\rangle, \langle C, D\rangle, \langle C, E\rangle\}\rangle$
3. $G_3 := \langle \{A, B, C, D, E\}, \{\langle A, B\rangle, \langle B, C\rangle, \langle B, D\rangle, \langle C, E\rangle\}\rangle$
4. $G_4 := \langle \{A, B, C, D, E\}, \{\langle A, B\rangle, \langle A, C\rangle, \langle B, D\rangle, \langle D, C\rangle, \langle C, B\rangle, \langle A, D\rangle\}\rangle$
5. $G_5 := \langle \{A, B, C, D, E\}, \{\langle D, A\rangle, \langle D, B\rangle, \langle D, E\rangle, \langle D, C\rangle\}\rangle$

Solution:

1.
   - Yes, because it has no cycles and the graph is connected.
   - 2
   - $\langle A, D, E\rangle$ - length 2
2.
   - No, because it has a cycle - $\langle A, B, C, A\rangle$.
   - infinite
   - $\langle A, B, C, E\rangle$ - length 3
3.
   - Yes, because it has no cycles and the graph is connected.
   - 3
   - $\langle A, B, C, E\rangle$ - length 3
4.
   - No, because it has a cycle - $\langle B, D, C, B\rangle$.
   - infinite
   - E is not reachable from A, since $indeg(E) = 0$ i.e. E is a source.
5.
   - Yes, because it has no cycles and the graph is connected.
   - 1
   - E is not reachable from A, since $outdeg(A) = 0$ i.e. A is a sink.

Problem 1.3 (In-degrees in acyclic digraphs) 25pt

Prove by induction or refute that any acyclic digraph with non-empty set of nodes has at least one node with in-degree 0.

Solution:

Proof: Proof by induction on the size of (number of nodes in) the graph:

P.1.1 n=1: trivial

P.1.2 Step case: $n \Longrightarrow n+1$:

P.1.2.1 Note that for any acyclic digraph, every subgraph is also acyclic (otherwise a cycle in the subgraph would make the entire graph cyclic). This is what makes induction possible.

P.1.2.2 So assume an acyclic digraph with n nodes and at least a node with in-degree 0 such that the graph is still acyclic. Call this node v_0. Now add an extra vertex, v_n, to the graph and any number of edges that connects it with the rest of the vertices.

P.1.2.3.1 This new node has no incoming edge.: This is the new 0 in-degree node. □

P.1.2.3.2 v_0 **still has no incoming edge.**: v_0 remains the 0 in-degree node. □

P.1.2.3.3 v_0 **has an incoming edge from** v_n **and indeg**$(v_n) > 0$: Then there exists an edge from v_i to v_n. If v_i has in-degree 0, then this is a 0 in-degree node. Otherwise, since the graph is acyclic, v_i has no incoming edge from v_0 or v_n, so there is an edge from another node in the graph. Continuing this process either leads to identifying a 0 in-degree node or to the situation where a last untouched node is reached but the graph is acyclic, so this node has to have in-degree 0. □

□

□

Assignment 2: Combinatorial Circuits
(Given Feb. 11., Due Feb. 18.)

20pt

Problem 2.4 (Trees)
The branching factor of a tree is the number of children every node has (except for the leaves). For example, binary trees have branching factor 2. We call a tree with branching factor b a b-tree. A fully balanced b-tree is a tree in which all leaves have the same depth. For all of the questions below, a full proof is expected:

1. How many leaves are there in a fully balanced b-tree with depth d? What about nodes?
2. What is the minimum depth a b-tree with $3n + 1$ nodes can have?
3. What about the maximum depth?

Solution:

1. **Proof**: Proof using the geometric progression formula

   P.1 Every node has b children, so there is 1 node on the root level, b on the next, b^2 on the next, and so on. On the last level there are b^d nodes, which are the leaves. Thus the overall number is $1 + b + ... + b^d = \frac{b^{d+1} - 1}{b - 1}$.

   P.2 It is the depth of the balanced tree (not necessairly fully balanced). Solve $3n + 1 = \frac{b^{d+1} - 1}{b - 1}$ for d and round up to the nearest integer: we get $\lceil \log_b(((3n + 1)(b - 1) + 1)) \rceil - 1$.

   P.3 A "chain" tree with every left-most node expanded into 3 children. Thus depth n. □

Problem 2.5 (DNF circuits and KV map optimization) 20pt

Design a combinational circuit for the following Boolean function by using KV map optimization:

X_1	X_2	X_3	$f_1(X)$	$f_2(X)$	$f_3(X)$
0	0	0	0	1	1
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1

Solution: The KV maps look like this:

f_1

	X_1X_2	$X_1\overline{X_2}$	$\overline{X_1X_2}$	$\overline{X_1}X_2$
X_3	F	T	T	F
$\overline{X_3}$	T	F	F	T

f_2

	X_1X_2	$X_1\overline{X_2}$	$\overline{X_1X_2}$	$\overline{X_1}X_2$
X_3	F	F	F	F
$\overline{X_3}$	T	T	T	F

f_3

	X_1X_2	$X_1\overline{X_2}$	$\overline{X_1X_2}$	$\overline{X_1}X_2$
X_3	T	T	F	F
$\overline{X_3}$	T	T	T	F

The minimal polynomials are: $f_1 = X_2\,\overline{X_3} + \overline{X_2}\,X_3$ $f_2 = X_1\,\overline{X_3} + \overline{X_2}\,\overline{X_3}$ $f_3 = X_1 + \overline{X_2}\,\overline{X_3}$

There are many possible ways to draw a circuit that implements these functions. Here is one option:

Problem 2.6 (Cyclic permutation detector) 25pt

Design a combinational circuit that detects whether a 4-bit number a is a cyclic permutation of another 4-bit number b. The circuit takes as input the two 4-bit numbers and has a single output X that should be 1 if the condition is met and 0 otherwise. You can use AND, OR, NOT, and XOR gates in your design. In addition state what is the depth of your circuit.

For instance, 1011 is a cyclic permutation of both 0111 and 1101 but not of 0101.

Note: n-bit binary number is a sequence of 1 or 0 with n elements ($n^{\{0,1\}}$).

Solution:

If we consider XOR to be a single gate then the depth is 6. If the depth of XOR is considered to be 3 then the total depth of the circuit is 8. In both cases the NOR gate is considered to be of depth 2.

Problem 2.7 (Is implication universal?) 20pt

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. □

□

Problem 2.8 (Converting to decimal in SML) 15pt

Write an SML function

to_int = fn : string -> int

that takes a string in binary, octal or hexadecimal notation and converts it to a decimal integer. If the string represents a binary number, it begins with 'b' (e.g. "b1011"), if it is an octal number - with '0' (e.g. "075") and if it is a hexadecimal number it begins with '0x' (e.g. "0x3A").

If the input does not represent an integer in one of these three forms raise the InvalidInput exception.

For example we have

to_int("b101010") -> 42

Solution:

```
exception InvalidInput;

fun reverse nil = nil
        | reverse (h::t) = reverse(t)@[h];

fun find_int(#"A") = 10
        |find_int(#"B") = 11
    |find_int(#"C") = 12
    |find_int(#"D") = 13
    |find_int(#"E") = 14
    |find_int(#"F") = 15
    |find_int(c) = if 0<= ord(c)- 48 andalso ord(c)-48 <=9 then ord(c)-48 else raise InvalidInput;

fun from_binary (nil) = 0
        | from_binary (h::l) = if 0<= ord(h)- 48 andalso ord(h)-48 <=1
        then ord(h) - 48 + 2 * from_binary(l) else raise InvalidInput;

fun from_octal (nil) = 0
        |from_octal (h::l) = if 0<= ord(h)- 48 andalso ord(h)-48 <=8
                then ord(h) - 48 + 8 * from_octal(l) else raise InvalidInput;

fun from_hexa (nil) = 0
        |from_hexa (h::l) = find_int(h) + 16 * from_hexa(l);

fun selector (#"0"::(#"x"::l)) = from_hexa(reverse(l))
        | selector (#"0"::l) = from_octal(reverse(l))
    | selector (#"b"::l) = from_binary(reverse(l))
    | selector _ = raise InvalidInput;

fun to_int number = selector(explode(number));

(*TEST CASES*)
val test1 = to_int("b101010")=42;
val test2 = to_int("052")=42;
val test3 = to_int("0x2A")=42;
val test4 = to_int("0x11A")=282;
```

```
val test5 = to_int("b101101")=45;
val test6 = to_int("12") = 0 handle InvalidInput => true| other => false;
val test7 = to_int("b12") = 0 handle InvalidInput => true| other => false;
val test8 = to_int("0x12H") = 0 handle InvalidInput => true| other => false;
val test9 = to_int("0129") = 0 handle InvalidInput => true| other => false;
val test10 = to_int("0-") = 0 handle InvalidInput => true| other => false;
```

Assignment 3: Adders, TCN
(Given Feb. 18., Due Feb. 25.)

35pt

Problem 3.9 (Conditional Sum Adder)
Draw the circuit of a Conditional Sum Adder (CSA) that adds two four-bit numbers. Go down to the level of elementary gates.

Problem 3.10 (Cost and depth of adders) 30pt

What is the cost and depth of an n-bit CCA? What about the n-bit CSA (for cost, big-O is enough)? Now what if we construct a new adder, that computes the two cases for the first half of the input just like CSAs do (and of course uses a multiplexer), but only does this once, and the $\frac{n}{2}$-bit adders are not also CSAs, but CCAs (so only one multiplexer is used overall) - what would the cost and depth of this adder be?

Solution: The CCA has depth $3n$ and cost $5n$, as shown in the slides. The CSA has depth $3\log(n) + 3$ and cost of complexity order n^{log3}. For the new adder, the depth is the one of the $\frac{n}{2}$-bit CCA, plus the $\frac{n}{2}$-bit multiplexer, which is 3. Thus the depth is $\frac{3n}{2} + 3$. The cost is that of three CCAs and one multiplexer, so $\frac{5n}{2} \cdot 3 + \frac{3n}{2} + 1$.

Problem 3.11 (TCN in SML) 35pt

Given the datatype: type tcn = int list write the following SML functions

- extend = fn : tcn -> int -> tcn which takes a number in Two's complement representation and an integer n and makes the tcn number n bits wide. Here n should always be bigger or equal to the current width of the number.
- int2tcn = fn : int -> int -> tcn which converts an integer number to a tcn number given the width.
- tcn2int = fn : tcn -> int which converts a tcn number to an integer number.

