Count by Writing Code

Saad Mneimneh

Chapter 1

Introduction to Counting

1.1 Counting is about representation

We all know how to count, it is seemingly easy like 1, 2, 3, 4, ... But such enumeration is not efficient or even feasible when the count is large. Other systematic methods are needed to count things like:

- the number of different ways we can select a dozen donuts if we have 5 varieties
- the number of 16 bit patterns with 4 ones

Enumerating the possibilities is very tedious. We might even miss some of them. To give an idea, both scenarios above have the same count: 1820. This may be surprising, but it is not a coincidence. To see this, one must realize that counting is also a matter of representation. For instance, here's a 16 bit pattern with 4 ones:

0010001001000010

If we imagine that the zeros are donuts, we can interpret this as 2 donuts of the first kind, 3 of the second, 2 of the third, 4 of the fourth, and 1 of the fifth. Thus 2+3+2+4+1=12 donuts. It should not be hard to argue that every dozen of donuts correspond to a 16 bit pattern with 4 ones and vice-versa. So representation is a very important aspect of counting. This course will focus on that aspect with an emphasis to create representations for counting problems in a standard way. Once a representation is obtained, the actual counting becomes a simple mechanical procedure. The premise of this course is that if we can transform a counting problem, given as a word problem, into some standard form using a specific programming language (representation), then we have solved the problem. If a computer program can produce the count, so can we.

1.2 Why is counting important?

Here are few reasons why counting is important:

- Modern computer science is almost entirely built on discrete mathematics, in particular combinatorics, which is essentially counting. Counting techniques play an important role in the analysis of algorithms.
- Basic probability theory relies on counting. For instance, in a game of cards, to determine the probability of having a winning hand, one must be able to count the number of winning hands. Probability plays an important role in all sciences.
- Counting leads to insightful results. For instance, equalities such as:

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

$$1 \times 3 \times 5 \times \ldots \times (2n-1) = \frac{(2n)!}{2^n n!}$$

  can be easily proved by showing that both expressions, the one on the left and the one on the right, count the same thing.
- Real life problems, e.g. the donut problem.

1.3 The product rule

In this course, we will rely on a basic principle of counting called the product rule:

If a task consists of n phases, and the i^{th} phase can be carried out in α_i ways, irrespective of how the previous phases are carried out, then the entire task can be carried out in $\alpha_1\alpha_2 \ldots \alpha_n$ ways.

Here's an example: In New York city, every taxi cab has a plate number that consists of a digit, followed by a letter, followed by a digit, followed by a digit. How many taxi cabs can we have? One way to model this problem is by imagining the task of making a plate. This tasks consists of 4 phases. The first phase can be carried out in 10 ways because we have 10 possible digits. The second phase can be carried out in 26 ways because we have 26 possible letters. Similarly, the third and the fourth stages can be carried out in 10 ways each. The number of possible taxi cabs is, therefore, $10 \times 26 \times 10 \times 10 = 26000$.

In general, every problem will look different. So how do we handle this multiplicity of carrying out the phases in a standard way? We will say that each phase represents a single choice, which consists of choosing an element from a set. In other words, the number of ways a phase can be carried out is equal to the size of the set from which we are making that choice.

1.4 Sets and the like

A set is an unordered (unless otherwise specified) collection of elements. Elements in a set have the same nature or "type". We call the set homogenous. A set is expressed using the following notation:

$$\{elem_1, elem_2, elem_3, \ldots\}$$

Therefore, $\{a, b, c\}$ and $\{c, a, b\}$ are the same set.

A tuple is an ordered collection of elements, not necessarily of the same nature. So a tuple may or may not be homogeneous. A tuple is expressed using the following notation:

$$(elem_1, elem_2, elem_3, \ldots)$$

Therefore, $(a, b, 1)$ and $(b, a, 1)$ are different tuples.

