



SCHEME
HOMEWORK
COLLECTION

GÖKTÜRK ÜÇOLUK



DEPARTMENT OF COMPUTER ENGINEERING



111 HOMEWORK COLLECTION

by

Göktürk Üçoluk

METU

2005

Contents

1	THE HW-1 MACHINE	1
2	NUMBERS TO SCRIPT	9
3	KNIGHT COVERAGE	11
4	PLURALIZE TURKISH	12
5	DNA	13
6	SOLVING ALPHAMETICS	17
7	ROMAN NUMERALS	21
8	NAVIGATION ON TREES	23
9	FIND WEEKDAY	26
10	ETUDE ON LIST PROCESSING	27
11	BRIDGE OPENING BID	29
12	THEOREM PROVING: WANG'S ALGORITHM	31
13	BATTLESHIP GAME	34
14	DIGITAL CIRCUIT SIMULATOR	37
15	SCRIPT TO NUMBER	40
16	D'HONT ELECTION SYSTEM	41
17	HUFFMAN CODING	44
18	SOLVING THE $k^2 - 1$ PUZZLE	47
19	SKY LINER PROBLEM	49

1 THE HW-1 MACHINE

SINCE 1996 ON HOMEWORK 1

"The name of the song is called 'Haddocks' Eyes.'"

"Oh, that's the name of the song, is it?" Alice said trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed.

*"That's what the name is **called**. Then name really is 'The Aged Aged Man.'"*

*"Then I ought to have said 'That's what the **song** is called?'" Alice corrected herself.*

*"No, you oughtn't: that's quite another thing! The **song** is called 'Ways and Means': but that's only what it's **called**, you know!"*

*"Well, what **is** the song, then?" said Alice, who was by this time completely bewildered.*

"I was comming to that," the Knight said. "The song really is 'A-sitting On A Gate': and the tune's my own invention."

–Lewis Carroll, THROUGH THE LOOKING GLASS

Problem

In this homework you will be experimenting with a given process on number sequences. This process resemble the action of a simplified Von Neumann Machine. You will observe that the number sequence (which will change from problem to problem) acts like a machine code program loaded into the memory of a Von Neumann machine. The two registers \mathcal{R}_1 , \mathcal{R}_2 and the instruction pointer \mathcal{I} , the only three internals of the process, carry out the functions of the data, address, condition and the instruction registers of a Von Neumann machine.

Our process (lets name it from now on as the **HW-1 machine**) works on a sequence of integers each of which are in the range $[-127, +127]$. Here is an example for such a sequence:

1	14	2	33	16	8	8	0	0
0	1	2	3	4	5	6	7	8

If we are speaking about the value of the 3rd element in the sequence then we will denote this by enclosing the 3 into square brackets, like $[3]$. In the example above $[3]$ is **33**.

Furthermore, the HW-1 machine has three registers which we name as \mathcal{R}_1 , \mathcal{R}_2 an \mathcal{I} . Each one of \mathcal{R}_1 and \mathcal{R}_2 can hold an integer in the range $[-127, +127]$. Storing negative values into \mathcal{I} is not allowed, it can hold any integer in the range $[0, +255]$ (though in your exercises the values of \mathcal{I} will be about 10-30, at most). The square bracket notation applies for these registers as well. Namely $[\mathcal{R}_1]$ means the content of the \mathcal{R}_1 th element in the sequence. So, for instance, if \mathcal{R}_1 has at any moment the value 3 and if at that moment the 3rd sequence element is **33** (as it is in the example above) then $[\mathcal{R}_1]$ refers that **33**.

The interesting point is that the values in the sequence as well as the values in the registers can be changed freely to new values. When this is the case the former value is erased and the new value is substituted in that place. We will call this action an *assignment* and will represent it by the following notation:

$$place \leftarrow new\ value$$

Here are some assignment examples:

$\mathcal{R}_1 \leftarrow 5$	\mathcal{R}_1 is set to the value 5
$\mathcal{R}_1 \leftarrow \mathcal{R}_2$	\mathcal{R}_1 is set to the same value \mathcal{R}_2 is holding now. (<i>Attention: This does not mean that any following changes on \mathcal{R}_2 will effect the value stored in \mathcal{R}_1</i>)
$\mathcal{R}_2 \leftarrow [2]$	\mathcal{R}_2 is set to the 2nd value in the sequence.
$[0] \leftarrow \mathcal{R}_1$	The 0th (zeroth) value in the sequence is changed to be the same value that is in \mathcal{R}_1 .
$\mathcal{I} \leftarrow \mathcal{I} + 1$	The content of \mathcal{I} is incremented by one.

The number of changes is not limited and can be performed as many times as desired on any register or sequence element.

Given a sequence, HW-1 starts with the \mathcal{I} register having 0 (zero) value and the other two registers having arbitrary values. It works by repeatedly going through a process cycle until a halt instruction is executed. A process cycle is:

1. Take $[\mathcal{I}]$ as an instruction.
2. If this instruction is the halt instruction then terminate the process,
3. else perform the action associated with that instruction.
4. Continue with step (1).

If any instruction described in the following two pages computes a result (at any cycle) that falls out of the limits then the machine automatically halts.

The HW-1 accepts 17 instructions which are explained below. Instructions are recognized as integers $[0,1,\dots,16]$.

Instruction 0

Halt the process..

Instruction 1

Load \mathcal{R}_1 with the next number in the sequence.

$$\mathcal{R}_1 \leftarrow [I + 1], \quad I \leftarrow I + 2$$

Instruction 2

Load \mathcal{R}_2 with the next number in the sequence.

$$\mathcal{R}_2 \leftarrow [I + 1], \quad I \leftarrow I + 2$$

Instruction 3

Load \mathcal{R}_1 with the sequence element which is at the position given as the next number in the sequence.

$$\mathcal{R}_1 \leftarrow [[I + 1]], \quad I \leftarrow I + 2$$

Instruction 4

Load \mathcal{R}_2 with the sequence element which is at the position given as the next number in the sequence.

$$\mathcal{R}_2 \leftarrow [[I + 1]], \quad I \leftarrow I + 2$$

Instruction 5

Load \mathcal{R}_1 with the content of \mathcal{R}_2 .

$$\mathcal{R}_1 \leftarrow \mathcal{R}_2, \quad I \leftarrow I + 1$$

Instruction 6

Load \mathcal{R}_1 with the sequence element which is at the position \mathcal{R}_2 .

$$\mathcal{R}_1 \leftarrow [\mathcal{R}_2], \quad I \leftarrow I + 1$$

Instruction 7

Change the sequence element which is at the position \mathcal{R}_1 to be the content of \mathcal{R}_2 .

$$[\mathcal{R}_1] \leftarrow \mathcal{R}_2, \quad I \leftarrow I + 1$$

Instruction 8

Change the sequence element which is at the position given as the next number in the sequence to the content of \mathcal{R}_1 .

$$[[I + 1]] \leftarrow \mathcal{R}_1, \quad I \leftarrow I + 2$$

Instruction 9

Take the sequence element which is at the position given as the next number in the sequence as the next instruction to be performed.

$$I \leftarrow [I + 1]$$

Instruction 10

If \mathcal{R}_1 contains zero continue with the sequence element following the next one as the next instruction to be performed, otherwise act like the instruction 9.

$$\begin{array}{ll} \text{if } \mathcal{R}_1 = 0 & : I \leftarrow I + 2 \\ \text{otherwise} & : I \leftarrow [I + 1] \end{array}$$

Instruction 11

Increment \mathcal{R}_1 by the content of \mathcal{R}_2 .

$$\mathcal{R}_1 \leftarrow \mathcal{R}_1 + \mathcal{R}_2, \quad I \leftarrow I + 1$$

Instruction 12

Decrement \mathcal{R}_1 by the content of \mathcal{R}_2 .

$$\mathcal{R}_1 \leftarrow \mathcal{R}_1 - \mathcal{R}_2, \quad I \leftarrow I + 1$$

Instruction 13

Multiply \mathcal{R}_1 by the content of \mathcal{R}_2 .

$$\mathcal{R}_1 \leftarrow \mathcal{R}_1 \times \mathcal{R}_2, \quad I \leftarrow I + 1$$

Instruction 14

Divide \mathcal{R}_1 by the content of \mathcal{R}_2 (integer division).

$$\mathcal{R}_1 \leftarrow \mathcal{R}_1 \div \mathcal{R}_2, \quad I \leftarrow I + 1$$

Instruction 15

Change the sign of the value in \mathcal{R}_1 .

$$\mathcal{R}_1 \leftarrow -\mathcal{R}_1, \quad I \leftarrow I + 1$$

Instruction 16

Compare the content of \mathcal{R}_1 with the content of \mathcal{R}_2 .

$$\begin{array}{ll} \text{if } \mathcal{R}_1 = \mathcal{R}_2 & : \mathcal{R}_1 \leftarrow 0 \\ \text{if } \mathcal{R}_1 > \mathcal{R}_2 & : \mathcal{R}_1 \leftarrow 1 \\ \text{otherwise} & : \mathcal{R}_1 \leftarrow -1 \\ \text{always} & I \leftarrow I + 1 \end{array}$$

An Example

If HW-1 is submitted the below given sequence, it will compute the sum of its 2nd and 3rd elements, then compare this sum with the 4th element. If the sum equals the 4th element then the 5th element of the sequence is changed to **1** else it is changed to **0**.