If in any of the functions a number can't fit into the requested number of bits raise the NumberDoesNotFit exception.

Note: A number's width is simply the number of bits that are used to represent that number.

As an example conisder

int2tcn 10 8 -> [0,0,0,0,1,0,1,0]

Solution:

```
type tcn = int list;
exception NumberDoesNotFit;

fun int2bin 0 = [0]
        | int2bin n = (int2bin (n div 2) ) @ [n mod 2];

fun negate bin = foldr (fn(c,p)=> (1-c)::p ) nil bin;

fun extend (num:tcn) (bits:int) :tcn =
        if bits < (length num)
                then raise NumberDoesNotFit
                else List.tabulate(bits-(length num), (fn _ => hd(num))) @ num;

fun int2tcn n (bits) : tcn =
        if n >= 0
        then extend ( int2bin n ) bits
        else extend ( negate( int2bin ((~n)-1) ) ) bits;

fun bin2int num = foldl (fn (c,p) => p*2+c ) 0 num;
fun tcn2int (num:tcn) =
        if hd(num) = 0
                then bin2int num
                else ~(bin2int (negate num) ) - 1;

(*TEST CASES*)
val test1 = (int2tcn 0 4) = [0,0,0,0];
val test2 = (int2tcn 1 4) = [0,0,0,1];
val test3 = (int2tcn 2 4) = [0,0,1,0];
val test4 = (int2tcn 3 4) = [0,0,1,1];
val test5 = (int2tcn 4 4) = [0,1,0,0];
```

```
val test6 = (int2tcn 5 4) = [0,1,0,1];
val test7 = (int2tcn 6 4) = [0,1,1,0];
val test8 = (int2tcn 7 4) = [0,1,1,1];
val test9 = (int2tcn ~1 4) = [1,1,1,1];
val test10 = (int2tcn ~2 4) = [1,1,1,0];
val test11 = (int2tcn ~3 4) = [1,1,0,1];
val test12 = (int2tcn ~4 4) = [1,1,0,0];
val test13 = (int2tcn ~5 4) = [1,0,1,1];
val test14 = (int2tcn ~6 4) = [1,0,1,0];
val test15 = (int2tcn ~7 4) = [1,0,0,1];
val test16 = (int2tcn ~8 4) = [1,0,0,0];
val test17 = (int2tcn ~10 4) = [1,0,1,0] handle NumberDoesNotFit => true| other => false;
val test18 = (int2tcn 16 4) = [1,0,1,0] handle NumberDoesNotFit => true| other => false;

val test19 = (extend [0] 3) = [0,0,0];
val test20 = (extend [1,0] 3) = [1,1,0];
val test21 = (extend [0,1,0] 3) = [0,1,0];
val test22 = (extend [1] 3) = [1,1,1];
val test23 = (extend [1,0,0] 3) = [1,0,0];
val test24 = (extend [1,0,1,0] 3) = [0] handle NumberDoesNotFit => true| other => false;

val test25 = tcn2int [0] = 0;
val test26 = tcn2int [0,0,0] = 0;
val test27 = tcn2int [0,0,1] = 1;
val test28 = tcn2int [1] = ~1;
val test29 = tcn2int [1,1,1,1] = ~1;
val test30 = tcn2int [1,1,1,1,1,1,1,1,0] = ~2;
val test31 = tcn2int [0,0,0,0,1,1,0,0] = 12;
val test32 = tcn2int [1,0,1,0] = ~6;
```

Assignment 4: Two's complement numbers, Sequential Logic Circuits, Memory
(Given Feb. 25., Due Mar. 4.)

30pt

Problem 4.12 (Average number of TCN bits)
Write an SML program to find out the average number of bits needed for the TCN representation of an 8-digits integer. The method you use is entirely up to you, but you need to return the result with 10^{-2} accuracy. Write a report explaining the method, attach you SML code and give the result together with a function call that reproduces it.

Solution:

```
type tcn = int list;
exception NumberDoesNotFit;
fun int2bin 0 = [0]
| int2bin n = (int2bin (n div 2) ) @ [n mod 2];
fun negate bin = foldr (fn(c,p)=> (1-c)::p ) nil bin;

fun size n = if n> 0 then length (int2bin n) else length (negate (int2bin (0-n)));

(*monte carlo simulation : pick n 8-digit numbers and average out the number of bits*)
val n = 1000000;
val randState =
            let
                val m = Date.minute(Date.fromTimeLocal(Time.now()))
                val s = Date.second(Date.fromTimeLocal(Time.now()))
            in
                Random.rand (m,s)
            end;


        (* random(l,h) returns a random integer, r, l<=r<=h
         Input must fullfill: l<=h *)
fun random (l,h) = if Random.randRange(0,1)randState = 0 then Random.randRange (l,h) randState
        else (0-(Random.randRange (l,h) randState));


fun ms 0 = 0
        | ms m = size(random(10000000,99999999)) + ms (m-1);

ms n;
(*27.02 is the answer*)

(*Important note: One needs to set the sample size high enough such that
the 2nd decimal does not flactuate with multiple runs - in this case 1000000*)
```

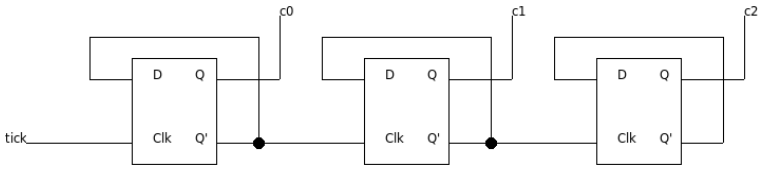
Problem 4.13 (Binary counters) 25pt

In the slides there is an implementation of a D-flipflop with an *enable* input. In practice a different version is more commonly used - the edge-trigerred D-flipflop. Here instead of an *enable* input there is a clock input (*clk*). The difference in operation is that the edge-trigerred D-flipflop only remembers the value of the D input at the one instant when the *clk* input switches from 0 to 1. If *clk* is constantly 0 or constantly 1 the flipflop will not change its state.

Using only such flipflops implement a 3-bit binary counter circuit. The circuit should have only one input 'tick' that will periodically change between 1 and 0. It should have three outputs that count the number of pulses on the input. After the counter counts to 111 it should continue from 000. You can assume the initial state of all flipflops is 0.

Note: For those of you who are curious here is how an edge-trigerred D-flipflop is built from NAND gates: `http://en.wikipedia.org/wiki/File:Edge_triggered_D_flip-flop.png`. If you're trying to understand this it will help to note that a real physical gate has a certain delay. When the input changes it takes some time (nanoseconds) for the output to react.

Solution:

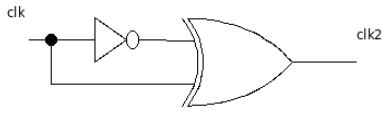


Problem 4.14 (Frequency multiplier) 15pt

Unlike the theoretical models we have discussed in the lecture real physical gates have a delayed reaction. As an example imagine that both inputs of an AND gate are 1 and correspondingly the output is also 1. Now if one of the inputs changes to 0 it will take a little time until the output also changes to 0.

Using this knowledge make a very simple circuit that has only one input (clk) and one output ($clk2$). The input is an alternating sequence of ones and zeroes, where each 1 or 0 lasts for 2 seconds. The output should also be an alternating sequence of ones and zeros but they should each last only for 1 second. You do not need to use any flipflops or memory elements, simple gates are enough. Assume that the propagation time between input and output for each type of gate (NOT, AND, OR, XOR NAND, NOR) is 1 second.

Solution: We pass the input signal through an inverter so that we get a new sequence that has a 1 second offset compared to the original. After that if we just use a XOR gate we will get the desired output.



Problem 4.15 (Combinational Circuit for Shift) 30pt

Suppose we had a device that takes as input 3 different signals of some electrical nature and, outputs processed combination of them on five channels. If the machine were turned off, then the output will be 0 on all channels.

Consider the following mixer function bellow as a simplified representation of this device (in the case it is turned on). Design a combinational circuit (using only NOT, AND and OR gates) that implements this mixer function. You can use an additional input variable that checks if the mixer is on. $f_{\text{mix}}\colon \mathbb{B}^3 \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}^5$ with

$$f_{\text{mix}}(\langle a,b,c\rangle, sw_1, sw_2) \begin{cases} \langle a,c,b,c,a\rangle & \text{if } s_1 = 0, s_2 = 0 \\ ,\langle c,a,0,a,c\rangle & \text{if } s_1 = 0, s_2 = 1 \\ ,\langle b,0,a,0,b\rangle & \text{if } s_1 = 1, s_2 = 1 \\ ,\langle a,0,c,b,c\rangle & \text{if } s_1 = 1, s_2 = 0 \end{cases}$$

Solution:

Assignment 5: Register Machine, Virtual Machine
(Given March 4., Due March 11.)

25pt

Problem 5.16 (sorting-by-selection)
Let $n \geq 1$ be stored in $P(0)$ and n numbers stored in $P(2) \dots P(n+1)$. Write an assembler program that performs a sorting by selection and outputs the result in $P(n+2) \dots P(2n+1)$. Write comments to each line of your code (like in the example codes from the slides). Uncommented code will not be considered.