1.5 The Taxi problem

Consider the following sets:

$$digit = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$letter = \{a, b, c, \ldots, z\}$$

The task of making a plate can be thought of as follows:

1. choose any elements from digit ... 10 ways
2. choose any elements from letter ... 26 ways
3. choose any element from digit ... 10 ways
4. choose any element from digit ... 10 ways

So the task of making a plate can be carried out in 26000 ways. In this example, the size of the set (whether digit or letter) does not change after making a choice. In other words, if we choose the digit 7 in phase 1, we could still choose the digit 7 in phase 3. In this problem, there was no indication that digits cannot be reused. So we call the set reusable. However, sometimes making a choice reduces the size of the set by 1. This will depend on the setting. Below is an example where the set is nonreusable.

1.6 The 16 bit problem

To make a 16 bit pattern with 4 ones, we have to decide where the ones are. This means, the task of making a 16 bit pattern with 4 ones consists of 4 phases. In each phase, we choose a position for a one. This time, however, a position cannot be reused. So the set is nonreusable

$$pos = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$$

1. choose any element from pos ... 16 ways
2. choose any element from pos ... 15 ways
3. choose any element from pos ... 14 ways
4. choose any element from pos ... 13 ways

So the task of making a 16 bit pattern with 4 ones can be carried out in $16 \times 15 \times 14 \times 13 = 43680$ ways. In the Introduction, it was mentioned that the answer is 1820, so this is wrong! This is a classical scenario of overcounting. While the 43680 ways correspond to "physically" different ways of making the 16 bit pattern, some of them are equivalent. This is because the order by which we make the four choices is irrelevant. So, for instance, if we choose positions 5, 3, 6, and 1, in that order in phases 1, 2, 3, and 4, respectively, then choosing positions 3, 6, 1, 5, in that order results in exactly the same 16 bit pattern.

Therefore, the choices made in the taxi problem are ordered and can be represented as a tuple. The choices made here are unordered and should be represented as a set. This brings us a step closer to writing code.

1.7 Code for Taxi and bit problems

```
#Taxi problem
digit=reusable {0,1,2,3,4,5,6,7,8,9}
letter=reusable 26
representation1=(?digit, ?letter, ?digit, ?letter)
count representation1

#16 bit problem
pos=nonreusable 16
representation2={?pos:4}
count representation2
```

The tuple in the taxi problem represents the 4 phases, where each phase consists of choosing an element from a set. The question mark stands for the word any, and is followed by the name of the set, to indicate that we are choosing any element from that set. The order of the choices is relevant, thus the use of a tuple.

The set in the 16 bit problem represents also 4 phases, where each phase consists of choosing an element from the same set. But the order of those choices is not relevant, thus the use of a set. Observe that we did not write:

```
{?pos, ?pos, ?pos, ?pos}
```

Such notation is not allowed by the language to prevent errors. With a set representation, all elements must be of the same nature, i.e. the choices must be made from the same set. To make that explicit, we use the syntax $\{?x : n\}$, where x is a set and n is the number of phases.

Chapter 2

Good Representations

2.1 Counting with representations

So far, we saw two examples:

- How many NYC taxi plates are there?
- How many 16 bit patterns with 4 ones are there? (equivalent to the donut problem)

In both cases, we approached the problem of counting by focusing on the task of generating one configuration. For instance, the task to generate one taxi plate, or the task to generate a box with a dozen donuts. With this idea in mind, we hope to apply the product rule if that task is carried out in phases and each phase consists of making a choice. The choice is always to pick an element from a set. Sets have different properties, such as:

- nonreusable: choosing an element from the set removes it and reduces the size of the set by 1
- reusable: choosing an element from the set keeps it in the set and does not change the size of the set. That element can be chosen in other phases.

As such, each choice (phase) can be carried out in a number of ways equal to the current size of the set.

This is the paradigm that will make it possible for us to handle all the counting problems we see in the same manner. We now have created a standard way to express our counting problems. So we can actually write code that corresponds to our problem. If a computer program can interpret this code and produce a count, so can we. Counting becomes a simple mechanical task. The most important aspect is, therefore, the mental representation that we create by following the procedure outlined above.

Recall the representation for the taxi problem:

```
(?digit, ?letter, ?digit, ?digit)
```

This representation encapsulates all the aspect mentioned above. First, we see that we need 4 phases to generate a taxi plate. The first phase consists of choosing a digit. The second phases consists of choosing a letter. The third phase consists of choosing a digit. Finally, the fourth phase consists of choosing a digit. This implies that we must define two sets, one for digits and one for letters, with their properties (reusable or nonreusable). In this case, they are reusable because nothing prevents us for instance from choosing the same digit more than once. Furthermore, the use of a tuple indicates that the choices are ordered. Obviously, 1A23 and 1A32 are different plate numbers. The reason is imposed by the nature of the problem.