9	6	12	27	39	99	3	2	4	3	11	4	4	16	10	20	2	1	9	22	2	0	1	5	7	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

We will go through each cycle of the process and explain it:

Cycle 1

The machine starts now. So the registers are as follows:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
?	?	0

Since \mathcal{I} contains 0, the 0th (zeroth) element of the sequence, $[\mathcal{I}]$ is fetched. This will be our current instruction. Looking at the sequence you may realize that this element is **9**. Look at the definition of the instruction 9 (given on the previous page), you will see that the action of the instruction 9 is:

$$\mathcal{I} \leftarrow [\mathcal{I} + 1]$$

$\mathcal{I} + 1$ is 1 and therefore $[\mathcal{I} + 1]$ is simply $[1]$, in other words the 1st element of the sequence. That is **6**. So the value in \mathcal{I} is changed by the assignment operation to be 6. And now the registers are:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
?	?	6

Cycle 2

\mathcal{I} contains 6. The machine fetches $[6]$ which is **3**. Instruction 3 is defined as:

$$\mathcal{R}_1 \leftarrow [[\mathcal{I} + 1]] , \quad \mathcal{I} \leftarrow \mathcal{I} + 2$$

The first assignment will change the content of \mathcal{R}_1 to be $[[\mathcal{I} + 1]]$. Now do not panic! \mathcal{I} was holding a value of 6 therefore the inner brackets are nothing else but

$$[[\underbrace{\mathcal{I} + 1}_{[6+1]}]] \quad \text{which is} \quad [[7]]$$

$[7]$ is the 7th element of the sequence, namely **2**. So $[[7]]$ is nothing but $[2]$. As you must have got used to it by now, this is the 2nd element of the sequence, we go and fetch it. It is **12**. Since the first assignment was $\mathcal{R}_1 \leftarrow [[\mathcal{I} + 1]]$ this value found on the right hand side of the assignment, namely the 12, will be stored into \mathcal{R}_1 . The second assignment increments the value of \mathcal{I} by 2. Thus \mathcal{I} is now $6 + 2$ which is 8. At the end of this cycle the register contents are:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
12	?	8

Cycle 3

Since \mathcal{I} is containing 8 we fetch [8] as the instruction of this cycle. The 8th element of the sequence is **4**. The instruction 4 is defined similar to the instruction 3 but now it is the \mathcal{R}_2 register that receives the value and not \mathcal{R}_1 . Going through a very similar argumentation that we did for cycle 2, we conclude that \mathcal{R}_2 will be assigned the value [3] which is **27**. Following this assignment, \mathcal{I} will be incremented by 2. That is how the registers look like:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
12	27	10

Cycle 4

This cycle's instruction is [10]. Looking at the sequence we see that this is the instruction **11**. An instruction which adds the content of \mathcal{R}_2 to the content of \mathcal{R}_1 and assigns the sum to \mathcal{R}_1 (*Please go and check the definition of instruction 11*). So the following operation is performed:

$$\mathcal{R}_1 \leftarrow 12 + 27$$

So, \mathcal{R}_1 is assigned a value of 39. Due to the definition of instruction 11, the \mathcal{I} register is incremented by 1. At the end of this cycle the registers read as:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
39	27	11

Cycle 5

Our instruction is [11] which is **4**. We have had a similar instruction in cycle 3. But this time [$\mathcal{I} + 1$] is **4** which means that \mathcal{R}_2 is assigned the value [4]. From the sequence we get that this value is 39. So $\mathcal{R}_2 \leftarrow 39$. By the definition, \mathcal{I} is incremented by 2. Now the registers are:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
39	39	13

Cycle 6

The instruction of this cycle is [13], namely **16**. This is a comparison test performed between \mathcal{R}_1 and \mathcal{R}_2 . If they are equal \mathcal{R}_1 is changed to 0, if the value of \mathcal{R}_1 is greater than the value of \mathcal{R}_2 then \mathcal{R}_1 is changed to 1. If it is the third possibility, which is the only one left, namely the case where the value of \mathcal{R}_1 is lesser than the value of \mathcal{R}_2 then \mathcal{R}_1 is changed to -1 . In all cases \mathcal{I} is incremented by one. Following this definition and looking at the contents of the registers we can conclude that HW-1 will change \mathcal{R}_1 to 0 since both \mathcal{R}_1 and \mathcal{R}_2 have the value 39, hence they are equal. The registers are:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
0	39	14

Cycle 7

The instruction is [14] which reads as **10**. This is a conditional jump ('jump on non-zero'). If the register \mathcal{R}_1 contains a non zero value then the instruction of the next cycle will be fetched from the sequence position that given at $[\mathcal{I}+1]$. In our case the register \mathcal{R}_1 indeed contains a 0. So the jump will not take place and the \mathcal{I} register will simply be incremented by 2. (If the jump would have taken place then \mathcal{I} would be set to **20** and this would be the position in the sequence from which the instruction for cycle 8 would be fetched) So the registers are

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
0	39	16

Cycle 8

The instruction is now [16], that is **2**. This will load \mathcal{R}_2 with $[\mathcal{I} + 1]$. In our case this is $\mathcal{R}_2 \leftarrow [17]$ and will result in $\mathcal{R}_2 \leftarrow 1$. For an instruction 2, \mathcal{I} will be incremented by 2. At the end of this cycle we have in the registers

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
0	1	18

Cycle 9

The instruction is [18] which is **9** an instruction that performs an unconditional jump. \mathcal{I} will be set to $[\mathcal{I} + 1]$ and that is **22**. The registers are

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
0	1	22

Cycle 10

The instruction is [22] which is **1**. This instruction will load \mathcal{R}_1 with $[\mathcal{I} + 1]$. For our case this is **5**. Instruction 1 increments \mathcal{I} by 2. Now we have the registers as

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
5	1	24

Cycle 11

We are almost at the end! The instruction in turn is [24], that is **7**. An instruction that performs

$$[\mathcal{R}_1] \leftarrow \mathcal{R}_2$$

and then increment \mathcal{I} by one. Due to this definition a $[5] \leftarrow 1$ is carried out. And for the first time we have changed a value in the sequence. Now the 5th element is no more **99** but 1. This was the claimed action of the HW-1 machine with this example sequence. After a following incrementation of \mathcal{I} the registers are:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
5	1	25

Cycle 12

Yes, unbelievable but true, the machine stops and we all go home! The instruction for this cycle is [25] which is simply **0**. And this means halt.

the final picture of the sequence is:

9	6	12	27	39	1	3	2	4	3	11	4	4	16	10	20	2	1	9	22	2	0	1	5	7	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

For sake of completeness we display the registers at the moment of the halt:

\mathcal{R}_1	\mathcal{R}_2	\mathcal{I}
5	1	25

2 NUMBERS TO SCRIPT

1996 HOMEWORK 2

Problem

In this homework you will convert a single number N which is

$$0 \leq N \leq 9999$$

into its written expression in a human language. Except Barış Güldalı, who has do it in german (this is not a joke! we mean it), all the other members of the class will choose among english or turkish for the language.

Specification

- You will name your function that does the conversion as `convert`. `convert` will take exactly one argument which is the integer to be converted.
- If the input is not an integer then the function shall return the string
`"The argument is not an integer"`
- If the input is not an integer in the specified range then the function shall return the string
`"In school we have not seen yet how to express this!"`
- You are allowed to write other functions that you will use in the definition of `convert`. You are free in naming them.
- You shall try to write short and concise functions. This will influence the grade you will get.

Example

Here is a list of possible inputs and the expected corresponding results for english and turkish **Attention:** You will choose only one language to implement.

Input	Output (english)
(convert 1606)	onethousandsixhundredandsix
(convert 111)	onehundredandeleven
(convert 6199)	sixthousandonehundredandninetynine
(convert 20)	twenty
(convert 0)	zero
(convert 1000)	Onethousand
(convert 1001)	Onethousandandone
(convert 'eltonjohn)	"The argument is not an integer"
(convert 10000)	"In school we have not seen yet how to express this!"

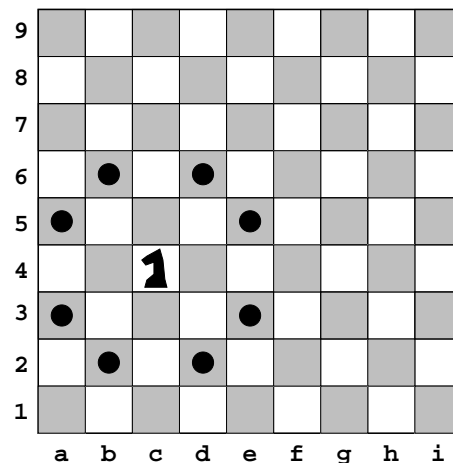
Input	Output (turkish)
(convert 1606)	binaltiyuzalti
(convert 111)	kirkbir
(convert 6199)	altibinyuzdoksandokuz
(convert 20)	yirmi
(convert 0)	sifir
(convert 1000)	bin
(convert 1001)	binbir
(convert 'eltonjohn)	"The argument is not an integer"
(convert 10000)	"In school we have not seen yet how to express this!"

3 KNIGHT COVERAGE

1996 HOMEWORK 3

Problem

Cover a chess board by knights (“at” in Turkish chess terminology). In a covered chess board each square is occupied by a knight or under attack by some knight (or both). Below you see for an arbitrary placed knight the squares under attack (*Marked with a black spot*).



It is not important whether some knights attack each other. Clearly it is desirable to achieve a covering by using a minimum number of knights. It is known that a standard 8×8 chess board can be covered by 12 knights.

Specifications

You are asked to write a Scheme procedure, called `cover`, that covers an $n \times n$ chessboard with k knights. More precisely,

```
(cover n k)
```

should yield a sentence with k words each showing the position of a knight on the board. If a coverage is not possible the yielded sentence should be empty.

A word will be of the form cr where c is a letter a through i denoting a column and r is a digit 1 through 9 denoting a row where a1 denotes the southwest corner. This is a standard notation for positions on a chess board.

Notice we have assumed that

$$1 \leq n \leq 9$$

to simplify data representation.

Hint

Study the 8-queens problem carefully. It is a classic. Reading *Chapter 16* of the text, with particular attention to the section on “backtracking”, will also help.

4 PLURALIZE TURKISH

1997 HOMEWORK 2

Problem

You are asked to develop a pluralizer for Turkish. More specifically, your pluralizer should take a word, presumably in Turkish, and produce its plural form. You are expected to write a Scheme function that takes a Turkish word in the form described below and return its pluralized form.

Call your pluralizing procedure `plural`.

The argument word will be lowercase except for `I`, `O`, `U`, `G`, `C` and `S`, which represent the Turkish characters `ı`, `ö`, `ü`, `ğ`, `ç` and `ş`, respectively.

Notice that to preserve the lowercase/uppercase distinction we need to represent the words as strings rather than symbols.

Here are some examples:

```
(plural "kuGu")  
kuGular
```

```
(plural "dert")  
dertler
```

```
(plural "nOron")  
nOronlar
```

```
(plural "kiler")  
kilerler
```

```
(plural "zmcuk")  
zmcuklar
```

```
(plural "zmck")  
zmck?
```

5 DNA

1997 HOMEWORK 3

Background

The life form on our planet is primarily based on a molecule that we call DNA. It carries within its structure the heredity information that determines the structure of proteins and the instructions that direct cells to grow and divide. So are the messages that bring about the differentiation of fertilized eggs into the multitude of specialized cells that are necessary for the successful functioning of higher plants and animals.

In DNA molecules, *nucleotides* are linked together to form long chains by bonds. The order and sequence of this chain is the information content of the DNA. Each DNA molecule contains many subparts that we call *gene*. A *gene* is the smallest functional unit and serves as a 'program' that synthesizes a protein (a chain of amino-acids).

On the left you see a part of such a synthesised protein: The A-chain of an *insulin*.

Just for your information the names of the building blocks of proteins namely the Amino acids are:

GLY
ILE
VAL
GLU
GLN
CYS
CYS
ALA
SER
VAL
CYS
SER
LEU
TYR
GLN
LEU
GLU
ASN
TYR
CYS
ASN

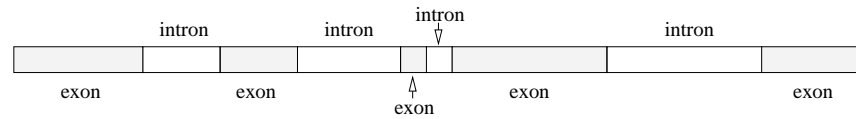
Glycine	GLY	Lysine	LYS
Alanine	ALA	Arginine	ARG
Valine	VAL	Asparagine	ASN
Isoleucine	ILE	Glutamine	GLN
Leucine	LEU	Cysteine	CYS
Serine	SER	Methionine	MET
Threonine	THR	Tryptophan	TRP
Proline	PRO	Phenilalanine	PHE
Asparic acid	ASP	Tyrosine	TYR
Glutamic acid	GLU	Histidine	HIS

A *nucleotide* is one of the the four molecules:

A : Adenine
C : Cytosine
G : Guanine
T : Thymine

In a gene, each group of three successive nucleotides is called a *codon*. Each codon is responsible of synthesizing an amino acid (one of the 20 amino acids given in the table above).