Solution:

P	instruction	comment
0	LOAD 0	$ACC: = P(0) = n$
1	ADD 0	$ACC: = ACC + n$
2	MOVE ACC $IN2$	The recursion index. Initializing $IN2 = 2n$ (will come in handy when printing the results)
3	LOAD 0	$ACC: = n$
4	MOVE ACC $IN1$	$IN1: = n$ Initialising the sequence index. Outer loop starts.
5	LOADIN 1 1	
6	STORE 1	Initialize $MAX = P(1) = P(n+1)$
7	MOVE $IN1$ ACC	$ACC: = IN1 = n$ Starting inner loop
8	SUBI 1	
9	MOVE ACC $IN1$	$IN1 - -$
10	JUMP$_=$ $= 7$	if $IN1$ becomes 0 we have a MAX
11	LOADIN 1 1	$ACC: = P(IN1 + 1)$
12	SUB 1	$ACC: = ACC - MAX$
13	JUMP$_=$ < 2	if $P(IN1 + 1) < MAX$ jump
14	LOADIN 1 1	else $MAX = P(IN1 + 1)$
15	STORE 1	
16	JUMP -9	End inner loop (for one max)
17	LOAD 1	$ACC = MAX$
18	STOREIN 2 1	$P(IN2 + 1): = ACC = MAX$
19	LOAD 0	now we have to make the max we chose to be equal to 0, so we wouldn't find it again
20	MOVE ACC $IN1$	we test if the current number equals $P(1)$
21	LOADIN 1 1	
22	SUB 1	
23	JUMP$_=$ $= 5$	if we find the number we make it 0
24	MOVE $IN1$ ACC	
25	SUBI 1	
26	MOVE ACC $IN1$	
27	JUMP -6	
28	LOADI 0	
29	STOREIN 1 1	
30	MOVE $IN2$ ACC	
31	SUBI 1	
32	MOVE ACC $IN2$	$IN2 - -$
33	SUB 0	$IN2: = IN2 - n$
34	JUMP$_=$ > -20	if $IN2 - n > 0$ go back again. End big loop.
35	STOP 0	

Problem 5.17 (Fibonacci Numbers) 15pt

Assume the data stack initialized with `con` n for some natural number n. Write a $\mathcal{L}(\texttt{VM})$ 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
```

Problem 5.18 (Simulating REMA in SML) 20pt

Given the following declarations:

```
datatype register = acc | in1 | in2;
datatype instr = load of int | loadi of int | loadin1 of int | loadin2 of int |
                 store of int | storein1 of int | storein2 of int |
                 add of int | addi of int | sub of int | subi of int |
                 move of register*register| nop of int | stop of int |
                 jump of int | jumpe of int | jumpne of int |
                 jumpl of int | jumple of int | jumpg of int | jumpge of int;
type program = instr list;
type memory = int list;

(* This is the state of the machine. From left to right the values mean:
   PC register; ACC register; IN1 register; IN2 resigter; Memory cells*)
type state = int*int*int*int*(int list);
```

Write two SML functions:

- execute_instr : instr -> state -> state
- run : program -> memory -> memory

The first function takes an ASM instruction and the current state of the REMA as arguments and returns the new state after the instruction is executed. The second function takes a program and the initial configuration of the memory. It then simulates the program until a STOP 0 instruction is reached and returns the memory at that point. In both functions 'memory' is just a list of integers that represent the current state of the memory of the REMA. Once the initial list is supplied, during simulation its length shoudn't change.

Note: For this problem and the next it will be very helpful to use built-in SML functions. Make sure to check the forums for more info.

Solution:

```
(* Needed in order not to truncate output. *)
Control.Print.printDepth := 100;
Control.Print.printLength := 100;
Control.Print.stringDepth := 100;

datatype register = acc | in1 | in2;
datatype instr = load of int | loadi of int | loadin1 of int | loadin2 of int |
                 store of int | storein1 of int | storein2 of int |
                 add of int | addi of int | sub of int | subi of int |
                 move of register*register| nop of int | stop of int |
                 jump of int | jumpe of int | jumpne of int |
                 jumpl of int | jumple of int | jumpg of int | jumpge of int;
type program = instr list;
type memory = int list;

(* This is the state of the machine. From left to right the values mean:
```

```
   PC register; ACC register; IN1 register; IN2 resigter; Memory cells*)
type state = int*int*int*int*(int list);

(* returns a list identical to mem, where the element index is replaced with new_val *)
fun modify mem index new_val = List.take(mem,index) @ [new_val] @ List.drop(mem,index+1);

(* Data LOAD and STORE instructions *)
fun execute_instr (load(i)) ((pc_r,acc_r,in1_r,in2_r,mem):state) :state = (pc_r+1,List.nth(mem,i),in1_r,in2_r,mem)
  | execute_instr (loadi(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,i,in1_r,in2_r,mem)
  | execute_instr (loadin1(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,List.nth(mem,i+in1_r),in1_r,in2_r,mem)
  | execute_instr (loadin2(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,List.nth(mem,i+in2_r),in1_r,in2_r,mem)
        | execute_instr (store(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,modify mem i acc_r)
        | execute_instr (storein1(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,modify mem (i+in1_r) acc
        | execute_instr (storein2(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,modify mem (i+in2_r) acc

       (* Arithmetic instructions *)
        | execute_instr (add(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r+List.nth(mem,i),in1_r,in2_r,mem)
        | execute_instr (addi(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r+i,in1_r,in2_r,mem)
        | execute_instr (sub(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r-List.nth(mem,i),in1_r,in2_r,mem)
        | execute_instr (subi(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r-i,in1_r,in2_r,mem)

  (* The MOVE instruction *)
        | execute_instr (move(acc,in1)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,acc_r,in2_r,mem)
        | execute_instr (move(acc,in2)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,acc_r,mem)
        | execute_instr (move(in1,acc)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,in1_r,in1_r,in2_r,mem)
        | execute_instr (move(in1,in2)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in1_r,mem)
        | execute_instr (move(in2,acc)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,in2_r,in1_r,in2_r,mem)
        | execute_instr (move(in2,in1)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in2_r,in2_r,mem)
       (* Just for match completeness. *)
        | execute_instr (move(acc,acc)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (move(in1,in1)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (move(in2,in2)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)

  (* The STOP and NOP instructions. *)
        | execute_instr (stop(_)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r,acc_r,in1_r,in2_r,mem)
        | execute_instr (nop(_)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)
```

Solution:

```
  (* The JUMP instructions. *)
        | execute_instr (jump(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+i,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpe(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r = 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpne(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r <> 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpl(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r < 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumple(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r <= 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
```

```
                                                    else (pc_r+1,acc_r,in1_r,in2_r,mem)
      | execute_instr (jumpg(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r > 0
                                                    then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                    else (pc_r+1,acc_r,in1_r,in2_r,mem)
      | execute_instr (jumpge(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r >= 0
                                                    then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                    else (pc_r+1,acc_r,in1_r,in2_r,mem);


fun run_helper (p:program) (s:state) =
  let
    val ins = List.nth(p, #1 s);
  in
    if ins = stop 0 then s else run_helper p (execute_instr ins s)
  end;

fun run (nil:program) (mem:memory) :memory = mem
  | run p mem = #5 (run_helper p (0,0,0,0,mem));

(* Test Cases - From slides *)
val p1 = [load 0, store 2, load 1, store 0, load 2, store 1, stop 0] : program;
val mem1 = [4, ~10, 0] : memory;
val res1 = [~10, 4, 4] : memory;

val p2 = [load 1, add 2, add 3, store 4, stop 0] : program ;
val mem2 = [0,4,6,~2,10] : memory;
val res2 = [0,4,6,~2,8] : memory;

val p3 = [load 0, move (acc,in1) , load 1, storein1 0, stop 0] : program;
val mem3 = [5,10,0,0,0,0] : memory;
val res3 = [5,10,0,0,0,10] : memory;

val p4 = [load 1, move (acc,in1), load 2, move (acc,in2), load 0, jumpe 13,
          loadin1 0, storein2 0, move (in1,acc), addi 1, move (acc,in1), move (in2,acc),
          addi 1, move (acc,in2), load 0, subi 1, store 0, jump ~12, stop 0] : program;
val mem4 = [5,3,10,~1,~2,~3,~4,~5,~6,0,0,0,0,0,0] : memory;
val res4 = [0,3,10,~1,~2,~3,~4,~5,~6,0,~1,~2,~3,~4,~5] : memory;

val test1 = res1 = run p1 mem1;
val test2 = res2 = run p2 mem2;
val test3 = res3 = run p3 mem3;
val test4 = res4 = run p4 mem4;
```

Problem 5.19 (Simulating ASM files) 40pt

Building on the previous exercise write an SML function: simulate_asm_file : string -> string -> memory
The first argument is the name of an input file that contains an ASM program and its initial memory. The second string is the name of an output file. The simulate_asm_file should do the following:

1. It reads in the ASM file. This file contains one instruction per line. Empty lines are allowed. In addition comments can appear on any line (including lines with instructions). A comment begins with the semicolon symbol ';' and continues until the end of the line. The first line of the file is simply a list of integers separated by either tabs or spaces. This represents the initial memory of the machine.

2. After the input is read it is converted to the program datatype. Note that during this conversion you must take care of labels. As on the slides labels are of the form '< *sometext* >'. To simplify things a bit labels and instructions can not be on the same line. When a jump instruction indicates a label you should convert this to the proper numerical value.

3. Finally it simulates the provided ASM code and returns the final memory state. In addition before the execution of each instruction the current state of the REMA should be written on one line in the output file.

Solution:

```
(* Needed in order not to truncate output. *)
Control.Print.printDepth := 100;
Control.Print.printLength := 100;
Control.Print.stringDepth := 100;

datatype register = acc | in1 | in2;
datatype instr = load of int | loadi of int | loadin1 of int | loadin2 of int |
                 store of int | storein1 of int | storein2 of int |
                 add of int | addi of int | sub of int | subi of int |
                 move of register*register| nop of int | stop of int |
                 jump of int | jumpe of int | jumpne of int |
                 jumpl of int | jumple of int | jumpg of int | jumpge of int;
type program = instr list;
type memory = int list;

(* This is the state of the machine. From left to right the values mean:
   PC register; ACC register; IN1 register; IN2 resigter; Memory cells*)
type state = int*int*int*int*(int list);

exception InvalidLabel;
exception DuplicatingLabel;
exception InvalidInstruction of string;
exception NotAnInteger of string;
exception UndefinedLabel of string;
```

```
(* returns a list identical to mem, where the element index is replaced with new_val *)
fun modify mem index new_val = List.take(mem,index) @ [new_val] @ List.drop(mem,index+1);

(* Data LOAD and STORE instructions *)
fun execute_instr (load(i)) ((pc_r,acc_r,in1_r,in2_r,mem):state) :state = (pc_r+1,List.nth(mem,i),in1_r,in2_r,mem)
  | execute_instr (loadi(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,i,in1_r,in2_r,mem)
  | execute_instr (loadin1(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,List.nth(mem,i+in1_r),in1_r,in2_r,mem)
  | execute_instr (loadin2(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,List.nth(mem,i+in2_r),in1_r,in2_r,mem)
        | execute_instr (store(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,modify mem i acc_r)
        | execute_instr (storein1(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,modify mem (i+in1_r) acc
        | execute_instr (storein2(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,modify mem (i+in2_r) acc