Also, recall the representation for the 16 bit problem:

```
{?pos:4}
```

Again, this representation encapsulates all the aspects mentioned above. First, we see that we need 4 phases to generate a 16 bit pattern with 4 ones. The first phase consists of choosing a position for the first one. The second phase consists of choosing a position for the second one. The third phase consists of choosing a position for the third one. Finally, the fourth phase consists of choosing a position for the fourth one. This implies that we must define the set of positions, with its property (reusable or nonreusable). In this case, it is nonreusable because we cannot choose a position more than once. Furthermore, the use of a set (not a tuple) indicates that the choices are unordered. Obviously, $\{1, 2, 3, 4\}$ and $\{4, 2, 1, 3\}$ are equivalent. Again, the reason is imposed by the nature of the problem.

2.2 When a representation is good

Those representations that we came up with so far are only mental. This is because they consist of imagining the task of generating one configuration while breaking the task into phases where each phase chooses an element from some set. For instance, we could have used the following representation for the taxi problem:

```
(?digit, ?digit, ?digit, ?letter)
```

Our interpretation is to choose 3 digits and a letter. Physically, we will place that letter in the second position on the plate, but that does not affect the count.

Therefore, being mental, a representation is not unique and, therefore, it is legitimate to ask when a representation is good for the given problem. How do we know our representation correctly mimics the desired setting? For example, we could have used the following representation for the 16 bit problem (and it would have been wrong, why?):

```
(?pos, ?pos, ?pos, ?pos)
```

To ensure that a representation is good, we must ask two golden questions:

- can the representation generate all possible configurations by replacing $?x$ with any element from x (for nonreusable sets those elements must be distinct)?
- do "equivalent" configurations correspond to permutations of the unordered choices within their sets (and vice-versa)?

If the answer is YES for both of them, our representation is good.

Let's examine our two problems again:

- Taxi: It is not hard to see that every plate number can be generated by the representation $(?\mathit{digit}, ?\mathit{letter}, ?\mathit{digit}, ?\mathit{digit})$ by replacing every occurrence of $?\mathit{digit}$ with a digit from $\{0, \ldots, 9\}$ and every occurrence of $?\mathit{letter}$ with a letter from $\{a, \ldots, z\}$. The answer to the first question is YES. The choices in a tuple cannot be permuted, and there are no equivalent configurations; for instance, $(1, A, 2, 3)$ is only equivalent to $(1, A, 2, 3)$; therefore, the answer to the second question is also YES.
- Bits: It is also not hard to see that every 16 bit pattern with 4 ones can be generated by the representation $\{?\mathit{pos} : 4\}$ by replacing every occurrence of $?\mathit{pos}$ (there are 4 of them) with a distinct element from $\{1, \ldots, 16\}$. The answer to the first question is YES. Given two equivalent configurations, say $\{1, 2, 3, 4\}$ and $\{4, 2, 1, 3\}$, it is obvious that one can be obtained from the other by permuting the unordered choices in the representation. Similarly, permuting the unordered choices results in equivalent configurations. Therefore, the answer to the second question is also YES. Observe that if we have used $(?\mathit{pos} : 4)$ instead, we would still be able to generate all configurations, but equivalent configurations do not correspond now to permutations of the unordered choices, simply because we have none.

Nevertheless, we might still make mistakes in our reasoning. The last line of defense is to actually check if the count is correct by considering smaller examples. For instance, we can reduce the digits and letters to $\{0, 1, 2\}$ and $\{a, b\}$ for the taxi problem, respectively, and check if we get the correct count. To check, this time we can actually generate all configurations manually. Similarly, we can reduce the positions for the bits problem to $\{1, 2, 3, 4, 5\}$, or even $\{1, 2, 3, 4\}$ or $\{1, 2, 3\}$ and check if we get the correct count.

2.3 Summary so far

Given a counting problem presented as a word problem, think about the task of generating one configuration. This task consists of phases. Each phases consists of choosing an element from some set.