Although a gene is responsible of synthesis of a protein, it has parts which are 'garbage' and parts which are 'functional'. These parts are called *introns* and *exons*, respectively. So, it is actually the exon zones which produces the protein and introns can practically be omitted from the sequence.



When it is time to produce a protein, a kind of a mask is generated from the gene. This mask is called the messenger RNA (mRNA) which also consists of only four kind of nucleotides. These are

A : Adenine
 C : Cytosine
 G : Guanine
 T : Urasil

It is only the useful parts that are masked, namely the exons. The introns are simply ignored. This mask generation is called *transcription*. Each nucleotide in the gene is represented by a one-to-one correspondence. Here is the table of transcription:

ORIGINAL DNA NUCLEOTIDE	TRANSCRIBED INTO mRNA NUCLEOTIDE
A	U
C	G
G	C
T	A

After the transcription is completed, the codons of the mRNA are used to manufacture the protein. Since there are 4 nucleotides, as you might have realized already, there are $4 \times 4 \times 4$ possibilities for them to group in four. That means there are 64 codons. But as we know they synthesize only 20 amino acids and some codons stop the process. So, evidently, some codons must synthesize the same amino-acid. That is true, and is called *degeneracy*.

Here is the table of which mRNA codon is synthesizing which amino acid.

FIRST POSITION	SECOND POSITION				THIRD POSITION
	U	C	A	G	
U	PHE	SER	TYR	CYS	U
	PHE	SER	TYR	CYS	C
	LEU	SER	stop	stop	A
	LEU	SER	stop	TRP	G
C	LEU	PRO	HIS	ARG	U
	LEU	PRO	HIS	ARG	C
	LEU	PRO	GLN	ARG	A
	LEU	PRO	GLN	ARG	G
A	ILE	THR	ASN	SER	U
	ILE	THR	ASN	SER	C
	ILE	THR	LYS	ARG	A
	MET	THR	LYS	ARG	G
G	VAL	ALA	ASP	GLY	U
	VAL	ALA	ASP	GLY	C
	VAL	ALA	GLU	GLY	A
	VAL	ALA	GLU	GLY	G

Problem

You will be given a list of nucleotides which represent a gene. You know also that there is exactly 3 exons and 2 introns. The gene starts with an exon and ends with an exon. You don't know where the intermediate exon is located. You also know that the both of the introns are non-empty. Since exons are sequences of codons (three nucleotides together) they are a multiple of 3. This is not so for introns, since they are garbage their length has not to be a multiple of 3. Here is an example of such a gene:

```
(T C T G C A G C A G A G G G G C C G T C G G C A G A A G G A G
G G C T C G G G C A G G C T C T G C G A C T C G T A G G C A C
C A G G C G T G A G A C C T G T A G C C C C C G A T C A C C A
T G T A C A G C T T C A T G G G T G G T G G C C T G T T C T G
T G C C T G G G T G G G G A C C A T C C T C C T G G T G G T G
G C C A T G G C A A C A G A C G G G G C C A A G G A C A C C T
G T A T T C C A G A T G G A G A A C T C T G C G G C T C A A A
G A G G G A A A G G G A G C A A C C C A A G G T C A C T C A G
C G G A G G C T G A C T C C T G G T C C T A G G C T G G A A G
G A G G A A G A A T A G G G C C C A T G G G A G G G A G C T G
A G A A G A C T)
```

You will also be given a protein (as a list of amino acids). Which is something like:

```
(ARG ARG ARG LEU PRO GLY SER ARG LEU PRO PRO GLU PRO VAL ARG ASP ALA GLU
LEU VAL VAL HIS VAL GLU VAL PRO THR THR GLY GLN ASP THR ASP PRO PRO LEU
VAL GLY GLY PRO PRO PRO VAL PRO LEU SER PRO PRO THR GLU ASP GLN ASP PRO
THR PHE LEU LEU LEU ILE PRO GLY THR LEU PRO ARG LEU PHE stop)
```

You are expected to find the introns' start and end positions. The answer will be given a list of two lists. Where each inner list is a pair of two positions.

$((intron_{1start} \quad intron_{1end}) \quad (intron_{2start} \quad intron_{2end}))$

If you discover that this protein is not the product of the gene given, then you shall return an empty list.

Name your function that takes the gene list as its first argument and the protein list as the second, as `locate_intron`

The answer for the above example shall be:

$((55 \ 85) \ (170 \ 249))$

6 SOLVING ALPHAMETICS

1997 HOMEWORK 4

...
*Ben, öteki, bir diğeri ona doğru ilerler
 ilerlerim
 zamanla anlarsın bu bir yanılsama
 ölü şairlerin imgelerinden kalma
 Sen de değilsin. O da değil
 Kuzey yıldızı daha uzakta
 yeniden yollara düşerler
 düşerim*
 —Murathan Mungan

Introduction

The Alphabetic Problem

In an *alphabetic* problem the task is to find a one-to-one mapping from a set of letters (or symbols) to a subset of digits such that a given arithmetic operation in which each digit is represented by the corresponding letter or symbol is mathematically correct. Since it is among the main interest subjects of recreational mathematics, you can find many example alphametics in journals like *Journal of Recreational Mathematics*. Here is a famous example:

$$\begin{array}{r}
 \text{S E N D} \\
 + \quad \text{M O R E} \\
 \hline
 \text{M O N E Y}
 \end{array}$$

The set of letters used is $\{\text{D, E, M, N, O, R, S, Y}\}$. We will call this set the *alphabet* of the problem. Without making use of some mathematical intelligence (like the deduction of $M = 1$ due to its appearance in the left most position of the sum), generally speaking, one can say that each column in the sum is an equation that holds modulo 10 with a possibility of having received a *carry* from the previous column (except the right most column). So, if there exist M such columns, and N different values a carry can take, there are N^{M-1} possibilities for forming a system of M equations.

Furthermore the restriction on the solution domain inhibits the use of standard equation solution techniques. A brute force computer solution which goes over all possibilities roughly requires a time proportional to $10!/(10 - \alpha)!$ where α is the *cardinality* (the count of elements) of the problem alphabet. For the above given example this means 1814400 trials for the worst case. In many cases this figure can be reduced drastically by building in some short cuts.

Backtracking

Backtracking is a technique used in searching for a solution. Assume there are M number of stages at which you have to make a decision (or choice) to arrive at a result. The problem is that the choice(s) you make may not lead to a satisfactory result (which we will call the *solution*). Now how to proceed? In other words how can we perform a systematic search among the possible choices? Let us assume that for stage i you have a set of choices S_i available. The backtracking method proceeds as follows:

1. Start with $i = 1$. Consider the first member (the first choice) of the S_1 (the set of all possible choices you could make for stage 1).
2. Consider the next i value. That is $i = 2$.
3. Consider the first possible choice of S_2 .
4. Now check the consistency of the choices you made for stage 1 and for stage 2. Is there any conflict among the choices made so far? That means: Can you conclude that the choice you made for stage 1 contradicts with the choice you made for stage 2?
5. If your answer is **yes** (there is a contradiction) then this setting of choices has no chance to lead to a solution. Therefore we try an alternative for the most recently made choice. The most recent choice we made was for stage 2, so we pick another choice for it: the next possible choice from the set S_2 ; if such an alternative exists we pick it and continue with step (4). But if no untested choice of S_2 exists anymore then you have to go back to the choice you made for stage 1 and change it to another choice (the next untried one from the set S_1) then continue from step (2).
6. If your answer is **no** (there is no contradiction) then you may well advance to the next stage where you will make a choice for stage $i + 1$ (namely stage 3).
7. You carry out this procedure until you have to hand a choice for the very last stage (stage M) which does not contradict with all the choices made at previous stages, Now, none of the choices are contradictory and that setting of choices is a solution.

More formally [with the assumption of $next_of(NONE, S) = first_of(S)$]:

```

1          $i \leftarrow 1$ 
2          $choice_i \leftarrow |NONE|$ 
3         if  $\neg exists\_next\_of(choice_i, S_i)$ 
4             then
5                  $decrement(i)$ 
6                 if  $i < 1$  then return ( $|NOSOLUTION|$ )
7                     else continue from step 3
8         else
9              $choice_i \leftarrow next\_of(choice_i, S_i)$ 
10            if  $contradict(next\_of(choice_i, S_i), \{choices_k \mid k = 1 \dots i - 1\})$ 
11                then
12                    continue from step 3
13            else
14                 $increment(i)$ 
15                if  $i > M$  then return ( $|SOLUTION : \{choices_i \mid i = 1 \dots M\}$ )
16                else continue from step 2

```

Study this algorithm. You shall understand the idea behind it and not attempt an line by line implementation. A Scheme implementation is more simple and does not require the index i .

The solution provided to the following problem is a good example of the backtracking-technique:

How is it possible to place N queens on a $N \times N$ chess board so that they do not threaten each other?

First we have to write a predicate that will determine whether two queens threaten each other. A queen in chess can move along the column, the row, and the two diagonals through her present position. We can encode the positions on the board by numbering the rows and columns. Then the following function will do:

```
(define (threat i j a b)
  (or (= i a) ; Same row
      (= j b) ; Same column
      (= (- i j) (- a b)) ; SW-NE diagonal
      (= (+ i j) (+ a b)))) ; NW-SE diagonal
```

Next, we can represent the configuration of several queens on the board using a list of two-element sublists, each sublist containing the row and column number of one queen on the board. Suppose now that we plan to add a queen to a board. The following function will tell us whether the position (N,M) is safe:

```
(define (conflict n m board)
  (cond ((null? board) #f)
        (#t (or (threat n m (caar board) (cadar board))
                 (conflict n m (cdr board))))))
```

With the preliminaries out of the way we can tackle the search problem. We write `queen` such that starting at row 0, an attempt is made to place a queen in column 0. After placing the first queen, it does not make sense to place another queen in the same row, so attention shifts to the next row. A safe square is found there. This process is to continue until all queens are placed on the board. If, at some stage, all squares on a particular row are threatened, the program has to back up. It is to do this by removing the last queen placed on board.

```
(define (queen size) (queen-aux () 0 size))

(define (queen-aux board n size)
  (cond ((= n size) (write (reverse board)))
        (#t (queen-sub board n 0 size))))

(define (queen-sub board n m size) ; Try next column
  (cond ((= m size) ()) ; Hit end of row?
        (#t (cond ((conflict n m board)) ; Conflict
                   (#t (queen-aux (cons (list n m) board) (+ n 1) size))) ; No
                  (queen-sub board n (+ m 1) size)))) ; Now move right one
```

Problem

You will write a function `solve-alphabetic` which will take a list of 3 words ($w_1 w_2 w_3$) as argument. This way you introduce the alphabetic problem:

$$w_1 + w_2 = w_3$$

It is a rule that none of the words w_k can start with a letter that means the digit 0 (zero). The result of the function call is expected to be an association list of the form:

$$((letter_1 digit_1) (letter_2 digit_2) \dots (letter_n digit_n))$$

which is the solution key. Here $\{letter_i \mid i = 1 \dots n\}$ is the set of all distinct letters which appear in at least one of the 3 words. The association list is expected to be accendingly ordered: $i < j \iff letter_i < letter_j$.