        (* Arithmetic instructions *)
        | execute_instr (add(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r+List.nth(mem,i),in1_r,in2_r,mem)
        | execute_instr (addi(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r+i,in1_r,in2_r,mem)
        | execute_instr (sub(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r-List.nth(mem,i),in1_r,in2_r,mem)
        | execute_instr (subi(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r-i,in1_r,in2_r,mem)

  (* The MOVE instruction *)
        | execute_instr (move(acc,in1)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,acc_r,in2_r,mem)
        | execute_instr (move(acc,in2)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,acc_r,mem)
        | execute_instr (move(in1,acc)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,in1_r,in1_r,in2_r,mem)
        | execute_instr (move(in1,in2)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in1_r,mem)
        | execute_instr (move(in2,acc)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,in2_r,in1_r,in2_r,mem)
        | execute_instr (move(in2,in1)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in2_r,in2_r,mem)
        (* Just for match completeness. *)
        | execute_instr (move(acc,acc)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (move(in1,in1)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (move(in2,in2)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)

  (* The STOP and NOP instructions. *)
        | execute_instr (stop(_)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r,acc_r,in1_r,in2_r,mem)
        | execute_instr (nop(_)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+1,acc_r,in1_r,in2_r,mem)
```

Solution:

```
  (* The JUMP instructions. *)
        | execute_instr (jump(i)) (pc_r,acc_r,in1_r,in2_r,mem) = (pc_r+i,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpe(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r = 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpne(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r <> 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpl(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r < 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumple(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r <= 0
                                                                  then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                  else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpg(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r > 0
```

```
                                                                then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                else (pc_r+1,acc_r,in1_r,in2_r,mem)
        | execute_instr (jumpge(i)) (pc_r,acc_r,in1_r,in2_r,mem) = if acc_r >= 0
                                                                then (pc_r+i,acc_r,in1_r,in2_r,mem)
                                                                else (pc_r+1,acc_r,in1_r,in2_r,mem);

(* Returns a list of all lines in a file (stream). *)
fun get_lines istream =
  let
    val line = TextIO.inputLine (istream);
  in
    case line of
      NONE => nil
      | SOME(l) =>
        let
          val cl = explode l;
          val cl = List.take(cl, length cl - 1);
          val l = implode cl;
        in
          l :: (get_lines istream)
        end
  end;

(* Makes all characters lowercase *)
fun lowercase_lines lines = map ( String.map Char.toLower) lines;

(* Tokenizes each line *)
fun tokenize_lines lines =
  map (String.tokens (fn ch => (ch = #" " orelse ch = #"\t")) ) lines;

(* Removes the comments from a tokenized line. *)
fun remove_comments nil = nil
  | remove_comments (a::l) = if hd (explode a) = #";" then nil else a::(remove_comments l);

(* Removes empty lines and comments *)
fun filter_lines nil = nil
  | filter_lines (nil::l) = filter_lines l
  | filter_lines (a::l) =
      let
        val nocomm = remove_comments a;
      in
        if nocomm = nil then (filter_lines l) else (nocomm::(filter_lines l))
      end;
```

Solution:

```
(* Converts a string to an integer raising an exception if the operation fails. *)
fun to_int str =
  let
    val num = Int.fromString str;
  in
    case num of
      NONE => raise NotAnInteger(str)
      | SOME(n) => n
```

```
    end;

(* Interprets the first line of a file as the memory configuration of the REMA. *)
fun get_memory lines = (map to_int (hd lines) , tl lines);

(* Creates a table of labels and removes them from the lines. *)
fun create_label_table num table code nil = (code,table)
  | create_label_table num table code (l::lines) =
    let
      val s = String.concatWith "␣" l;
    in
      if hd (explode s) <> #"<"
        then (create_label_table (num+1) table (code @ [l]) lines)
        else
          if List.last (explode s) <> #">"
            then raise InvalidLabel
            else
              if List.exists (fn (ls,_) => ls=s) table
                then raise DuplicatingLabel
                else create_label_table num ((s,num)::table) code lines
    end;

(* Converts an argument of a jump function to relative offset. *)
fun get_offset str labels linenum =
  let
    val num = Int.fromString str;
  in
    case num of
      SOME(n) => n |
      NONE =>
        let
          val addr = List.find (fn (l,p) => l=str) labels;
        in
          case addr of
            NONE => raise UndefinedLabel (str) |
            SOME((l,p)) => p - linenum
        end
   end;
```

Solution:

```
  (* Translate LOAD and STORE instructions *)
fun translate_instr ["load",n] _ _ = load(to_int(n))
  | translate_instr ["loadi",n] _ _ = loadi(to_int(n))
  | translate_instr ["loadin1",n] _ _ = loadin1(to_int(n))
  | translate_instr ["loadin2",n] _ _ = loadin2(to_int(n))
  | translate_instr ["store",n] _ _ = store(to_int(n))
  | translate_instr ["storein1",n] _ _ = storein1(to_int(n))
  | translate_instr ["storein2",n] _ _ = storein2(to_int(n))

  (* Arithmetic instructions *)
  | translate_instr ["add",n] _ _ = add(to_int(n))
  | translate_instr ["addi",n] _ _ = addi(to_int(n))
```

```
  | translate_instr ["sub",n] _ _ = sub(to_int(n))
  | translate_instr ["subi",n] _ _ = subi(to_int(n))

  (* NOP and STOP *)
  | translate_instr ["nop",n] _ _ = nop(to_int(n))
  | translate_instr ["stop",n] _ _ = stop(to_int(n))

  (* translate the MOVE instruction *)
  | translate_instr ["move","acc","in1"] _ _ = move (acc,in1)
  | translate_instr ["move","acc","in2"] _ _ = move (acc,in2)
  | translate_instr ["move","in1","acc"] _ _ = move (in1,acc)
  | translate_instr ["move","in1","in2"] _ _ = move (in1,in2)
  | translate_instr ["move","in2","acc"] _ _ = move (in2,acc)
  | translate_instr ["move","in2","in1"] _ _ = move (in2,in1)

  (* JUMP instructions *)
  | translate_instr ["jump",n] labels line_num = jump ( get_offset n labels line_num)
  | translate_instr ["jumpe",n] labels line_num = jumpe ( get_offset n labels line_num)
  | translate_instr ["jump=",n] labels line_num = jumpe ( get_offset n labels line_num)
  | translate_instr ["jumpne",n] labels line_num = jumpne ( get_offset n labels line_num)
  | translate_instr ["jump<>",n] labels line_num = jumpne ( get_offset n labels line_num)
  | translate_instr ["jumpl",n] labels line_num = jumpl ( get_offset n labels line_num)
  | translate_instr ["jump<",n] labels line_num = jumpl ( get_offset n labels line_num)
  | translate_instr ["jumple",n] labels line_num = jumple ( get_offset n labels line_num)
  | translate_instr ["jump<=",n] labels line_num = jumple ( get_offset n labels line_num)
  | translate_instr ["jumpg",n] labels line_num = jumpg ( get_offset n labels line_num)
  | translate_instr ["jump>",n] labels line_num = jumpg ( get_offset n labels line_num)
  | translate_instr ["jumpge",n] labels line_num = jumpge ( get_offset n labels line_num)
  | translate_instr ["jump>=",n] labels line_num = jumpge ( get_offset n labels line_num)

  (* Raise an exception if this is an unknown instruction. *)
  | translate_instr line _ _ = raise InvalidInstruction (String.concatWith "␣" line);

fun translate_program _ nil _ = nil
  | translate_program line_num (ins::code) (labels) =
      (translate_instr ins labels line_num):: translate_program (line_num+1) code labels ;

fun state_to_string (pc_r,acc_r,in1_r,in2_r,mem) =
  Int.toString(pc_r) ^"\\t"^Int.toString(acc_r)^"\\t"^Int.toString(in1_r)^"\\t"^Int.toString(in2_r)^"|\\t"
  ^ foldl (fn (c,p) => p ^ "␣" ^ Int.toString(c) ) "" mem;
```

Solution:

```
fun run_helper_outfile (p:program) (s:state) ostream =
  let
    val _ = TextIO.output (ostream, (state_to_string s) ^ "\\n");
    val ins = List.nth(p, #1 s);
  in
    if ins = stop 0 then s else run_helper_outfile p (execute_instr ins s) ostream
  end;

fun run_outfile (nil:program) (mem:memory) (_) :memory = mem
  | run_outfile p mem ostream = #5 (run_helper_outfile p (0,0,0,0,mem) ostream);
```

```
fun simulate_asm_file in_filename out_filename =
  let
    val input = TextIO.openIn in_filename;
    val output = TextIO.openOut out_filename;
    val lines = get_lines input;
    val _ = TextIO.closeIn input;
    val lines = lowercase_lines lines;
    val lines = tokenize_lines lines;
    val lines = filter_lines lines;
    val (mem,lines) = get_memory lines;
    val (code,labels) = create_label_table 0 nil nil lines;
    val compiled = translate_program 0 code labels;
    val mem = run_outfile compiled mem output;
    val _ = TextIO.closeOut output;
  in
    mem
  end;
```

Solution:

```
(* TEST CASES *)
(* below please find the content of 4 input files that should be provided to the function *)

(*
(* in1.txt *)
4 -10 0
load 0
store 2
load 1
store 0
load 2
store 1
stop 0

(* in2.txt *)
0 4 6 -2 10
load 1
add 2
add 3
store 4
stop 0

(* in3.txt *)
5 10 0 0 0 0
load 0
move acc in1
load 1
storein1 0
stop 0

(* in4.txt *)
5 3 10 -1 -2 -3 -4 -5 -6 0 0 0 0 0 0
load 1
move acc in1
```

```
load 2
move acc In2
load 0

<loop>
jumpe <end>
loadin1 0
storein2 0
move in1 acc
addi 1
move acc in1
move in2 acc
addi 1
move acc in2
load 0
subi 1
store 0
jump <loop>

<end>
stop 0
*)

(* Now the actual simulation. *)
val test1 = simulate_asm_file "in1.txt" "out1.txt" = [~10, 4, 4];
val test2 = simulate_asm_file "in2.txt" "out2.txt" = [0,4,6,~2,8];
val test3 = simulate_asm_file "in3.txt" "out3.txt" = [5,10,0,0,0,10];
val test4 = simulate_asm_file "in4.txt" "out4.txt" = [0,3,10,~1,~2,~3,~4,~5,~6,0,~1,~2,~3,~4,~5];
```

Assignment 6: SW and static procedures
(Given March 11., Due March 18.)

20pt

Problem 6.20 (Stack drawing)
Write down a static procedure in $\mathcal{L}(\texttt{VM})$ that computes the factorial of a natural number ($(0)! = (1)!$). Make sure you comment what you are doing. Draw the stack evolution for calling your procedure on the argument 2.

Solution:

```
proc 1 21

;if n = 0
arg 1 cjp 13

;f(n−1)
con 1 arg 1 sub
call 0

;f(n−1)∗n
arg 1 mul
return

;if n=0 case
con 1 return
```

10
2

Problem 6.21 (Static Procedure for Binomial Coefficients) 20pt

Write a $\mathcal{L}(\texttt{VM})$ static procedure that computes the value of the binomial $C(n,k)$. Use the recursion formula:

$$C(n+1,k+1) = C(n,k+1) + C(n,k)$$

$$C(n,0) = C(0,0) = 1$$

$$C(0,n) = 0$$

Solution:

```
; C(n,k)
proc 2 46

; if k == 0 return 1
con 0 arg 2 leq cjp 5
con 1 return

; if n == 1 return 0
con 0 arg 1 leq cjp 5
con 0 return

; C(n−1,k)
arg 2
con 1 arg 1 sub
call 0

; C(n−1,k−1)
con 1 arg 2 sub
con 1 arg 1 sub
call 0

; return C(n−1,k) + C(n−1,k−1)
add return
```

Problem 6.22 (Simple While program on Fibonacci) 15pt

Write a Simple While Program that takes a number N and computes the N^{th} Fibonacci number. Then provide the Abstract Syntax for your code.

Show how the $\mathcal{L}$(VM) version of it looks like by compiling it.

Solution:

```
var n := N; var a := 0;
var b := 1; var c=b; ([ ("n", Con N), ("a", Con 0), ("b", Con 1), ("a", Con 1)],
while 2<= n do While(Leq(Con 2, Var"n"),
        c=b+a; Seq [Assign("c", Add(Var"b", Var"a")),
        a=b; Assign("a", "b"),
        b=c; Assign("b", "c"),
        n:=n−1; Assign("n", Sub(Var"n", Con 1))]
end ),
return b; Var"c")
```

VM code:

```
con N con 0 con 1; con 1
peek 0 con 2 leq cjp 22
peek 1 peek 2 add poke 3
peek 2 poke 1
peek 3 poke 2
con 1 peek 0 sub poke 0
jp −27
peek 3 halt
```

Problem 6.23 (Local variables in static procedures) 20pt

Show the assembler implementation of two new `VM` instructions that can be used to implement local variables:

- `lpeek` i - pushes on the stack the $i + 2^{nd}$ data cell after the current frame pointer
- `lpoke` i - pops the value on the stack and stores it in the $i + 2^{nd}$ data cell after the current frame pointer

---

Solution: `lpeek` is exactly like `arg` except that the fifth and sixth lines from the slides become:

label	instruction	effect	comment
	... `ADDI` 2 `ADD` 0 ...	$ACC := \text{FP} + 2 + i$	load local var position

`lpoke` is also quite similar (changes to `arg` are marked with *):

label	instruction	effect	comment
⟨`lpoke`⟩	`LOADIN` 1 1	$ACC := P(\text{VPC} + 1)$	load i
	`STORE` 0	$P(0) := ACC$	cache i
	`MOVE` $IN3$ ACC		
	`STORE` 1	$P(1) := \text{FP}$	cache FP
⟨∗⟩	`ADDI` 2		
⟨∗⟩	`ADD` 0	$ACC := \text{FP} + 2 + i$	load local var position
	`MOVE` ACC $IN3$	$\text{FP} := ACC$	move it to FP
⟨∗⟩	`LOADIN` 2 0	$ACC := P(\text{SP})$	load the top of the stack
⟨∗⟩	`STOREIN` 3 0	$P(\text{FP}) := ACC$	store the value in the local variable i
⟨∗⟩	`dec` $IN2$	$\text{SP} := \text{SP} - 1$	pop the stack
	`LOAD` 1	$ACC := P(1)$	load FP
	`MOVE` ACC $IN3$	$\text{FP} := ACC$	recover FP
	`MOVE` $IN1$ ACC		
	`ADDI` 2		
	`MOVE` ACC $IN1$	$\text{VPC} := \text{VPC} + 2$	next instruction
	`JUMP` ⟨jt⟩		jump back

Problem 6.24 (Static procedure for logarithm) 25pt

Write down a static procedure in $\mathcal{L}(\texttt{VM})$ that computes $f(x) = \lfloor \log_2(x) \rfloor$. This procedure should not be recursive. Use the new `lpeek` and `lpoke` instructions from the previous exercise. Is there something you do at the end of your procedure that is not part of your algorithm. If yes, then describe a more elegant way of doing that by modifying the behavior of an existing `VM` instruction.

Solution:

`proc` 1 34	$f(x)$
`con` 0 `con` 2	local variables n, y
`arg` 1 `lpeek` 1 `leq` `cjp` 16	if $y \leq x$
`con` 1 `lpeek` 0 `add` `lpoke` 0	$n := n + 1$
`con` 2 `lpeek` 1 `mul` `lpoke` 1	$y := y * 2$
`con` 0 `lpoke` 0 `add`	else: eliminate all excess elements on the stack
`return`	return n

At the end we have two variables left on the stack n and y. We need to eliminate y. Therefore we make it 0 and then add this to n. In this way only the final result is left on the stack.

A better way of implementing this for example is to make the return instruction take one argument which represents the number of excess variables on the stack. The implementation of this instruction can then take care of adjusting the stack pointer to the right position.

Assignment 7: Turing Machines and Problem Solving
(Given March 25, Due April 1)

15pt

Problem 7.25 (AND the Tape)

Design a TM that implements the n-ary AND operator on its tape: Started with a sequence of 0s and 1s on the tape, it writes the results at the end of this input and halts. For example, a tape with 111 on it will be transformed in 1111. Your TM needs to have at most 3 states, halting state included.

Note: For exercises about TM construction, please format the transition table according to the TM simulator at `http://ironphoenix.org/tril/tm/` (here you will also find some example programs). This way you will be able to check your "code" and your TAs will have an easier time grading.

Solution:

1,_ H,1,>
1,0 2,0,>
1,1 1,1,>
2,_ H,0,>
2,1 2,1,>
2,0 2,0,>

Problem 7.26: Design a Turing Machine that decides whether its input is of the type $0^n1^{2^n}$. The string is initially on the tape and the head of the TM is on the first, leftmost element of the string. If the input is of the correct type the machine should halt and the current tape position should contain a 1. Otherwise the machine should halt and the current tape position should contain a 0. 30pt

Solution: The ideea of the following solution is that the input string will be deleted until we will come to an empty string if the initial input is accepted. The deletion is done by halfing the 1s each time a 0 is deleted. Also, between the 0s and 1s the number of blank symbols shouldn't be let to be arbitrary. The following set of productions provide the solution of the problem by following the above outline (q_2 is the final accepting state and B is the blank symbol):

state	read	write	move	new state
q_0	B	B	R	q_1
q_0	0	0	S	q_3
q_1	B	B	L	q_2
q_3	0	0	R	q_3
q_3	1	B	R	q_4
q_4	1	1	R	q_5
q_5	1	B	R	q_4
q_5	B	B	L	q_6
q_6	1	B	L	q_7
q_7	1	1	L	q_7
q_7	B	1	R	q_8
q_7	0	1	L	q_9
q_8	1	1	R	q_8
q_8	B	B	S	q_5
q_9	B	B	R	q_{10}
q_9	0	0	L	q_{11}
q_{10}	1	B	R	q_0
q_{11}	0	0	L	q_{11}
q_{11}	B	B	R	q_0

Problem 7.27 (Halting Reductions) 30pt

The fact that a TM cannot decide if another TM halts on a given input is not the only limit of computation. There are a lot of other things TM's cannot do, and the halting problem can be used to prove this. This process is called "reduction to the halting problem": for proving that a TM cannot decide a certain a property P, assume that it could and then use it to construct another TM that can decide the halting problem (i.e. to decide if some TM halts on some given input).

For the following statements, provide a proof by reduction to the halting problem or a counterexample:

- No TM can decide in general whether another TM halts on all inputs.
- TM can decide in general whether another TM uses all its states in the computation on a given input x.

Solution:

- Construct K such that it halts on every input but x, and on x it simulates N. Then use M on K, and if the output is yes, then it means N halted on x, otherwise no.
- Construct K such that it simulates N on x and if it halts, then it goes through all the states of N. This means that N halting on x is equivalent to K uses all its states on x (since if N doesn't halt, then the halting state will not be used). Then run M on K.

Problem 7.28 (Problem formulation) 25pt

You and your roommate just bought an 8 liter jug full of beer. In addition you have two smaller empty jugs that can hold 5 and 3 liters respectively. Being good friends you want to share the beer equally. For this you need to split the amount in two separate jugs and each should contain exactly 4 liters. Write a formal description of this problem. What is one possible solution? What is the cost of your solution?

Solution:

We encode the states as three digits where the first digit is the amount of beer in the 8 liter jug, the second digit is the amount in the 5 liter jug and the last digit is the amount in the 3 liter jug.

- Initial state: 800
- Actions:
  1. $pour8in5$
  2. $pour8in3$
  3. $pour5in3$
  4. $pour5in8$
  5. $pour3in5$
  6. $pour3in8$

  Each of these actions pours beer from one jug to anonther until either the first jug is empty or the second jug is full. The corresponding successor function S can be derived from these actions.
- Goal test: $x = \mathbf{440}$
- Path cost: The amount of actions we perform to reach the solution.

A sample solution is:

$$[pour8in5, pour5in3, pour3in8, pour5in3, pour8in5, pour5in3, pour3in8]$$

$$800 \to 350 \to 323 \to 620 \to 602 \to 152 \to 143 \to 440$$

It has a cost of 7.

Assignment 8: Problems and Searching
(Given April 01, Due April 15)

30pt

Problem 8.29 (Implementing Search)
Implement the depth-first and breadth-first search algorithms in SML. The corresponding functions dfs and bfs 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 a pair of two elements:

- a list of actions that reaches the goal state from the initial state
- the goal state

The signatures of the two functions should be:

```
dfs : 'a -> ('a -> ('b * 'a) list) -> ('a -> bool) -> 'b list * 'a
bfs : 'a -> ('a -> ('b * 'a) list) -> ('a -> bool) -> 'b list * 'a
```

where 'a is the type of states and 'b is the type of actions.

In case of an error or no solution found raise an InvalidSearch exception.

Solution:

```
exception InvalidSearch;
val tick = false; (* used for debugging *)

local

    fun add_actions x nil = nil
        | add_actions x ((a,s)::l) = (x @ [a],s)::(add_actions x l);

    fun depthFirst_strategy nil next = raise InvalidSearch
        | depthFirst_strategy ((a,s)::l) next = ( add_actions a (next s) ) @ l;

    fun breadthFirst_strategy nil next = raise InvalidSearch
        | breadthFirst_strategy ((a,s)::l) next = l @ ( add_actions a (next s) );

    fun sl strategy nil next goal = raise InvalidSearch
        | sl strategy ((a,s)::l) next goal =
        let
        val _ = if tick then print "#" else print "";
```

```
        in
                        if goal(s)
                then (a,s)
                else
                        let
                        val new_fringe = strategy ((a,s)::l) next;
                in
                        sl strategy new_fringe next goal
                end
        end;

        fun search strategy i next goal =
                if goal(i)
                then (nil,i)
        else sl strategy (add_actions nil (next i)) next goal;
in
        fun dfs i next goal = search depthFirst_strategy i next goal;
        fun bfs i next goal = search breadthFirst_strategy i next goal;

end;
```

Solution:

```
(* TEST CASES *)

datatype action = a1to2 | a1to4 | a1to5 | a2to3 | a4to5 | a4to6 | a5to1 | a5to7 | a3to6;
datatype state = one | two | three | four | five | six | seven;

fun next1(one) = [(a1to2,two),(a1to4,four),(a1to5,five)]
 | next1(two) = [(a2to3,three)]
 | next1(three) = [(a3to6,six)]
 | next1(four) = [(a4to5,five),(a4to6,six)]
 | next1(five) = [(a5to1,one),(a5to7,seven)]
 | next1(six) = []
 | next1(seven) = [];

fun next2(one) = [(a1to2,two),(a1to4,four),(a1to5,five)]
 | next2(two) = [(a2to3,three)]
 | next2(three) = [(a3to6,six)]
 | next2(four) = [(a4to5,five),(a4to6,six)]
 | next2(five) = [(a5to7,seven),(a5to1,one)]
 | next2(six) = []
 | next2(seven) = [];

fun goal1(six) = true
 | goal1(_) = false;

fun goal2(four) = true
 | goal2(three) = true
 | goal2(_) = false;

fun goal3(seven) = true
 | goal3(_) = false;
```

```
val test4 = bfs one next1 goal1 = ([a1to4,a4to6],six);
val test5 = dfs one next1 goal1 = ([a1to2,a2to3,a3to6],six);
val test6 = bfs one next1 goal2 = ([a1to4],four);
val test7 = dfs one next1 goal2 = ([a1to2,a2to3],three);
val test8 = bfs one next2 goal3 = ([a1to5,a5to7],seven);
val test9 = dfs one next2 goal3 = ([a1to4,a4to5,a5to7],seven);
val test10 = bfs one next1 goal3 = ([a1to5,a5to7],seven);
val test11 = dfs one next1 goal3; (*should run endlessly*)
```

Problem 8.30: Describe a state space in which iterative deepening search performs much worse than depth-first search (for example $O(n^2)$ vs. $O(n)$). 20pt

---

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:

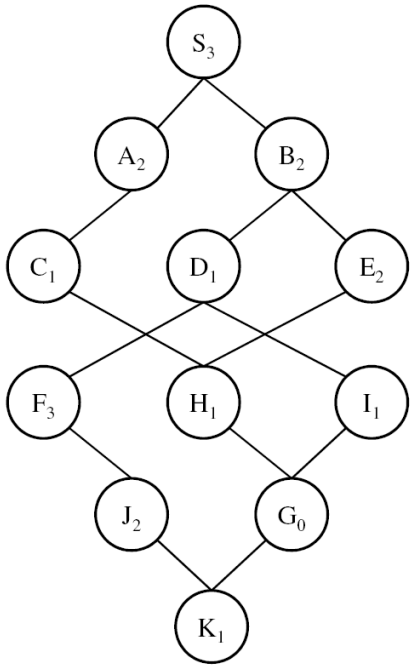
- $b = 100$
- $d = 33$
- $m = 36$

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 8.31 (Moving a Knight) 20pt

Consider the problem of moving a knight on a 3x4 board, with start and goal states labeled as S and G in the figure below. The search space can be translated into the following graph. The letter in each node is its name and you do not need to worry about its subscript for now.

S_3	H_1	D_1	K_1
I_1	J_2	A_2	E_2
C_1	B_2	G_0	F_3



Make the following assumptions:

- The algorithms do not go into infinite loops (i.e. once a node appears on a path, it will not be considered again on this path)
- Nodes are selected in alphabetical order when the algorithm finds a tie.

Write the sequence of nodes in the order visited by the specified methods (until the goal is reached). Note: You may find it useful to draw the search tree corresponding to the graph above.

- DFS
- BFS

Solution:

- DFS : S A C H E B D F J K G
- BFS : S A B C D E H F I G

Problem 8.32 (Sudoku) 30pt

					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8				4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

This question will give you an excuse to play Sudoku (see www.websudoku.com for explanation) while doing homework. Consider using search to solve Sudoku puzzles: You are given a partially filled grid to start, and already know there is an answer.

- Define a state representation for Sudoku answer search. A state is a partially filled, valid grid in which no rows, column, or 3x3 square contains duplicated digits. Also specify what transitions would be.
- If the puzzle begins with 28 digits filled, what is L, the length of the shortest path to goal using your representation?
- On a typical PC, which search algorithm would you choose: BFS, DFS or IDS? Why?

Solution:

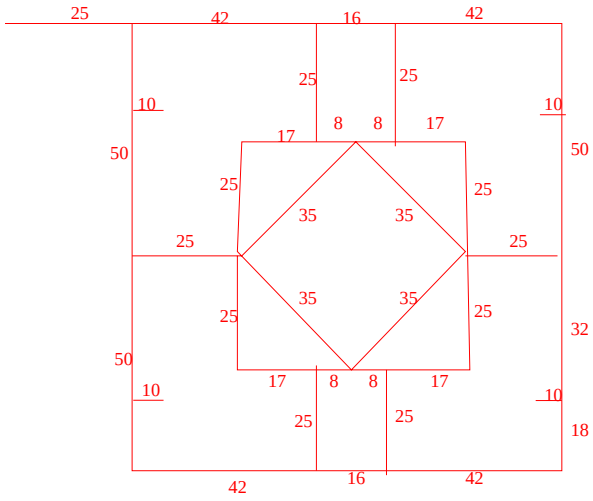
- A 9x9 matrix whose elements are 1 to 9 or 0 as empty. Transitions are any valid filling of an empty cell. Only valid matrices (no duplication in rows, column, or 3x3 squares) are allowed.
- $L = 81 - 28 = 53$
- DFS is the most suitable, and actually almost all Sudoku search program use DFS. B, the branching factor, can be in the range of 9×53 so $B >> L$. Hence BFS and IDS will have serious memory problems on a typical PC. Since L is fixed so $L = L_max = L_min$, DFS is the only choice here.

Assignment 9: Informed Search
(Given April 15, Due April 22)

35pt

Problem 9.33 (A^* **search on Jacobs campus**)
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 9.34 (Monotone heuristics) 25pt

Let $c(n, a, n')$ be the cost for a step from node n to a successor node n' for an action a. A heuristic h is called *monotone* if $h(n) \leq h(n') + c(n, a, n')$. Prove or refute that if a heuristic is monotone, it must be admissible. Construct a search problem and a heuristic that is admissible but not monotone. Note: For the goal node g it holds $h(g) = 0$. Moreover we require that the goal must be reachable and that $h(n) \geq 0$.

Solution: For the heuristic h to be admissible we have to show that $h(x)$ is less or equal the minimum coast to a goal state.

Let n_1 any node different from the goal node g. Suppose $< n_1, n_2, \ldots, n_p, g >$ is the minimum cost path from n_1 to g. Its cost is $C = c(n_1, a_1, n_2) + c(n_2, a_2, n_3) \ldots + c(n_p, a_p, g)$. Using $h(n) - h(n') \leq c(n, a, n')$ we get $C \geq h(n_1) - h(n_2) + h(n_2) - h(n_1) + \ldots + h(n_p) - h(g) = h(n_1) - h(g) = h(n_1)$. Hence we have proven that $h(n_1)$ is admissible.

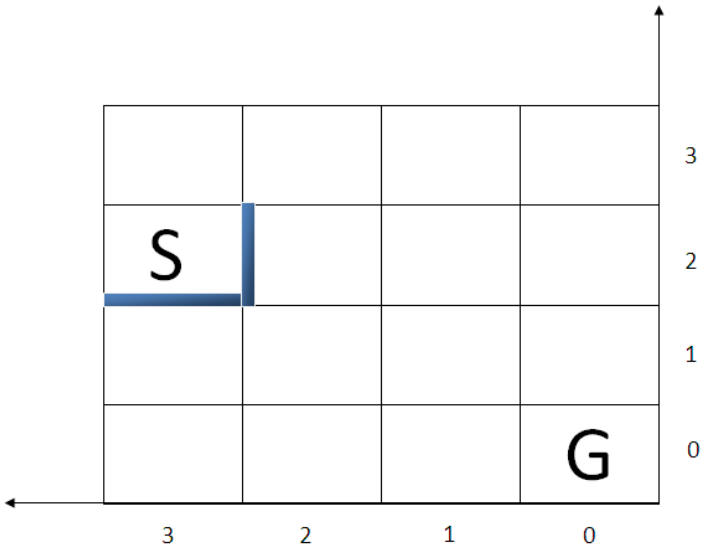
We consider the minimum distance search problem with three cities A, B, G where G is the goal city and the distances are $dist(A, B) = 2$ and $dist(B, G) = 100$. The heuristic $h(A) = 6, h(B) = 3, h(G) = 0$ is admissble since $h(A) < dist(A, B) + dist(B, G)$. But is is not monotone since $h(A) > h(B) + dist(A, B)$.

Problem 9.35 **(A^* vs BFS)** 10pt

Does A^* search always expands fewer nodes than BFS?

Solution: No. With a bad heuristic, A^* can be forced to explore the whole space, just like BFS.

Problem 9.36 (A Good Old Friend, the Maze) 30pt



Given a maze like the one above, consider using search to find the way from start to goal. The shaded areas are walls. You start from S and can only go left, right, up or down (unless there is a wall). All movements cost the same. The heuristic function is the Manhattan distance, $h = |x_1 - x_2| + |y_1 - y_2|$. For the following questions, explanations are required (simple answer is not enough).

1. Is this an admissible heuristic for A^* for the maze problem?
2. Is it an admissible heuristic if you can move in 8 directions instead of 4 (so also diagonally), if any movement still costs the same?
3. Which performs better with this heuristic, A^* or simple Greedy?
4. For the case of moving in all 8 directions, is the Euclidean distance, $h_e = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, admissible?
5. For the case of moving in all 8 directions, provide an admissible heuristic that is different from h and h_e, call it h_1, such that h_1 is non-trivial (non-constant and not the hardcoded actual cost).
6. Getting back to the 4 direction movement, is h_e more efficient for A^* than h?

Solution:

1. Yes, the Manhattan distance always underestimates the cost - in the case of walls constricting the path it will be strictly smaller than the actual cost.

2. No, it overestimates. In the example above, S has Manhattan distance 5 but can reach the goal in 4 steps.

3. A^* doesn't get stuck like Greedy.

4. No, because the position diagonally next to the goal has a cost of 1 and a Euclidean distance of $\sqrt{2} > 1$

5. Divide the Euclidean distance by $\sqrt{2}$ to fix the problem.

6. In 4 moves, Manhattan is a closer estimate of the real cost and will thus perform better.

Assignment 10: Local Search
(Given April 22, Due April 29)

30pt

Problem 10.37 (Implementing simulated annealing)
Write an SML function that implements the simulated annealing algorithm to find the x value where a function $f(x)$ has a maximum. Your function should take the following arguments:

- f : real−>real the SML implementation of $f(x)$
- (a,b) : real∗real an interval $[a; b]$ in which to search for the maximum
- schedule : int−>real a function that maps time steps to temperature values

For example the maximum of $f(x) = -(x-2)^2$ in $[0.0; 5.0]$ is at $x = 2.0$. Given a good temperature schedule your implementation should be able to compute the maximum of $sin(x)$ with an accuracy of 0.0001. Show this at the end of your program by computing the maximum of $sin(x)$ in the interval $[0.0; 5.0]$.

The complete signature of the function should look like this:

find_max : (real −> real) −> real ∗ real −> (int −> real) −> real

---

Solution:

```
val min_temp = 0.000001;

(* First set random seed. *
val rand_state =
  let
                val now = Time.toMicroseconds ( Time.now () );
        val x = IntInf.div(now,1000);
        val y = IntInf.toInt (IntInf.mod(x,10000));
        in
                Random.rand (y,y)
        end;

(* picks a random point in the interval [current−eps;current+eps] with a lower limit of a and
   an upper limit of b*)
fun get_random_successor current a b eps =
  let
    val random_num = Random.randReal rand_state;
    val upper = if ( current + eps) > b then b else current + eps;
    val lower = if (current − eps) < a then a else current − eps;
  in
    lower + ( random_num ∗ (upper−lower) )
  end;

(* Choses the next state based on the current state the temperature and the energy difference. *)
fun pick_with_probability current next deltaE temp =
```

```
  if deltaE > 0.0
  then next
  else
    if (Random.randReal rand_state) < Math.exp(deltaE/temp)
    then next
    else current;

(* Uses the simulated annealing algorithm to find the maximum of f in the interval [a,b] given
   the specified schedule. Furthermore this function must know the current solution, time and an
   epsilon value. The epsilon value is used to limit the neighborhood in which successor states
   can be chosen.*)
fun sim_ann f a b schedule time current eps =
  let
    val temp = schedule time;
    val next = get_random_successor current a b eps;
    val deltaE = (f next) - (f current);
  in
    if temp < min_temp
      then current
      else sim_ann f a b schedule (time+1) (pick_with_probability current next deltaE temp) eps
  end;

(* Uses the simulated annealing algorithm to find the maximum of f in the interval [a,b] given
   the specified schedule. *)
fun find_max f (a,b) schedule = sim_ann f a b schedule 0 ((a+b)/2.0) ((b-a)/10.0);

(* TESTING *)
fun schedule_lin time = 2.0 - 0.0002 * real(time);
fun schedule_exp time = Math.pow(0.95, real(time) ) * 50.0;

fun compute_num_steps schedule time =
  if schedule time < min_temp then time else compute_num_steps schedule (time+1);

val test_lin = Math.sin (find_max Math.sin (0.0,5.0) schedule_lin);
val test_lin_ok = (1.0 - test_lin) < 0.0001;
val test_lin_steps = compute_num_steps schedule_lin 0;

val test_exp = Math.sin (find_max Math.sin (0.0,5.0) schedule_exp);
val test_exp_ok = (1.0 - test_exp) < 0.0001;
val test_exp_steps = compute_num_steps schedule_exp 0;
```

Problem 10.38 (Simulated annealing schedules) 15pt

In the simulated annealing algorithm one has to choose a temperature schedule. Two possible schedules are:

- **The linear cooling scheme:** $T_{k+1} = T_k - \alpha = T_0 - (k+1) * \alpha$
- **The exponential cooling scheme:** $T_{k+1} = \alpha T_k = \alpha^{k+1} T_0$ where $\alpha < 1.0$ (the typical value is 0.95, but this really depends on the problem - and the smaller this is, the less iterations you will have).

The exponential cooling scheme typically performs better. Explain why this might be the case. To help you with this you should do an experiment where you try to achieve the desired accuracy in the pevious question by using both a linear and an exponential schedule.

Solution: It is important that near the end of our search we focus on only good solutions. By this time ideally we should be in the region of the global maximum and we should use the last iterations in order to find an optimal solution within the current "hill". The exponential cooling schedule allows just that. In the beginning it is still possible to make big jumps therefore escaping local maxima but at the end it leaves quite a lot of time where the algorithm tries to fine tune the current solution (because with low temperatures, the probability of accepting a worse state is very small, which means the state improves or stays the same at each iteration, thus allowing for a (randomized) hill-climbing like search for the top of the hill). The exponential schedule allows for more time at smaller temperatures.

To achive similar performance with the linear schedule we typically need a lot more iterations.

Problem 10.39 (Easter Bunnies in Boxes) 30pt

Imagine there are n Easter bunnies and n different coloured boxes, and each bunny has specific color preferences and will like their box on a scale of 1 to 10. We want to makes as many bunnies as happy as we can, so the overall fitness of an assignment of bunnies in boxes will be the sum of how much each bunny likes its box. An assignment is admissible if each bunny has exactly 1 box. Think about applying Genetic Algorithms for this problem: your task is to come up with an encoding that allows only admissible states and with crossover and mutation operators that preserve admissibility. Don't take the term crossover too literally though - it is not a must that you split the chromosomes and cross over their parts, you can think about the concept of reproduction in general. Similarly for mutation.

Solution: Encoding: An n-permutation (e.g. for 8, 12376548) Crossover: There are many ways to do this, one of them is to compose the 2 permutations - permutations are functions and they can be composed (apply the first one and then the second one), yielding a permutation. E.g. composing 132 with 213 yields 312 (or 231 depending on the order, this is up to convention). Mutation: Compose with any 2-cycle permutation (i.e. one that switches 2 of the entries). E.g 12376548 can become 21376548.

Problem 10.40 (Similarities among algorithms) 25pt

The lecture on Local Search has introduced you to Hill Climbing, Local Beam Search, Simulated Annealing and Genetic Algorithms. There is also an important variation of Hill Climbing you should know about - Randomized Hill Climbing is when you choose a random neighbor rather than the best one, and you select it only if its fitness is better.

1. For what temperature schedule is Simulated Annealing the same algorithm as Randomized Hill Climbing? Why?

2. For what population size is Genetic Algorithm the same algorithm as Randomized Hill Climbing? Why? Consider GAs that always keep one or more of the fittest chromosomes in the new population - this is called elitism). See slide 268 for the pseudocode of GA.

Solution:

1. When the temperature is always very very small (0 for all practical purposes), SA will create a random neighbor and only accept it if it is better, since the probability of accepting a worse state will be practically 0. This is then the same as RHC.

2. When the population size is 1, GA will not crossover and will just mutate (which is the same as selecting a random neighbor). Then, due to elitism (always keeping the fittest individual), it will only preserve it in the population (of 1 chromosome) if it is better - exactly like RHC.

Assignment 11: ProLog
(Given April 29, Due May 6)

30pt

Problem 11.41 (Family relations)

Using the following `ProLog` predicates:

- woman(X)
- man(X)
- mother_of(X,Y) X is the mother of Y
- father_of(X,Y) X is the father of Y

write a sample knowledge base that contains facts about the family relations of several people. Afterwards define rules for the following predicates and test them on your knowledge base:

- sister_of(X,Y) X is the sister of Y
- brother_of(X,Y) X is the brother of Y
- sibling_of(X,Y) X is the sibling of Y
- grandma_of(X,Y) X is the grandmother of Y
- grandpa_of(X,Y) X is the grandfather of Y
- uncle_of(X,Y) X is the uncle of Y
- aunt_of(X,Y) X is the aunt of Y

Solution:

```
woman(alice).
woman(sally).
woman(kate).
woman(jessica).
woman(marry).
man(john).
man(vincent).
man(patrick).
man(kevin).

mother_of(sally, kate).
mother_of(sally, vincent).
mother_of(susan,jessica).
```

```
mother_of(marry,patrick).
mother_of(alice,marry).
mother_of(sally,kevin).
father_of(john, kate).
father_of(john, vincent).
father_of(vincent,jessica).
father_of(vincent,patrick).
father_of(john,kevin).

% ------------------------------------------------------------------
% Here are the relation predicates

parent_of(X,Y) :- mother_of(X,Y).
parent_of(X,Y) :- father_of(X,Y).
sister_of(X,Y) :- woman(X), parent_of(Z,X), parent_of(Z,Y).
brother_of(X,Y) :- man(X), parent_of(Z,X), parent_of(Z,Y).
sibling_of(X,Y) :- sister_of(X,Y).
sibling_of(X,Y) :- brother_of(X,Y).
grandma_of(X,Y) :- mother_of(X,Z), parent_of(Z,Y).
grandpa_of(X,Y) :- father_of(X,Z), parent_of(Z,Y).
uncle_of(X,Y) :- brother_of(X,Z), parent_of(Z,Y).
aunt_of(X,Y) :- sister_of(X,Z), parent_of(Z,Y).