1. Define your sets and their properties (reusable vs. nonreusable)
2. Make a mental representation of the task using tuples and sets of ordered and unordered choices respectively (this may also help you determine what sets you need)
3. Answer the two golden questions
4. Try on smaller examples and compare the answer to the number you get by explicit enumeration

2.4 Some examples

2.4.1 Handshakes

We have 10 people. They all shook hands. How many handshakes were there?

First, imagine the task of generating one handshake. In the context of making choices by choosing elements from sets, we may choose any person, and then choose any (different) person again. This pair defines a handshake. We observe that our set must be a set of people, and it has to be nonreusable. In addition, the order by which we choose the two people is irrelevant. Our program should look like this:

```
person=nonreusable 10
representation={?person:2}
count representation
```

Ask the golden questions and try for different numbers of people like 2 and 1. Also try the following and compare the two answers for different numbers of people. What do you conclude? Does it make sense?

```
person=nonreusable 10
representation=(?person:2)
count representation
```

2.4.2 Donut

We have 5 varieties of donuts. How many possible boxes of 12 donuts can we make?

We have seen that the 16 bits problem is equivalent to this donut problem. We will now write code specifically for the donut problem. As usual, we focus on the task of generating one box. To do this in phases, we can pick 12 donuts. Each time, the donut comes from a specific variety of donuts. So our set could be simply those 5 varieties. It is reusable because we can pick the same variety multiple times. The order by which we make our 12 choices is irrelevant.

```
donut=reusable {A,B,C,D,E}
count {?donut:12}
```

Ask the two golden questions and try this on a smaller box, say of size 1 or 2. Also, what happens if we change reusable to nonreusable? Try to determine the answer on your own first. Then check. Does it make sense?

Chapter 3

Nested Representations

Sets and tuples may be nested to create more complex representations. We will consider two examples.

3.1 Seating people on chairs

Consider the problem of seating 3 people on 3 chairs. As usual, we start by considering the task (in phases) of generating one possible seating. To do this, we can think of 6 phases to produce 3 assignments of people to chairs, like this:

$$(?person, ?chair, ?person, ?chair, ?person, ?chair)$$

We basically choose any person and any chair to make our first assignment, then any person and any chair to make the second, and finally any person and any chair to make the third. At this point, we also know that we need a set of people and a set of chairs, both nonreusable. Let's assume the following:

```
person=nonreusable {a,b,c}
chair=nonreusable {1,2,3}
```

Now we have to make sure that our representation is good. It is not hard to see that we can generate all possible seatings: we simply have to make the corresponding choices for each phase. What about equivalent configurations? Consider the following:

$$(a, 1, b, 2, c, 3)$$

This seating is equivalent to, for instance, the following:

$$(b, 2, c, 3, a, 1)$$

Therefore, we must be able to obtain one from the other by permutation of the unordered choices. But we have none! One attempt would be to change the tuple to a set, like this:

$$\{?person, ?chair, ?person, ?chair, ?person, ?chair\}$$

But we know that this is not legitimate since person and chair are not the same "type". In fact, we should not be able to permute people and chairs. We observe that only certain types of permutations give equivalent configurations. Essentially, every person must remain on his/her chair. We have to preserve the pairs. So one way to say this is the following:

$$\{(?person, ?chair) : 3\}$$

Observe that all three elements of this set have the same type given by the pair $(person, chair)$. Permuting the unordered choices results in equivalent configurations, and all equivalent configurations can be obtained by permutations. Now the full program is given as follows:

```
person=nonreusable {a,b,c}
chair=nonreusable {1,2,3}
count {(?person,?chair):3}
```

What if we have 2 people and 3 chairs? Almost the same program should work.

```
person=nonreusable {a,b}
chair=nonreusable {1,2,3}
count {(?person,?chair):2}
```

What happens if we change $\{(?person, ?chair) : 2\}$ to $\{?person, ?chair) : 3\}$? Experiment and try to justify/explain the result.

We could also consider the possibility of having more people than chairs; for instance, 5 people and 3 chairs. The following program works:

```
person=nonreusable {a,b,c,d,e}
chair=nonreusable {1,2,3}
count {(?person,?chair):3}
```

Again, experiment with changing the representation to $\{(?person, ?chair) : 5\}$ and try to explain the answer.