Furthermore, $digit_i \in [0, 9]$ and $digit_i = digit_j \iff i = j$ (verbally: all digits are distinct). If all $letter_i \leftarrow digit_i$ substitutions are made according to the key, all the words will turn into numbers. A correct solution will be recognized as producing:

$$number_1 + number_2 = number_3$$

Here is the function call and the expected result for the introductory example:

```
(solve-alphabetic '(SEND MORE MONEY))
((D 7) (E 5) (M 1) (N 6) (O 0) (R 8) (S 9) (Y 2))
```

7 ROMAN NUMERALS

1998 HOMEWORK 2

Overview

Roman numerals were developed around 500 BC at least partially from primitive Greek alphabet symbols which were not incorporated into Latin. Using predominantly addition, they are read from left to right.

The symbol 'I' for 1 was derived from one finger. Five fingers held up indicated five of whatever was being counted. The 'V' then was the hand outstretched vertically with the space between the thumb and first finger forming the 'V'.

Originally the Greek letter 'χ', or *chi*, meant 50, but in monument transcriptions it is easy to trace the original symbol's change to 'L', and 'X' came to mean 10. Another theory for 'X' is that ten 1's were written in a row, and then crossed out with an 'X' to simplify counting. Then the 'X' alone became a shorthand version of 10. Yet another idea is that 'V' looks like the top half of 'X', as 5 is half of 10. Other scholars think that 'V' doubled with an upside-down 'V' meant 5 times 2, or 'X'. 'C', indicating 100, came from the Latin word *centum*, a hundred. (Also century, centennial, etc.) 'M' is from "mille", a thousand. Larger numbers, like 5,000, are shown by putting a small bar called a *vinculum* above the 'V' symbol, indicating multiplication by 1,000.

Until fairly recently a commonly used Roman numeral for 1,000 was 'Φ'. Half of this symbol, led to 'D' for 500, half of 1000.

Generally, decoding Roman numerals is very straightforward. The largest numeral is at the left, with descending numerals moving to the right. Numbers are added as you go, as seen in these examples:

$$\text{CCLXVII} : 100 + 100 + 50 + 10 + 5 + 1 + 1 = 267$$

$$\text{MMMCCCLXXXI} : 1000 + 1000 + 1000 + 100 + 100 + 50 + 10 + 10 + 10 + 1 = 3281$$

$$\text{DCCXVII} : 500 + 100 + 100 + 10 + 5 + 1 + 1 = 717$$

Rather than continuing to add 1's to make 4 **IIII** or 9 **VIIII**, subtraction was included in the computation of the numerals to simplify and shorten the resulting numbers. Therefore, 4 is shown **IV**, or 5 minus 1. The smaller numeral BEFORE the larger one means subtract. **IX** is 9, or 10 minus 1. 40 is **XL**, 50 minus 10; 90 is **XC**, 100 minus 10; **CD** is 400, or 500 minus 100; and **CM** is 900, or 1000 minus 100. Students can follow the principle that subtraction takes place ONLY when the smaller numeral is before the larger one, and involves 4 and 9 in various place values. There are two rules about these smaller numerals which are placed to the left of a bigger one and subtracted.

- Only **I**, **X**, and **C** can be used in this way; **V**, **L**, and **D** cannot and of course **M** cannot because it is the biggest numeral anyway.
- The subtracted number must be as close as possible in size to the number it is subtracted from. So an **X** can be placed to the left of a **C** or an **L** but not to the left of an **M** or a **D**.

These two rules limit the usefulness of the subtraction rule in reducing the length of Roman numerals. So 99 is not the very short **IC** (100 minus 1) but is **XCIX**. And although the year 2000 is quite neat at **MM**, 1999 is something else. Try it!

Obviously, the cumbersome aspect of Roman numerals is one of the main reasons that they have been replaced by the Arabic system in our daily mathematical lives. Roman numerals remain important as a part of the world's cultural past, and a unique way to express numbers.

Problem

Write two functions, `roman_to_arabic` that converts a roman number expressed as a *word* into a arabic number and , `arabic_to_roman` which does the inverse. In all cases the value expressed will be positive and < 4000 .

Examples:

```
(roman_to_arabic 'MCDXXXIX)  
1439
```

```
(arabic_to_roman 928)  
CMXXVIII
```

8 NAVIGATION ON TREES

1998 HOMEWORK 3

Background

You will have three tasks to be solved in Scheme. They are about trees. The grades are listed along with the tasks. You are allowed to define slave functions. (*But do not exaggerate*)

We start with a review of the basics about trees: A *tree* is a nonempty collection of *vertices* and *edges* that satisfies certain requirements. A vertex is a simple object (also referred to as a *node*) that can have a name and can carry other associated information; an edge is a connection between two vertices. A path in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree. One node in the tree is designated as the *root* — the defining property of a tree is that there is exactly one path between the root and each of the other nodes in the tree. If there is more than one path between the root and some node, or if there is no path between the root and some node, then what we have is a *graph*, not a tree.

Though the definition implies no “direction” on the edges, we normally think of the edges as all pointing away from the root or towards the root depending upon the application. We usually draw trees with the root at the top (even though this seems unnatural at first), and we speak of node *y* as being *below* node *x* (and *x* as *above* *y*) if *x* is on the path from *y* to the root (that is, if *y* is below *x* as drawn on the page and is connected to *x* by a path that does not pass through the root). Each node (except the root) has exactly one node above it which is called its *parent*; the nodes directly below a node are called its *children*. We sometimes carry the analogy to family trees further and refer to the “grandparent” or the “sibling” of a node. Nodes with no children are sometimes called *leaves*, or *terminal* nodes. To correspond to the latter usage, nodes with at least one child are sometimes called *nonterminal* nodes. Terminal nodes are often different from nonterminal nodes: for example, they may have no name or associated information. Especially in such situations, we refer to nonterminal nodes as *internal* nodes and terminal nodes as *external* nodes. Any node is the root of a *subtree* consisting of it and the nodes below it.

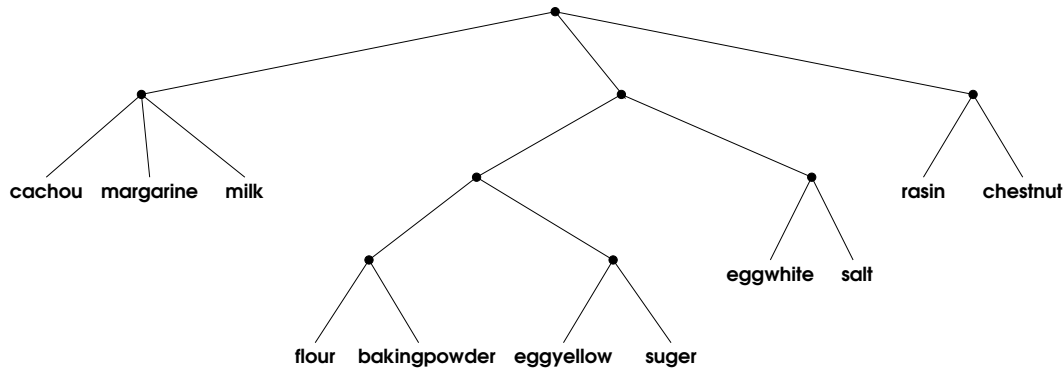
Sometimes the way in which the children of each node are ordered is significant, sometimes it is not. An *ordered* tree is one in which the order of the children at every node is specified. Of course, the children are placed in some order when we draw a tree, and clearly there are many different ways to draw trees that are not ordered. As we will see below, this becomes significant when we consider representing trees in a computer, since there is much less flexibility in how to represent ordered trees. It is normally obvious from the application which type of tree is called for.

The nodes in a tree divide themselves into *levels*: the level of a node is the number of nodes on the path from it to the root (not including itself). The *height* of a tree is the maximum level among all nodes in the tree (or the maximum distance to the root from any node). The *path length* of a tree is the sum of the levels of all the nodes in the tree (or the sum of the lengths of the paths from each node to the root). The tree in Figure 4.1 is of height 3 and path length 21. If internal nodes are distinguished from external nodes, we speak of *internal path length* and *external path length*.

Trees are intimately connected with recursion. In fact, perhaps the simplest way to define trees is recursively, as follows:

“a tree is either a single node or a root node connected to a set of trees”.

You will be given a tree which has information stored only in its leaves. The information is of *atom* type. The data representation of this tree is a list type *sexpr*. In this expression the information in the leaves are represented as themselves; children of the same parent are grouped into a pair of parenthesis. Here follows an example, consider a subject tree:



the *sexpr* representation that corresponds to the tree above is

```
((cachou margarine milk) (((flour bakingpowder) (eggyellow suger)) (eggwhite sa
(rasin chestnut)))
```

(In the description of the three tasks below, we will refer to this example, but by no means this shall mean that you will write functions that work only for the given example! Of course your solutions will be tested with other trees.)

TASK 1 (20 points)

Write a predicate function `has_leave?` which takes a tree representation as described above as the first argument and a list of atoms as the second argument. `has_leave?` shall return `#t` if at least one of the members of the second argument list is a leaf of the tree given in the first argument. If that is not the case the returned value is `#f`.

So

```
(has_leave? '((cachou margarine milk)
              (((flour bakingpowder) (eggyellow suger)) (eggwhite salt))
              (rasin chestnut))
            '(meat suger chips flour))
```

will return `#t` and

```
(has_leave? '((cachou margarine milk)
              (((flour bakingpowder) (eggyellow suger)) (eggwhite salt))
              (rasin chestnut))
            '(meat chips))
```

will return `#f`.

TASK 2 (30 points)

Write a function `in_tree_path` which takes a tree representation as described above as the first argument and a single leaf as the second. `in_tree_path` shall return a list of integers which describes the path which has to be followed starting from the root and ending at the given leaf. At every node the left-most edge is numbered as 1, the next to its right is numbered as 2, and so on.

Here is an example:

```
(in_tree_path '((cachou margarine milk)
               (((flour bakingpowder) (eggyellow sugar)) (eggwhite salt))
               (rasin chestnut))
              'eggyellow))
```

will return (2 1 2 1). A change in the second argument to 'rasin would yield a return value of (3 1).