% ------------------------------------------------------------------
% Test queries

% The following should result in a YES
?- grandma_of(sally, jessica).
?- grandpa_of(john, jessica).
?- sibling_of(jessica, patrick).
?- sibling_of(patrick, jessica).
?- brother_of(patrick, jessica).
?- sister_of(jessica, patrick).
?- aunt_of(kate, jessica).
?- uncle_of(kevin,jessica).
?- brother_of(vincent,kate).
?- brother_of(vincent,kevin).
?- brother_of(kevin,vincent).
?- sister_of(kate,kevin).
?- sister_of(kate,vincent).
?- sibling_of(kate,vincent).
?- sibling_of(kate,kevin).
?- sibling_of(vincent,kevin).
?- sibling_of(vincent,kate).
?- sibling_of(kevin,kate).
?- sibling_of(kevin,vincent).
?- grandma_of(alice,patrick).
?- aunt_of(kate,patrick).
?- uncle_of(kevin,patrick).
?- grandma_of(sally,patrick).
?- grandpa_of(john,patrick).
```

```
% The following should FAIL
?- grandma_of(alice,jessica).
?- sibling_of(susan,marry).
?- sibling_of(susan,vincent).
?- sibling_of(kevin,john).
?- brother_of(jessica,patrick).
```

Problem 11.42 (Fraction representation in ProLog) 30pt

Propose one way of representing fractions of the type $\frac{x}{y}$ where $x, y \in \mathbb{Z} y \neq 0$ in ProLog. Write an infinite knowledge base that is able to represent any such fraction and give an example of how $\frac{1}{2}, -\frac{5}{3}$ and 0 can be realized with your representation.

Solution:

```
pos( one ).
pos( s(X) ) :- pos(X).
neg( n(X) ) :- pos(X).
not_zero( X ) :- pos(X).
not_zero( X ) :- neg(X).
int( zero ).
int( X ) :- not_zero(X).
frac( X,Y ) :- int(X), not_zero(Y).

% 1/2 would be:
?- frac(one,s(one)).

% -5/3 would be:
?- frac(n(s(s(s(s(one))))),s(s(one)) ).
% or:
?- frac(s(s(s(s(one)))),n(s(s(one))) ).

% There are a lot of options for 0:
?- frac(zero, one ).
?- frac(zero, n(s(s(s(s(s(one)))))) ).
```

Problem 11.43 (Paths in a Graph) 30pt

Given a directed graph, represented by edge(from, to) facts, write a predicate trip(A, B, L) that succeeds if the node B is accessible from A via the intermediate nodes L (an ordered list of nodes).

Note: It is not required to avoid cyclic trips.

Solution:

```
trip(A, B, []) :- edge(A, B).
trip(A, B, [H | T]) :- edge(A, H), trip(H, B, T).
```

Problem 11.44 (Who is jealous?) 10pt

Given the following simple knowledge base:

```
loves(vincent,mia).
loves(mary,joe).
loves(joe,mary).
loves(anthony,mia).
loves(donny,mia).
loves(mia,joe).
knows(vincent,donny).
knows(joe,anthony).
knows(mia,vincent).
knows(vincent,joe).
knows(mary,mia).
knows(donny,joe).

jealous(X,Y) :− loves(X,Z),loves(Y,Z),knows(X,Y).
jealous(X,Y) :− loves(X,Z),loves(Z,Y),knows(X,Y).
```

Suppose we pose the query: ?−jealous(vincent,X). What would `ProLog` return? What about: ?−jealous(Y,W). What do you have to type to get all the jealous pairs and how do you know if there are any more left? Show how `ProLog` would answer the first question using Backtracking search (see slide 261 for example). Test your intuition.

Solution:

The query works around these lines:

```
?− jealous(vincent,X).
?− loves(vincent,Z),loves(X,Z),knows(vincent,X)
Z = mia
?− loves(X,mia).
X = anthony
?− knows(vincent,anthony).
FAIL
X = donny
?− knows(vincent,donny).
X = donny
Yes
```

In order to get all the jealous pairs you first type ?−jealous(Y,W). This returns you the first pair found. Then type ?− ; that stands for "are there any more of that?". `ProLog` either will give you a new solution to the query or answer "No" if there are no solutions left. The jealous pairs are: (vincent,donny), (vincent,joe), (mary,mia) and (donny, joe).

Assignment 12: Practice tasks
(Given May 6, Due May 13)

20pt

Problem 12.45 (Adding numbers)

Write a ProLog predicate add(A,B,S) where A+B=S. A,B and S are positive integers represented as lists.

For instance we have

```
add([1,2,4],[3,9],X).
X = [1,6,3] .
```

Solution:

```
append([], L, L).
append([X|R], L, [X|S]) :- append(R, L, S).

reverse([], []).
reverse([X|R], L) :- reverse(R, S) , append(S, [X], L).

add_rev([],[],[],0).
add_rev([],[],[C],C).
add_rev([A|AA],[],[S|SS],C) :- S is (A + C) mod 10, X is (A+C)//10, add_rev(AA,[],SS,X).
add_rev([],[B|BB],[S|SS],C) :- S is (B + C) mod 10, X is (B+C)//10, add_rev([],BB,SS,X).
add_rev([A|AA],[B|BB],[S|SS],C) :- S is (A + B + C) mod 10, X is (A+B+C)//10, add_rev(AA,BB,SS,X).

add(A,B,S) :- reverse(A,AA), reverse(B,BB), reverse(S,SS), add_rev(AA, BB, SS, 0).

% Testing

% True queries
?- add([1,2,3],[6,2,2],[7, 4, 5]).
?- add([5,7,8],[8,4,2],[1, 4, 2, 0]).

% False queries
?- add([1,2,3],[6,2,2],[7, 4, 6]).
?- add([5,7,8],[8,4,2],[1, 4, 2, 1]).
```

Problem 12.46 (Number representation) 25pt

Write the two predicates rep(X,L) and rep_all(X,L) in ProLog. X is a positive integer and L is a list of positive integers. rep should generate all possible ways to represent X as a sum of the positive integers 1, 2 and 3. rep_all should generate all possible ways to represent X as a sum of any positive integer.

For instance, we have

```
rep(3,X).
X = [1, 1, 1] ;
X = [1, 2] ;
X = [2, 1] ;
X = [3] ;
```

Solution:

```
%a
rep(0,[]).
rep(X,[1|L]) :- K is X - 1, K>=0, rep(K,L).
rep(X,[2|L]) :- K is X - 2, K>=0, rep(K,L).
rep(X,[3|L]) :- K is X - 3, K>=0, rep(K,L).

%b
rep_all_helper(0,[],_).
rep_all_helper(X,[A|L],D) :- member(A,D), K is X - A, K>=0, rep_all_helper(K,L,D).

smaller(1,[1]).
smaller(X,[X|L]) :- X>=2, K is X - 1, smaller(K,L).

rep_all(X,L) :- smaller(X,XX), rep_all_helper(X,L,XX).
```

Problem 12.47 (Permutations in ProLog) 20pt

1. Construct a predicate $eliminate(X, Y, Z)$ that eliminates the element X from the list Y(the result being list Z). If the element is not in the list, the predicate should yield no solution ($false$).

2. Use the predicate above to define another predicate, $permute(X, Y)$, that computes all the permutations of list X. $permute(X, Y)$ is true if Y is a permutation of X.

Solution:

```
eliminate(X,[X|T],T).
eliminate(X,[H|T],[H|R]):−eliminate(X,T,R).
permute([X|Y],Z) :− permute(Y,W), eliminate(X,Z,W).
permute([],[]).
```

Problem 12.48 (TSP-decision in ProLog) 35pt

Remember the Traveling Salesmen Problem from Genetic Algorithms: given a bidirectional graph, find a minimum-weight tour that includes each node once and returns to the last node. In this task, you will focus on a small part of this problem: given the graph below, define a predicate $tsp(L, C)$ that is true when L is a solution to the TSP with cost less than C (instead of minimum cost, it is enough if the tour has a sufficiently small cost). Your program

- should return a solution S immediately when calling $tsp(S, 8)$.
- must only say yes when S is a valid tour.
- need not say no immediately when there is no solution, e.g.$tsp(S, 7)$.

The graph below has vertices given by the predicate *vertices* and bidirectional edges given by $bi - edge$ that also specify the cost of the edge.

```
edge(a,e,1).
edge(a,g,1).
edge(a,f,3).
edge(b,d,1).
edge(b,e,1).
edge(b,g,2).
edge(c,e,1).
edge(c,f,1).
edge(c,g,2).
edge(d,f,1).

vertices([a,b,c,d,e,f,g]).

bi_edge(X,Y,C) :- edge(X,Y,C).
bi_edge(X,Y,C) :- edge(Y,X,C).
```

Solution:

```
edge(a,e,1).
edge(a,g,1).
edge(a,f,3).
edge(b,d,1).
edge(b,e,1).
edge(b,g,2).
edge(c,e,1).
edge(c,f,1).
edge(c,g,2).
edge(d,f,1).

vertices([a,b,c,d,e,f,g]).

bi_edge(X,Y,C) :- edge(X,Y,C).
bi_edge(X,Y,C) :- edge(Y,X,C).
```

```
eliminate(X,[X|T],T).
eliminate(X,[H|T],[H|R]):−eliminate(X,T,R).
permute([X|Y],Z) :− permute(Y,W), eliminate(X,Z,W).
permute([],[]).

cost([X,Y],C) :− bi_edge(X,Y,C).
cost([X|[Y|Z]],C) :− bi_edge(X,Y,T), cost([Y|Z],R), C is R+T.

last([X],X).
last([X|Y],Z) :− last(Y,Z).

equal(X,X).

tsp([X|Y],C) :− vertices(V), last(Y,T), equal(X,T), cost([X|Y],Z), Z=<C,

permute(Y,V).


\% testcases
?−tsp(S,8).
S = [a, e, b, d, f, c, g, a] ;
S = [a, e, c, f, d, b, g, a] ;
...
```