3.2 Making teams

We have 6 people and we want to make 3 teams of two. In how many ways can we do that? Here's a representation that is good for this problem:

$$\{\{?person : 2\} : 3\}$$

Observe how close this representation is to the way we actually describe the teams. We have 3 groups (with no particular order), each consists of two people (with no particular order). Consider the two golden questions. Here are some equivalent configurations:

$$\{\{a, b\}, \{c, d\}, \{e, f\}\}$$

$$\{\{b, a\}, \{c, d\}, \{e, f\}\}$$
$$\{\{c, d\}, \{f, e\}, \{a, b\}\}$$

Try to also figure out why the following representation is not good. Observe that it can generate all configurations, and equivalent configurations can be obtained by permutations:

$$\{?person : 6\}$$

Here are some equivalent configurations:

$$\{a, b, c, d, e, f\}$$
$$\{b, a, c, d, e, f\}$$
$$\{c, d, f, e, a, b\}$$

3.3 Incomplete representations

For the two examples above, seating people on chairs, and making teams, it is tempting to come up with incomplete representations. For instance, if we seat two people, the third assignment of person to chair becomes automatic. Similarly, once we form the first two teams, the third team is implicit. As such, one might use the following representations, respectively:

$$\{(?person, ?chair) : 2\}$$
$$\{\{?person : 2\} : 2\}$$

Avoid this kind of "shortcut". While it makes sense logically, one needs to take care of other aspects to make this work. For instance, in both we have two unordered choices. But effectively, they are three, the third being invisible. This will result in a wrong count. But we do not have to revert to guessing when it comes to such heuristics. Again, we need to consider the two golden questions. And in that regard, here are equivalent configurations that do not correspond to permutations of the unordered choices:

Seating people on chairs: $\{(a, 1), (b, 2)\}$ and $\{(a, 1), (c, 3)\}$ are equivalent.

Making teams: $\{\{a, b\}, \{c, d\}\}$ and $\{\{a, b\}, \{e, f\}\}$ are equivalent.

These incomplete representations refer to other problems. The first is the number of ways we can seat 2 out of 3 people on 2 out of 3 chairs. The second is the number of ways we can make two teams of two given 6 people.

3.4 The use of *any* and *next*

So far, we have used the notation $?x$ to signify making a choice by picking *any* element from set x. We will now explore another directive, $!x$, which signifies making a choice by picking the *next* available element from set x. This implicitly imposes an order on the elements of the set, but we do not care so much about the order itself. For one thing, $!x$ can be carried out is only 1 way. Let's consider the problem of making teams. Before, our task looked like this:

1. choose any person ... 6 ways
2. choose any person ... 5 ways
3. choose any person ... 4 ways
4. choose any person ... 3 ways
5. choose any person ... 2 ways
6. choose any person ... 1 way

By the product rule, we have $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$ ways of carrying out the task, which is wrong. But because order is not relevant, the program will compute the correct answer by adjusting based on the supplied representation:

$$\{\{?person : 2\} : 3\}$$

We will see how this is done in the next section. The correct answer is 15.

Here's another way of carrying out the task:

1. choose next person ... 1 way
2. choose any person ... 5 ways
3. choose next person ... 1 way
4. choose any person ... 3 ways
5. choose next person ... 1 way
6. choose any person ... 1 way

This corresponds to the following representation:

$$(!person, ?person, !person, ?person, !person, ?person)$$

We could also make the grouping of pairs explicit:

$$((!person, ?person), (!person, ?person), (!person, ?person))$$

By the product rule, we have 15 ways, which is the correct answer. We will consider below the two golden questions and show that the answer is YES for both, thus proving this is a good representation. For this, assume that person is the set $\{a, b, c, d, e, g\}$ and that the implicit order is the alphabetical order. We first check if we can generate all possible configurations. This time, $!person$ must be replaced by the next available element from set $person$ given the alphabetical order. Given a configuration, person a must be teamed with someone, say b. The choice for phase 1 will always result in a. We can definitely choose b in the second phase. Now the next available person is c, and say that c is teamed with d. The choice for the third stage will result in c, and we can definitely choose

d in the fourth phase. Finally, the next available person is e, and he must be teamed with f. The choice in the fifth phase will result in e, and f is the only choice left for phase six. We generated (a, b, c, d, e, f). But any configuration can be generated in this way, so the answer to the first question is YES. We then argue about equivalent configurations. Observe that there are none! Since the first member of every team is chosen in order, two configurations are equivalent if and only if they are actually the same. The tuple cannot be permuted either. So the answer to the second question is also YES.