The second argument is definitely one of the leaves. No test case will breach this.

TASK 3 (50 points)

Write a function `breadth_first` which takes a single argument, namely a tree representation as described above, and returns a list of leaves in the order of levels and from left to right. For the example tree given above:

```
(breadth_first '((cachou margarine milk)
                (((flour bakingpowder) (eggyellow sugar)) (eggwhite salt))
                (rasin chestnut)))
```

will return

```
(cachou margarine milk rasin chestnut eggwhite salt flour bakingpowder eggyellow
```

9 FIND WEEKDAY

1999 HOMEWORK 2

Problem

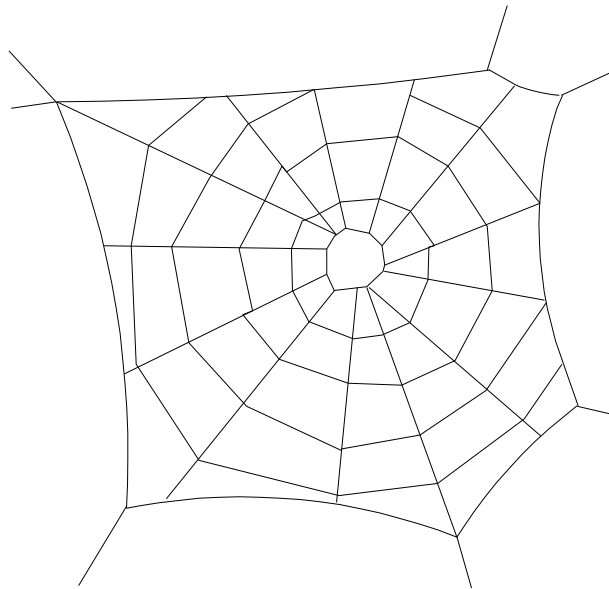
Write a function `date_to_weekday` that takes as argument three numbers, namely the day, month, year information and returns the week day. You can assume that year > 1789.

Here are some examples:

```
(date_to_weekday 15 12 1999)  
WEDNESDAY
```

```
(date_to_weekday 27 7 1821)  
FRIDAY
```

Hint



10 ETUDE ON LIST PROCESSING

1999 HOMEWORK 3

TASK 1 (15 points)

Write a function `flatten` which takes a list of any structure and returns the a (flat) list of all non-list sexprs in it where no repeated sexprs occur.

```
(flatten '((a b) ((c d 2) (b r a)) (((7))))))
```

will return

```
(a b c d 2 r 7)
```

(It is allowed that the result occur in a different order in your implementation)

TASK 2 (15 points)

Write a function `sameset` which takes two argument that are lists of any structures and are representing sets. These lists may contain other lists (to any depth) as elements which are also representing sets. `sameset` shall return `#t` if the two lists are representing the same set and `#f` otherwise.

```
(sameset '(x a (b (c x (y d)) 7) (2 3) u r)
         '(r a u ((x c (d y)) 7 b) x (2 3)))
```

will return

```
#t
```

and

```
(sameset '(x a (b (c (x y) d)) 7) (2 3) u r)
         '(r a u ((x c (d y)) 7 b) x (2 3)))
```

will return

```
#f
```

TASK 3 (20 points)

Write a function `permutation` which takes a single list as argument and produces a list of lists each of which is one permutation of the source list.

```
(permutation '(x a b y))
```

will return

```
((x a b y) (x a y b) (x b a y) (x b y a) (x y a b) (x y b a) (a x b y)
 (a x y b) (a b x y) (a b y x) (a y x b) (a y b x) (b x a y) (b x y a)
 (b a x y) (b a y x) (b y x a) (b y a x) (y x a b) (y x b a) (y a x b)
 (y a b x) (y b x a) (y b a x))
```

(It is allowed that the permutations occur in a different order in your implementation)

TASK 4 (25 points)

Write a function `combination` which takes two arguments. The first is a list l the second is an integer n . `combination` returns a list of all combinations of n elements from the list l .

```
(combination '(x a b y z) 3)
```

will return

```
((b y z) (x y z) (x a z) (x a b) (x a y) (x b z) (x b y) (a y z)
 (a b z) (a b y))
```

(It is allowed that the combinations occur in a different order in your implementation)

TASK 5 (25 points)

Some elements of a set may be equivalent. Such equivalences can be expressed using a list of pairs of equivalent elements. Equivalence is *symmetric* and *transitive*. *Equivalence classes* are subsets, all elements of which are equivalent. Write a function `coalesce`, a function which takes a list of pairwise equivalences and return a list of equivalence classes.

```
(coalesce '((a e) (z f) (m b) (p k)
           (e i) (f s) (b d) (t p)
           (i o) (s v) (d g) (k p)
           (o u) (v z) (g m) (p t)))
```

will return

```
\begin{verbatim}
((a e i o u) (f s v z) (b d g m) (k p t))
```

(It is allowed that the result occur in a different order in your implementation)

11 BRIDGE OPENING BID

2000 HOMEWORK 2

Problem

Assume we are building a program that does the opening bid of contract bridge play. You will be given the hand as a list of 13 atoms each of which is of the form:

$$\mathcal{L}\mathcal{F}$$

Where \mathcal{L} is from the set $\{S,H,D,C\}$, representing the *suits*: ♠, ♥, ♦, ♣, respectively and \mathcal{F} is either an integer in the range $[2,10]$ or a letter from the set $\{A,K,Q,J\}$, representing *Ace, King, Queen* and *Jack*, respectively.

Here is a possible hand:

(H4 HA S3 C10 H9 SA SJ C4 SQ D2 DA HJ C7)

As you have observed the hand is not sorted.

A hand is evaluated to yield an integer *point* value by the following point system:

<i>Ace</i>		4 points
<i>King</i>	also suit length must be > 1	3 points
<i>Queen</i>	also suit length must be > 2	2 points
<i>Jack</i>	also suit length must be > 3	1 points
Void	(means suit length is $= 0$)	3 points
Singleton	(means suit length is $= 1$)	2 points

If a singleton is an *Ace* the points are summed, so it yields 6. A singleton *King* will only yield a 2 (because of being a singleton. The other condition failed to hold).

Furthermore, we define two properties of a hand: *balanced* and *stoppers*.

Balanced : A hand is balanced if there is at least three cards in each suit of the hand.

Stoppers : A hand has stoppers if the hand contains one of the following combinations for each suit:

- *Ace* or *King* and suit length is > 1
- *Queen* and suit length is > 2 .

Based on this, the opening bid schedule¹ is as follows:

¹This system is not "Goren"!

CONDITION	BID
If $points < 13$, and the longest suit has < 7 cards	PASS
If $12 < points < 16$ and \mathcal{L} is the longest suit	(1 \mathcal{L})
If $15 < points < 20$ and either the hand does not have stoppers or is not balanced, and \mathcal{L} is the longest suit	(1 \mathcal{L})
If $15 < points < 20$ and the hand has stoppers and is balanced	(1 NO TRUMP)
If $points \geq 20$ and either the hand does not have stoppers or is not balanced, and \mathcal{L} is the longest suit	(2 \mathcal{L})
If $points \geq 20$ and the hand has stoppers and is balanced	(2 NO TRUMP)
If $6 < points < 13$ and the longest suit has > 6 cards where \mathcal{L} is the longest suit	(3 \mathcal{L})
All other cases	PASS

The rules are tried to be applied from top to bottom. If there are more than one 'longest suite' then prefer in the order of $\clubsuit, \diamond, \heartsuit, \spadesuit$. So, for example, prefer \diamond instead of \spadesuit .

Write a function `OPENBID` which takes a hand as argument and returns the bid according to the schedule given above. You are encouraged to define other (helper) functions.

Your function will be tested with correct input.

12 THEOREM PROVING: WANG'S ALGORITHM

2000 HOMEWORK 3

Introduction

Wang's Algorithm is used to prove theorems in propositional calculus. So, for example, it is possible to start with three promises

$$P \rightarrow Q, Q \rightarrow R, \neg R$$

and prove the proposition $\neg P$.

To begin proving a theorem with Wang's algorithm, all premises are written on the left-hand side of an arrow that we may call the *sequent arrow* (\Rightarrow). Note that the sequent arrow is different than the *implication* arrow. The desired conclusion is written to the right of the sequent arrow. Thus we have:

$$P \rightarrow Q, Q \rightarrow R, \neg R \Rightarrow \neg P$$

This string of symbols is called a **sequent**. This particular sequent contains four *top-level* formulas; there are three on the left and one on the right (it contains more than four formulas if we count embedded ones such as the formula P in $P \rightarrow Q$.)

Successively, we apply transformations to the sequent that break it down into simpler ones. The general form of a sequent is:

$$F_1, \dots, F_m \Rightarrow F_{m+1}, \dots, F_{m+n}$$

where each F_i is a formula. Intuitively, this sequent may be thought of as representing a larger formula,

$$F_1 \wedge \dots \wedge F_m \rightarrow F_{m+1} \vee \dots \vee F_{m+n}$$

Here are the transformation (R1 through R5) and termination (R6 and R7) rules:

R1 : If one of the top-level formulas of a sequent has the form $\neg \square$, we may drop the negation and move \square to the other side of the sequent arrow. Here \square is any formula, e.g., $(P \vee \neg Q)$. If the negation is to the left of the sequent arrow, we call the transformation *NOT on the left*; otherwise it is *NOT on the right*.

R2 : If a top-level formula on the left of the arrow has the form $\square \wedge \Delta$, or on the right of the arrow has the form $\square \vee \Delta$, the connective may be replaced by a comma. The two forms of this rule are called *AND on the left* and *OR on the right*, respectively.

R3 : If a top-level formula on the left has the form $\square \vee \Delta$, we may replace the sequent with two new sequents, one having \square substituted for the occurrence of $\square \vee \Delta$, and the other having Δ substituted. This is called *splitting on the left* or *OR on the left*.

R4 : If the form $\square \wedge \Delta$ occurs on the right, we may also split the sequent as in Rule R3. This is *splitting on the right* or *AND on the right*.

R5 : A formula (at any level) of the form $\square \rightarrow \Delta$ may be replaced by $\neg \square \vee \Delta$, thus eliminating the implication connective.

- R6** : A sequent is considered proved if some top-level formula \square occurs on both the left and right sides of the sequent arrow. Such a sequent is called an *axiom*. No further transformations are needed on this sequent, although there may remain other sequents to be proved. (The original sequent is not proved until all the sequents obtained from it have been proved.)
- R7** : A sequent is proved invalid if all formulas in it are individual proposition symbols (i.e., no connectives), and no symbol occurs on both sides of the sequent arrow. If such a sequent is found, the algorithm terminates; the original *conclusion* does not follow logically from the premises.

Wang's algorithm always converges on a solution to the given problem. Every application of a transformation makes some progress either by eliminating a connective and thus shortening a sequent (even though this may create a new sequent as in the case of R3), or by eliminating the connective " \rightarrow ". The order in which rules are applied has some bearing on the length of a proof or refutation, but not on the outcome itself.