What is the important aspect of using *next*? There is nothing specific. It is just another way of envisioning the task of generating one configuration. But it is insightful because it helps us relate different expressions by counting the same thing in different ways. We will explore this fact in the following section.

NOTE: ! cannot be used within { } because it is about order and sets are unordered. In other words, elements obtained by using *next* cannot be permuted. Consider $\{!person : 2\}$. If $\{a, b\}$ is one possibility, then $\{b, a\}$ should be considered equivalent. But they cannot both satisfy the order given by *next*, whatever that order may be: Either a comes before b, or b comes before a.

3.5 How does counting work so far?

So far, we have focused on coming up with representations for the counting problems. The rest was handled by the program. This is the goal. But it is now the time to explore a little bit how the program actually determines the count from the representation.

Given the representation, the count is generated by following the multiplication rule and decreasing the size of a set by 1 after every choice if applicable (e.g. set is nonreusable). To adjust for unordered choices in sets, we divide by $k!$ for each such set, where k is the number of unordered choices in that set. Note: $k! = 1 \times 2 \times \ldots \times k$ and $(0! = 1)$

- Taxi. $(?digit, ?letter, ?digit, ?digit)$. $10 \times 26 \times 10 \times 10$
- Bits. $\{?pos : 4\}$. $(16 \times 15 \times 14 \times 13)/4!$
- Handshake. $\{?person : 2\}$. $(10 \times 9)/2!$
- 3 people and 3 chairs. $\{(?person, ?chair) : 3\}$. $(3 \times 3 \times 2 \times 2 \times 1 \times 1)/3!$
- 2 people and 3 chairs. $\{(?person, ?chair) : 2\}$. $(2 \times 3 \times 1 \times 2)/2!$
- 5 people and 3 chairs. $\{?person, ?chair) : 3\}$. $(5 \times 3 \times 4 \times 2 \times 3 \times 1)/3!$
- Teams. $\{\{?person : 2\} : 3\}$. $(6 \times 5 \times 4 \times 3 \times 2 \times 1)/(2! \times 2! \times 2! \times 3!)$
- Teams. $((!person, ?person) : 3)$. $1 \times 5 \times 1 \times 3 \times 1 \times 1$

  Observe that if we generalize this to $2n$ people and n teams, the above two counts will reveal the equality:

$$\frac{(2n)!}{2^n n!} = 1 \times 3 \times 5 \times \ldots \times (2n-1)$$

That's the importance of counting the same thing in different ways.

- Ficticious example:

```
a=nonreusable 3
b=nonreusable 3
count (?a,{?b:2},{?a:2})
```

$(3 \times 3 \times 2 \times 2 \times 1)/(2! \times 2!)$

Note: this strategy does not work with reusable sets. For instance, recall the donut problem:

```
donut=reusable 5
count {?donut:12}
```

Based on the above strategy, the count will be $5^{12}/12!$, which is wrong. The reason why it fails here is because the set donut is reusable. Therefore, choices are not distinct. Dividing by $k!$ does not adjust for permutations. Examine the following example of permuting 3 things:

$$a, b, c$$

$$a, c, b$$

$$b, a, c$$

$$b, c, a$$

$$c, a, b$$

$$c, b, a$$

There are $3! = 6$ ways of permuting them. In general, the number of permutations of k things is $k!$. This is true only when the k things are distinct. When they are not distinct, we cannot claim this. Here's an example (only 3 permutations exist for the 3 things):

$$a, a, b$$

$$a, b, a$$

$$b, a, a$$

Chapter 4

Program Transformations I

We will now consider one problem with different settings. The idea is to explore what kind of transformations in the program will be required to accommodate the different settings. In addition, we will observe how the different programs will provide us with new perspectives on the problem. In many cases, the transformed programs mimic certain abstractions that we make while counting.