We may now proceed with the proof for our example. We label the sequents generated, starting with S1 for the initial one.

LABEL	SEQUENT	COMMENT
S1:	$P \rightarrow Q, Q \rightarrow R, \neg R \Rightarrow \neg P$	Initial sequent.
S2:	$\neg P \vee Q, \neg Q \vee R, \neg R \Rightarrow \neg P$	Two applications of R5.
S3:	$\neg P \vee Q, \neg Q \vee R \Rightarrow \neg P, R$	R1.
S4:	$\neg P, \neg Q \vee R \Rightarrow \neg P, R$	S4 and S5 are obtained from S3 with R3. Note that S4 is an axiom since P appears on both sides of the sequent arrow at the top level.
S5:	$Q, \neg Q \vee R \Rightarrow \neg P, R$	The other sequent generated by the application of R3.
S6:	$Q, \neg Q \Rightarrow \neg P, R$	S6 and S7 are obtained from S5 using R3.
S7:	$Q, R \Rightarrow \neg P, R$	This is an axiom.
S8:	$Q \Rightarrow \neg P, R, Q$	Obtained from S6 using R1. S8 is an axiom. The original sequent is now proved, since it has successfully been transformed into a set of three axioms with no unproved sequents left over.

Problem

The goal is to write a two-argument function `wang`, that takes two lists, which represent the right-hand side and the left-hand side, as arguments and carries out the Wang's algorithm. The result is either `#T` or `#F` meaning the conclusion is *proved valid* or *proved invalid*, respectively.

The first argument list will be of the structure:

$$(f_1 \ f_2 \ \dots \ f_m)$$

and the second will be:

$$(f_m \ f_{m+1} \ \dots \ f_{m+n})$$

Where, referring to the explanation above, f_i is the Scheme representation of the formula F_i . f_i 's are in *prefix* form (in contrast to F_i). The prefix atoms used for the logical operations is as follows:

INFIX	SCHEME PREFIX
\wedge	and
\vee	or
\rightarrow	>

PREFIX	SCHEME PREFIX
\neg	not

So, to solve the example of above we would call wang as:

(wang '((> P Q) (> Q R) (not R)) '((not P)))

Note: It is allowed to have (and $\square_1 \square_2 \dots \square_k$) and (or $\square_1 \square_2 \dots \square_k$) in the formulas. In other words and and or are nary. You are supposed to generalize the rules accordingly.

13 BATTLESHIP GAME

2001 HOMEWORK 2

Problem

As many of you know there is a game named **Battleship** (Amiral Batti). the game is realized on an $N \times N$ grid that is called the *Battle field* where you place your ships on. The ships have to be placed on the grid so that none of the them is closer to another by less then an empty square. In other words ships cannot touch each other. A sample battle field is given in the figure below. Each ship is represented by a unique letter. All ships have a width of 1 square. Their length though, can vary from 2 to 5 squares. The game is played by two players. Each player places his/her own ships on the battle field in a pattern unknown to the other player. In turn, each player makes a shot to a square of the battle field.

If that square is occupied by one of the ships of the opponent then the opponent will reply by a message of the form "Ship X hit n . time" where X is the letter of that identifies that ship and n is an integer which is $1 + \langle \text{count of hits that ship has received before} \rangle$. If all parts of a ship got hit then the opponent will say "Ship X has sunk".

otherwise the shot is a miss and the opponent call out "miss" (karavana).

The goal of the game is to sink all the ships of the opponent before he/her sinks all of your ships.

	0	1	2	3	4	5	6	7	8	9
0	.	.	A	A	.	.	.	G	G	G
1
2	.	B	.	.	F	F	F	.	.	.
3	.	B
4	.	B	.	.	C
5	C	.	.	E	.	.
6	C	.	.	E	.	.
7	C
8	C	.	D	D	D	D
9

Figure 1: Sample Battle Field

In this homework, you are going to write a *Scheme* program which will perform a non-interactive battlefield simulation for a single player (All shots of the opponent is given together). The program will take a battlefield where the ships are already placed, and a list of square coordinates which stands for all the shots of the opponent. You are not expected to make any shots. The program should return the result of each shot in a list.

The battle field will be given as follows: The Battle field is represented by a single string. Each square of the field will be represented by a single character in this string. `.` is used for

empty squares. Ships are represented by letters other than **m** and **x**. To construct this string, we start from the top left corner of the battle field and list the squares in the *rowwise* order.

Example: The battle field given above will be represented by the string:

```
"..aa...ggg.....b..fff...b.....b..c.....c..e.....c..e.....c.....c.dddd....."
```

The shots of the opponent will be given in a list where each item is a string of 2 digits denoting the coordinates of a shot. First digit is the row number, second digit is the column number of the given shot.

Example: Two shots which will sink the ship A of the example above are ("02" "03")

Instructions

- The name of the main function must be **battlefield**.
- This function will take two arguments. First argument is the battle field represented as a string. Second argument is a list, containing the coordinates of shots (first shot is the first member of the list). It is guaranteed that the second argument does not contain the same coordinate more than once.
- The size of the battle field will always be 10×10 but the number of ships and their size can vary.
- The **battlefield** function should return a list containing the result of each shot and the final battle field, as the last element.

Modification of the battle field should be done as follows:

- If a shot is a miss, then the corresponding square should be marked as **m**.
- If a shot is a hit, then the corresponding square should be marked as **x**.

Answer for each shot should be given as follows:

- If the shot is a miss, then you should return **"miss"** as an answer.
- If the shot has hit ship **g**, then:
 - If the ship is hit for the first time, then you should answer as **"ship g hit 1. time"**,
 - if the ship is hit for the second time, then you should answer as **"ship g hit 2. time"**,
 - and so on for other hits.
 - If all squares of the ship are hit, then you should answer as **"ship g has sunk"**.

Important Note: When you return the result of each shot, you should give the exact result written above. There is one space between the words and all letters are lower case. Since your homeworks will be controlled by a program, any mistake in the result string would be interpreted as a wrong answer. Remember, the control program will not be intelligent!

Example: As a simple example consider a battle field with size 4×4 (Your program will be dealing always with 10×10 cases). If the given battle field is **"..a...a.....bbb."** and the shot list is ("00" "12" "23" "31" "02") then

	0	1	2	3
0	.	.	A	.
1	.	.	A	.
2
3	B	B	B	.

(a) Initial field

	0	1	2	3
0	M	.	X	.
1	.	.	X	.
2	.	.	.	M
3	B	X	B	.

(b) final field

Figure 2: Battle Field of size 4×4

- The final battle field should be `"m.x...x....mbxb."`
- The list containing the result of each shot should be `("miss" "ship a hit 1. time" "miss" "ship b hit 1. time" "ship a has sunk")`

So, for this example the correct function call and its result should be:

```
>(battlefield "..a...a.....bbb." '("00" "12" "23" "31" "02"))
("miss" "ship a hit 1. time" "miss" "ship b hit 1. time" "ship a has sunk"
 "m.x...x....mbxb.")
```

Note: the size of the battle field in this example was given as 4×4 but your program will always be given a battle field of size 10×10 .

If your main function (battlefield) does not work correctly while evaluation, we will check for the following functions to give you partial grade. Note that these functions are not essential for a correct solution of the homework. Attempt only if you think your main function will not work under some conditions.

- Write a function named `bfconvert` which will take a battle field and a list of shots as arguments and returns the final battle field.
- Write a function named `content` which will take a battlefield and a shot as arguments and returns the content of the corresponding square in the battle field.
- Write a function named `answerlist` which will take a battle field and a list of shots as arguments and returns the result of each shot in a list.

If we consider the example given above, the correct function calls and their results should be:

```
>(content "..a...a.....bbb." "00")
"."
>(content "..a...a.....bbb." "12")
"a"
>(bfconvert "..a...a.....bbb." '("00" "12" "23" "31" "02"))
"m.x...x....mbxb."
>(answerlist "..a...a.....bbb." '("00" "12" "23" "31" "02"))
("miss" "ship a hit 1. time" "miss" "ship b hit 1. time" "ship a has sunk")
```

14 DIGITAL CIRCUIT SIMULATOR

2001 HOMEWORK 3

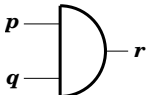
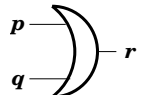
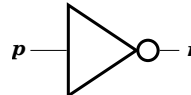
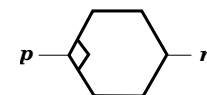
Problem

In this homework you will be writing a digital circuit simulator for the i/o (input/output) level. the digital circuit elements that you will be dealing with is a small subset of the real world. Furthermore, restrictions are imposed to ease your job.

The circuits that you will be dealing with can contain four types of components:

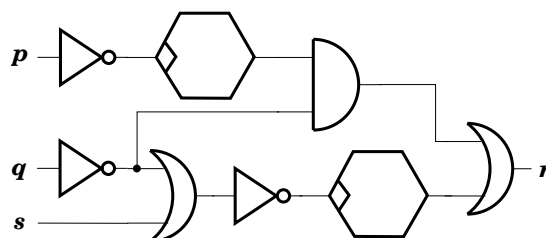
- AND gate
- OR gate
- NOT gate
- τ -flip-flop

Their schematic representations as well as the functions that describe their actions are expressed as truth tables below:

AND	OR	NOT	T-FLIP-FLOP																																																			
																																																						
<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>p</th><th>q</th><th>r</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	p	q	r	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>p</th><th>q</th><th>r</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	p	q	r	0	0	0	0	1	1	1	0	1	1	1	1	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>p</th><th>r</th></tr> </thead> <tbody> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </tbody> </table>	p	r	0	1	1	0	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr><th>p</th><th><i>former</i> r</th><th>r</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	p	<i>former</i> r	r	0	0	0	0	1	1	1	0	1	1	1	0
p	q	r																																																				
0	0	0																																																				
0	1	0																																																				
1	0	0																																																				
1	1	1																																																				
p	q	r																																																				
0	0	0																																																				
0	1	1																																																				
1	0	1																																																				
1	1	1																																																				
p	r																																																					
0	1																																																					
1	0																																																					
p	<i>former</i> r	r																																																				
0	0	0																																																				
0	1	1																																																				
1	0	1																																																				
1	1	0																																																				

The τ -flip-flop has a hidden input which is invoked at the start of each experiment. So, that all τ -flip-flops start a *run* with their *former output* state set to 0.

Below you see a circuit example:



We name a set of input values of a circuit as an *input vector* for the circuit. The input vector has as many elements as the circuit has inputs. A *run* of the simulator is defined as a sequence of input vectors being presented to the inputs.

The purpose of the simulation is to determine what the output of the circuit (in the example r) is after a run.

Note that this output value is not only a function of the last element of the sequence (the last presented input vector). Because the behavior of a τ -flip-flop is depending on its former state

of its output. So the circuit exhibits a memory effect. Hence, even if the same input vector is presented to the input for the second time, it is quite possible that the output may not be the same.