Our main problem (theme) will be the following: we have 5 kids and 3 gifts. In how many ways can we distribute the gifts among the kids?

4.1 Setting 1: at most 1 gift per kid

This is similar to the problem of seating 5 people on 3 chairs (every person gets at most one chair, and only 3 people get to sit). A typical task to generate one configuration would proceed in 6 phases where we first choose any kid and then choose any gift to make a pair, and repeat this three times. The pairs are unordered. Since every kid gets at most one gift, the set of people is nonreusable. The same is true about the gifts because each gift can be gifted at most once. We have the following:

```
kid = nonreusable {a,b,c,d,e}
gift = nonreusable {1,2,3}
count {(?kid,?gift):3}
```

Observe that the representation $\{(?kid, ?gift) : 3\}$ is good. First, we can generate all possible configurations by making the appropriate choice in each phase. Second, all equivalent configurations can be obtained by permutations of the unordered choices and vice-versa.

Based on our knowledge so far of how the program computes the count, we obtained the count as $(5 \times \times 3 \times 4 \times 2 \times 3 \times 1)/3! = 60$.

4.2 Setting 2: no limit on gifts/kid

For this version of the problem, a kid can receive multiple gifts. Therefore, this can be achieved by making the set of kids reusable. The representation $\{(?kid, ?gift) : 3\}$ is still valid.

```
kid = reusable {a,b,c,d,e}
gift = nonreusable {1,2,3}
count {(?kid,?gift):3}
```

Trying to obtain the count again, we now have $(5 \times 3 \times 5 \times 2 \times 5 \times 1)/3! = 5^3 = 125$. The difference is due to the fact that we can now choose the same kid more than once; therefore, when it comes to choosing a kid, we can carry out our choice in 5 ways every time.

4.3 Setting 3: identical gifts, 1 gift/kid

What if the gifts are identical? For instance, each gift can be a one dollar bill. We reasonably assume that these are now indistinguishable. Intuitively, the number of ways we can distribute the dollar bills among the kids should now decrease. We can handle this in different ways. One way would be to change the representation so that more configurations become equivalent. For instance, we can use

$$(\{?kid : 3\}, \{?gift : 3\})$$

Our interpretation is that if $(\{a, b, c\}, \{1, 2, 3\})$ is one such configuration, than kid a gets gift 1, kid b gets gift 2, and kid c gets gift 3. We can verify that this new representation generates all configurations, and that equivalent configurations correspond to permutations of the unordered choices, and vice-versa. Therefore, this representation is good.

```
kid = nonreusable {a,b,c,d,e}
gift = nonreusable {1,2,3}
count ({?kid:3},{?gift:3})
```

There is another way to handle this setting of identical gifts without changing our representation. This can be done by introducing a new kind of set: an identical set. We can instruct the program that all gifts are identical, like this:

```
kid = nonreusable {a,b,c,d,e}
gift = identical 3
count {(?kid,?gift):3}
```

With an identical set, we cannot explicitly list elements. We can only provide the size of the set. This eliminates confusing definitions like:

```
gift = identical {1,2,3}
```

where obviously the elements of the set are not identical!

When a set is declared identical, choosing any elements from that set can be done in 1 way. Since all the elements are identical, it makes no difference what element we choose. An identical set behaves like a nonreusable set, so when we exhaust that set, choosing another element can be done is 0 ways (i.e. cannot be done). For instance, the following program results in 0 because we cannot choose 5 gifts from the set of gifts.

```
kid = nonreusable {a,b,c,d,e}
gift = identical 3
count {(?kid,?gift):5}
```

Now back to the correct program:

```
kid = nonreusable {a,b,c,d,e}
gift = identical 3
count {(?kid,?gift):3}
```