Consider the circuit above. Assume the circuit is presented with two input vectors, which are of the form (p, q, s) and are $(0,1,0)$ and $(0,1,1)$, in sequence. We call this a single *run*.

The p value of the first input, a 0, will be inverted by the not gate to a 1 and fed into the τ -flip-flop. Since at the start of a run all τ -flip-flops are set to have a *former output* state of 0, according to the third row of the truth table, the output of that τ -flip-flop will be 1. This 1 is then fed into the and gate, and so on.

The p value of the second input is again a 0. Again it will be inverted by the not gate that is connected to the p input. This 1 will enter the τ -flip-flop as it was in the previous case. But this time the *former output* state is not 0 but is 1. So according to the fourth line of the truth table the output of that τ -flip-flop will be 0 this time. This time this 0 is fed into the and gate which is different then the previous case.

Specifications

- The circuit is guaranteed to be of combinatorial type. That means if you consider a one directional flow of information from the inputs to the output of each circuit element, no loop is formed in this information flow in the whole circuit.
- You are expected to write a function `run` which takes three arguments and produces either 0 or 1 as result (as explained above). The arguments are as follows:

First argument: A list of input variables.

For the example above: '(p q s)'

Second argument: A list of input vectors where left most is the first to be applied. There is no limitation on the count of elements of this list.

A possible example for the circuit above: '((0 1 0) (0 1 1))'

Third argument: An `sexpr` that describes the circuit.

For the example circuit above this would be:

```
'(or (and (tflipflop (not p)) zort)
      (tflipflop (not (or (zort (not q)) s))))'
```

Here the `zort` atom (word) is used to label an output of some circuit element. The form of usage and definition is as follows:

A usage of the atom (`zort` in the example) in an input position of a circuit element means that there is a wire connection from this point to the point of a circuit element that appears as the second element of a list which has as the first element the same atom.

For the example circuit above: The input of the and gate is such a connection. It is labeled with the atom zort. This atom appears also as a part of the sexpr as (zort (not q)). This all together, means that there is a wire connection from the circuit output of (not q) to the second input of the and gate.

It is possible that the same atom is used at several input positions (meaning that all those inputs are connected by wire and share the same signal data). But, it is not possible that such an atom refers to more than one outputs. In other words such an atom will show up exactly once as the first member of a list.

Note that these labels are not given in the first argument of `run` (because they are not among the input variables of the whole circuit). So, it is a part of your job to find them out.

- On any path of the signal flow that connects any input to the output you are guaranteed to have at most one τ -flip-flop.
- *The output of the `run` function executed with the above mentioned example inputs will be 1.*

15 SCRIPT TO NUMBER

2002 HOMEWORK 2

Problem

In this homework you will convert a written english expression of a single number N which is

$$0 \leq N \leq 9999$$

into its integer equivalent.

Specifications

- You will name your function that does the conversion as `convert`. `convert` will take exactly one argument which is a list of atoms. This list is the verbal expression of the number.
- You are guaranteed that the your function will be tested with **valid** data.
- You are allowed to write other functions that you will use in the definition of `convert`. You are free in naming them.
- You shall try to write short and concise functions. This will influence the grade you will get.

Example

Here is a list of possible inputs and the expected corresponding results.

Input	Output (english)
<code>(convert '(one thousand six hundred and six))</code>	1606
<code>(convert '(one hundred and eleven))</code>	111
<code>(convert '(six thousand one hundred and ninety nine))</code>	6199
<code>(convert '(twenty))</code>	20
<code>(convert '(zero))</code>	0
<code>(convert '(one thousand))</code>	1000
<code>(convert '(one thousand and one))</code>	1001

16 D'HONT ELECTION SYSTEM

2004 HOMEWORK 3

Problem

In this homework you are going to implement the election system which is known as the D'Hont system. This system is used in various countries all over the world as well as in our country. You are given a simplified version of it. (No regional percentage restriction).

The rules are as follows:

- If a participating party does not receive a 10% of **all** the votes used over the country it will get no seats.
- For all of the parties that have a vote sum which is at least equal to 10%, the votes are considered on a region base. There is a known number of seats in the parliament assigned to each region. The first seat is assigned to the party with the highest vote in that region. After this assignment the count of votes for that party is divided by two (the others are left as they are) these counts are compared. The owner of the biggest figure among these will receive the next seat. If it is again the party which had won the first seat then its original count is divided by three and used for the further comparison. Generalizing we can say that if a party has m seats allocated its votes will be divided by $m + 1$ and used in the comparison that leads to the determination of the next seat. This process is continued until all seats are assigned to a party. In case of equality the party with higher votes (without any division) in that region gets the seat. If an equality exists in this figure, the order of the country percentage will be considered. Other possibilities are ruled out in the test data.

Here is an example:

Let us assume that it is parties A, B, C, D, E, F, G that participate the election. All over the country the vote sums are as follows.

A	B	C	D	E	F	G
120000	23000	150000	40000	100000	10000	27000
25.5%	4.9%	31.9%	8.5%	21.3%	2.1%	5.7%

[The 0.1% shortcome in the sum of the percentages comes from round up errors, and is negligible]

So it is only the parties A, C, E that will considered to have a seat. The B, D, F and G will not be considered in the comparison process in none of the regions.

Now let us assume that in a particular region the count of seats is 7 and the vote distribution is as follows.

A	B	C	D	E	F	G
7000	10000	2500	4000	5000	600	2700

Since B, D, F and G is discarded we are left with:

A	C	E
7000	2500	5000

The seat allocation is performed as follows:

Seat	PARTIES			Winner	Reason
	A	C	E		
1.	7000	2500	5000	A	highest
2.	3500	2500	5000	E	highest
3.	3500	2500	2500	A	highest
4.	2333.3	2500	2500	E	higher region percentage
5.	2333.3	2500	1666.6	C	highest
6.	2333.3	1250	1666.6	A	highest
7.	1750	1250	1666.6	A	highest

Hence the seat allocation for this region is:

A	C	E
4 seats	1 seat	2 seats

Specifications

- You are going to write a function with the name `dhont` that admits exactly the following arguments.
 - argument:** A list of party names ($Party_1$ $Party_2$... $Party_N$).
 - argument:** A list with R elements. Each element of this list is a list where the k^{th} list corresponds to the vote distribution obtained in the k^{th} region. These sublists will contain exactly N integers. The i^{th} element of such a sublist is the count of votes that $Party_i$ has got in that region.
 - argument:** A list of integers. The i^{th} element of this list gives the count of seats the i^{th} region will have in the parliament.

Function `dhont` is expected to return a list of lists with exactly N elements. Any member of this list is a sublist with two elements: The first element is a party name and the second is the seats this party has won in the parliament. The result list shall be in the order of descending value of the seats. So the first element is the list of that party name and seats this party has won, which is going to become *majority*, the following is the list that corresponds to the *main opposition* and so on. In case of equality the party which comes first in lexicographical order comes first.

Note that the sum of seats in the result is expected to be equal to the sum of the integers in the list that was entered as the 3. argument.

- You are expected to use recursion.
- You are not allowed to use any higher order function of the book: `every`, `keep`, `accumulate`, `repeated`. Doing so, will automatically be graded as zero.

- **Input example**

Assume the following status and results:

Region	Seats in Parliament	VOTES OF PARTIES						
		A	B	C	D	E	F	G
1	7	7000	10000	2500	4000	5000	600	2700
2	40	80000	10600	20500	5100	35000	7000	22400
3	40	11000	300	118400	15500	34300	1100	100
4	17	22000	2100	8600	15400	25700	1300	1800

The corresponding input would be:

```
(dhont '(A B C D E F G)
        '( (7000 10000 2500 4000 5000 600 2700)
          (80000 10600 20500 5100 35000 7000 22400)
          (11000 300 118400 15500 34300 1100 100)
          (22000 2100 8600 15400 25700 1300 1800) )
        '(7 40 40 17))
```

17 HUFFMAN CODING

2002 HOMEWORK 4

Introduction

One of the best methods for compression based on an alphabet is *Huffman coding* which was discovered by D. Huffman in 1952.

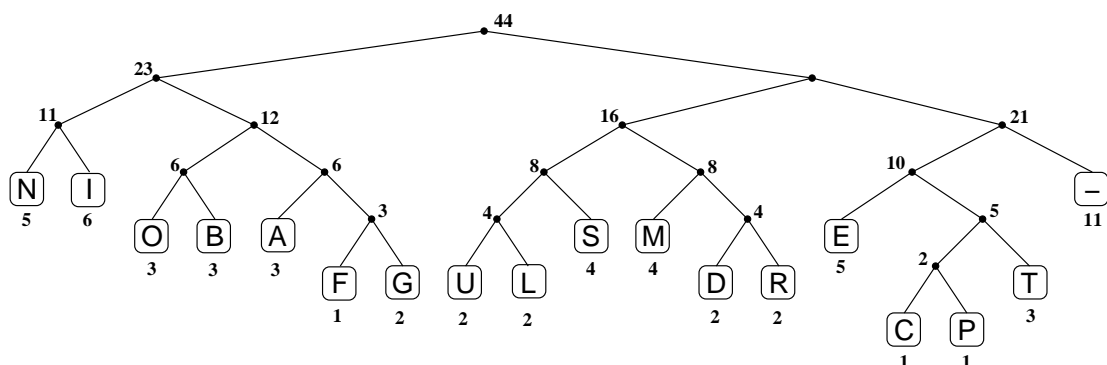
The first step in building the Huffman code is to count the frequency of each character within the message to be encoded. For example, suppose we wish to encode the information

A-SIMPLE-STRING-TO-BE-ENCODED-USING-A-MINIMAL-NUMBER-OF-BITS

There are eleven dashes, three A's, three B's, etc. The count table produced is shown below:

A	B	C	D	E	F	G	H	I	L	M	N	O	P	R	S	T	U	-
3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2	2	11

The next step is to build the coding tree from the bottom up according to the frequencies. In building the tree, we'll view it as a binary tree with frequencies stored in the nodes: after it has been built we'll view it as a tree for coding, as above. First a tree node (leaf) is created for each (non zero) frequency, as shown on the left in the (the order in which the nodes appear is determined by the dynamics of the algorithm described below, but is not particularly relevant to the current discussion). Then the two nodes with the smallest frequencies are found, and a new node is created with those two nodes as children and with frequency value the sum of the values of the children. (It doesn't matter which nodes are used if there are more than two with the smallest frequency.) Then the two nodes with smallest frequency in that forest are found, and a new node created in the same way. Continuing in this way, we build up larger and larger subtrees and at the same time reduce the number of trees in the forest by one at each step (remove two, add one). Ultimately, all the nodes are combined together into a single tree:



Note that nodes with low frequencies end up far down in the tree and nodes with high frequencies end up near the root of the tree. In the figure above the number below the external (square) nodes in this tree is a frequency count, while the number near each internal (round) node is the sum of the labels of its two children.

Now, the Huffman code is derived simply by replacing the frequencies at the bottom nodes with the associated letters and then viewing the tree as an encoding tree, with "left" corresponding to a code bit of 0 and "right" corresponding to a code bit of 1, exactly as above.

The code for N is 000, the code for l is 001, the code for C is 110100, etc.

Clearly, letters with high frequencies are nearer the root of the tree and are encoded with fewer bits, so this is a good code, but why is this the best code?

Property 1 *The length of the encoded message is equal to the weighted external path length of the Huffman frequency tree.*

The *weighted external path length* of a tree is the sum over all external nodes of the *weight* (in this case the associated frequency count) times the distance to the root. Clearly, this is one way to compute the length of the message: it is equivalent to the sum over all letters of the number of occurrences times the number of bits per occurrence.

Property 2 *No tree with the same frequencies in external nodes has lower weighted external path length than the Huffman tree.*

Any tree can be reconstructed by the same process that we use to construct the Huffman tree, but not necessarily picking the two nodes of smallest weight at each step. It can be proven by induction that no strategy can do better than that of picking the two smallest weights first.

The description above gives a general outline of how to compute the Huffman encoding, in terms of algorithmic operations that we've studied.

Problem

You are expected to write two scheme functions, one for encoding and the other for decoding a message by Huffman coding. The encoder function will take a list of characters in (the message) and return a list of two list. The first element is the list representation of the Huffman tree, and the second is the encoded message, namely a list of (only) 0's and 1's.

The decoder function will take in two arguments, the first is a list representation of a Huffman tree and the second is a encoded message (list of 0's and 1's), the return value is a list, which is of the same structure of the first message (a list of characters).

Example

```
>(encode
 '( ( A - S I M P L E - S T R I N G - T O - B E - E N C O D E D - U
     S I N G - A - M I N I M A L - N U M B E R - O F - B I T S ) )
 ((( (N I) ((O B) (A (F G)))) (((U L) S) (M (D R))) ((E ((C P) T)) -)))
(0 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1
 1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 0 1
 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 0 0 1 1 0 1 0 0 0 1
 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0
 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 1 0 0 1
 1 0 1 0 0 0 1 1 1 1 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 1 1 0 0 1 0 1 1
 1 1 1 1 0 1 0 0 0 1 1 1 0 1 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 0 0 1))
>(decode
 '((( (N I) ((O B) (A (F G)))) (((U L) S) (M (D R))) ((E ((C P) T)) -)))
```

```
'(0 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1
  1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 0 1
  1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 0 0 1 1 0 1 0 0 0 1
  0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0
  0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 1 0 0 1
  1 0 1 0 0 0 1 1 1 1 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 1 1 0 0 1 0 1 1
  1 1 1 1 0 1 0 0 0 1 1 1 0 1 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 0 0 1))
```

(A - S I M P L E - S T R I N G - T O - B E - E N C O D E D - U S I N G -
A - M I N I M A L - N U M B E R - O F - B I T S))

Note that it is possible to have a different huffman tree, hence the encoding may differ. But the length of the encoded message (the list of 0's and 1's) is always the same.

Specifications

- For the function `encode`:

Argument: A list of one character atoms (words) which is the message to be encoded.

Return value: A list of two lists:

The first is the list representation of the Huffman tree. The representation is understandable from the example.

The second is a list of 0's and 1's that is the encoding of input message.

- For the function `decode`:

1. argument: A list which is similar to the first element of the return value of `encode`.

2. argument: A list which is similar to the second element of the return value of `encode`.

Return value: A list of the character atoms that make up the transcribed message. Similar to the input argument to `encode`.

- The use of global variables is strictly forbidden. Any program doing so will be graded zero.
- You can make use of any higher order function this time.

Hints and How to Do's

- In the *encode* process, you need to construct an association list (after you have constructed the Huffman tree), for all the used characters in the Huffman tree. An element of such an association list, is a list with two elements, the first is (in your case) a used character, where the second is a list of 0's and 1's that is the encoding of that character. To look up such an association list you may use the built-in function `assoc`. To build the association list you have to traverse the Huffman tree. During the traverse, when you reach a leaf you shall 'memorize' the path that led to it. Be warned that the construction of this Huffman tree \Rightarrow association list is a delicate job and will take some time to get it working for a novice.

18 SOLVING THE $k^2 - 1$ PUZZLE

2003 HOMEWORK 3

Introduction**The $k^2 - 1$ Puzzle**

There is a square board of size $k \times k$ that contain $k^2 - 1$ numbered tile in an arbitrary initial arrangement. The object is to place these tiles into a specific goal arrangement. The tiles may be moved only by sliding one of the tiles onto an adjacent empty square (in other words the movement is orthogonal). Below you see an example of an eight-puzzle problem.

1	2	3
4	8	5
7		6

Initial State

1	2	3
4	5	6
7	8	

Goal State

For the initial state we have illustrated, there are only three possible immediate successor states:

1	2	3
4	8	5
	7	6

1	2	3
4		5
7	8	6

1	2	3
4	8	5
7	6	

We will represent these moves by $(7 \ r)$, $(8 \ d)$ and $(6 \ l)$. As you would guess, if the successor state is obtained by moving a tile up, that would be represented by a letter u.

So, denoting a *solution* of the puzzle is to provide a sequence (a list) of such moves. Our choice is that the left most of the list (the car element) is the first move (the move that is applied to the initial state).

Backtracking

Read the relevant section that explains what backtracking is, from [1997 HOMEWORK 4](#).

Problem

You will write a function `solve-puzzle` which will take a list k^2 elements. The tiles are represented in this list by their labels which are integers in the range $[1, k^2 - 1]$. No label is repeated. The empty position is denoted by the atom `empty`. The order of the input list is the left-to-right, top-to-bottom **reading order**. So the solution to the eight-puzzle example introduced in the introduction section would be initiated by:

```
(solve-puzzle '(1 2 3 4 8 5 7 empty 6))
```

The solution (the value of this function call) is a list of *moves* (the car of this list is the earliest move). A move is denoted by a two element list

(*label direction*)

where *label* is the label of the moved tile. (attention: the empty position is not a tile, therefore no move can have (*empty direction*)). *direction* is one of the letters u, d, l, r. Meaning the tile is moved in *up*, *down*, *left* or *right* direction, respectively.

- $k \leq 10$. As k increases, test cases will be increasingly easier :) .
- Your move sequence does not have to be the minimized. Though if there is/are a student(s) who solves **all** test cases with least moves among the class will receive additional 50% bonus points for this homework (least means that there is no lesser moves found among the class).
- Any illegal move will lead to a zero grade from that test (moving a tile in a direction it cannot move. or see item below).
- In a test case, during the application of the moves, to create a board state (appearance) that was created already is considered as an illegal move. (Yes, our evaluation program will keep and check all intermediate board states)
- All initial boards in test cases will have solutions.
- All goal states have the same form, namely in the *reading order* the labels will strictly increase and the empty position is at the right-bottom corner. Obviously for each k there exists only one such goal state. The goal state of the example in the introduction section is such a goal state.
- No 'Simply Scheme' higher-orders are allowed. The only allowed higher orders are `map` and `for-each`.

19 SKY LINER PROBLEM

2004 HOMEWORK 3

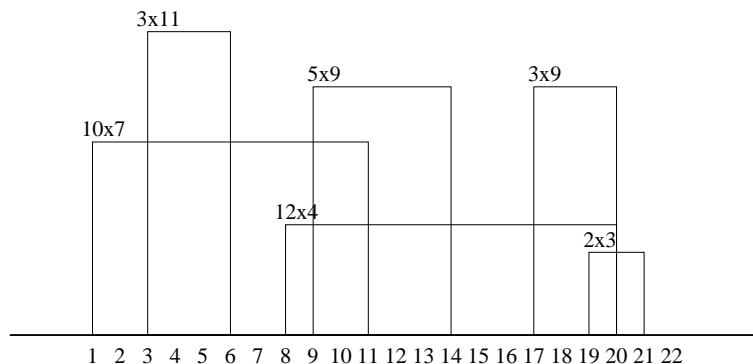
Problem

This problem is known as the *Sky Liner Problem*. It can be summarized as follows: Assume you are looking at a city from a very long distance, the sun is behind the city (from your point of view). You will not be able to recognize individual buildings. The whole city will be observed as a silhouette. The problem is:

Assume you are given the dimensions of the shadow of each building which is a rectangular. Furthermore each of these rectangulars are placed on a common line (the ground) and extent in the same direction (upwards). In addition to the dimensions you are also given the positions of each rectangular. Your duty, if you accept, is to find the vertices of the broken line that surrounds the silhouette.

Example

Consider the below given example



As given in the picture above, there are 6 rectangulars. which could be listed as

WIDTH	HEIGHT	POSITION ON GROUND
3	11	3
10	7	1
5	9	9
12	4	8
3	9	17
2	3	19

'Position on Ground' is the position of the lower-left corner of the rectangular. We will represent the broken line that surrounds the silhouette by a ordered set of vertices where the leftmost vertex on the broken line is stated at the leftmost position of the set, then the one which is next on the broken line is following it, etc. A vertex information will be represented by a 2-tuple:

$$(Position_on_Ground, Height)$$

So for this example the solution would be

```
[(1, 0), (1, 7), (3, 7), (3, 11), (6, 11), (6, 7), (9, 7), (9, 9), (14, 9),
(14, 4), (17, 4), (17, 9), (20, 9), (20, 3), (21, 3), (21, 0)]
```

Specifications

- You are expected to write a function that will be named `skyliner` and takes a single list as argument. This list constitutes of elements which are themselves lists of the form (*Width Height Position_on_Ground*)
- You are given that there are at most 100 rectangles.
- A possible input list for the example above is:

```
((3 9 17)(5 9 9)(12 4 8)(3 11 3)(10 7 1)(2 3 19))
```

- There is no order imposed on the input elements. This means that the rectangle information can be given in any order.
- The output is a list that contains the vertex information of the solution in the order specified in the previous section. (*Attention: There is an order!*)

An element of the vertex information on the output list is a list of two integers. The first integer is the *Position_on_Ground* and the second is the *Height* of the vertex.

The expected output for the above given example is:

```
((1 0)(1 7)(3 7)(3 11)(6 11)(6 7)(9 7)(9 9)(14 9)(14 4)(17 4)(17 9)
(20 9)(20 3)(21 3)(21 0))
```

- **You are restricted to use some of the standard Scheme functions only.** You cannot use any of the *Simply Scheme* functions defined in conjunction with the book. No * marked functions in the book cover is allowed. The list of forbidden functions is:

```
appearances, before?, butfirst (bf), butlast (bl), count, empty?,
first, item, last, member?, sentence (se), sentence?, word,
word?, filter, reduce, children, datum, make-node, accumulate,
every, keep, reduce, repeated, align, all vector functions,
all input/output functions, all file related functions
```

- You can and you shall make use of recursion. But this is not a must.