The same technique for checking whether a representation is good applies. For instance, it is not hard to see that we can generate all possible configurations. Furthermore, one possible configuration is $\{(a, \$1), (b, \$1), (c, \$1)\}$, where \$1 indicates one of the identical gifts (a 1 dollar bill). Configuration equivalent to this one correspond to permutations of the unordered pairs, and vice-versa. Therefore, the representation is good. If we do the math, we see that we have $(5 \times 1 \times 4 \times 1 \times 3 \times 1)/3! = 10$ ways to distribute 3 identical gifts among 5 kids. This way of writing the program gives us another perspective: Observe that the use of an identical set has no effect on the count because every phase involving a choice from an identical set contributes a multiplication by 1. This means we can eliminate the identical set entirely. Here's the outcome:

```
kid = nonreusable {a,b,c,d,e}
count {(?kid):3}
```

which is equivalent to:

```
kid = nonreusable {a,b,c,d,e}
count {?kid:3}
```

From this representation, it seems that the problem boils down to choosing 3 kids with no particular order. In fact, that's really what it is. Since the gifts are identical, we only need to choose 3 kids. The choice of 3 kids determines completely who gets a gift. We managed to abstract away the gifts. The only aspect of gifts that we see in the representation is now the number 3, which stands for 3 gifts.

4.4 Setting 4: identical gifts, unlimited gifts/kid

As before, this can be done by turning the set of kids to reusable.

```
kid = reusable {a,b,c,d,e}
gift = identical 3
count {(?kid,?gift):3}
```

Observe that now we cannot rely on the mathematical formula we used so far to produce the count. This is because the unordered choices are not distinct. For instance, one possible configuration is this: $\{(a, \$1), (a, \$1), (b, \$1)\}$, where \$1 indicates one of the three identical gifts (a one dollar bill). Running the program reveals the count 35.

Again, we can eliminate the identical set entirely to reveal a new perspective and abstract away the gifts.

```
kid = reusable {a,b,c,d,e}
count {?kid:3}
```

which boils down to making a choice of 3 kids with possible repetition.

4.5 More on *any* "?" and *next* "!"

We have seen that making a choice can be done by picking any available element from a set or the next available element from a set. We did not mention the rules that govern the use of these two directives. Depending on the properties of sets, they may or may not be used. The following tables lists all cases. A new set type, ordered, is also introduced.

	?	!	count	side effect
nonreusable	✓	✓	$size$ with ?	
			$\min(1, size)$ with !	$size = \max(0, size - 1)$
reusable	✓	✗	$size$	none
identical	✓	✗	$\min(1, size)$	$size = \max(0, size - 1)$
ordered	✗	✓	$\min(1, size)$	$size = \max(0, size - 1)$

To summarize, nonreusable is the typical set where both ? and ! can be used. Every choice reduces the size of the set by 1. Both reusable and identical make no sense with ! because ! assumed an existing order. Therefore, for reusable sets, repeated ! choices keep picking the same element. This is not useful. Similarly, for identical set, it makes no sense to have a notion of order because the elements are identical. For reusable, the size of the set never changes. For identical, the set behaves like nonreusable, so the size of the set is reduced by 1 with every choice. An ordered set also behaves like nonreusable, but forces us to use !.

4.6 Using an ordered set

We will now redo Setting 1 and Setting 2 using an ordered set. This will also give us a new perspective on the problem. Consider the following program:

```
kids = nonreusable {a,b,c,d,e}
gifts = ordered {1,2,3}
#ordered set must use !
count {(?kid,?gift):3}
```

This program will produce an error because gift is an ordered set and forces us to use !. But making this change will also fail:

```
kids = nonreusable {a,b,c,d,e}
gifts = ordered {1,2,3}
# ! cannot be used within { }
count {(?kid,!gift):3}
```

The reason of the failure now is that ! cannot appear within { }. We are forced to change our representation as follows:

```
kids = nonreusable {a,b,c,d,e}
gifts = ordered {1,2,3}
count ((?kid,!gift):3)
```

But does that provide a good representation? We need to ask the two golden questions. Every configuration looks like this: $((-,1),(-,2),(-,3))$ where the gifts appear in order. By choosing any kid for each phase, we can make every possible assignment of gifts to kids. So the answer to the first question is YES. When it comes to the second question, observe that there are no equivalent configurations, also because gifts appear in order. Tuples cannot be permuted either. Therefore, the answer to the second question is also YES. So this is indeed a good representation.

In fact, if s is a nonreusable set of size n, then the following are equivalent:

$$\{(?s,...):n\}$$

$$((!s,...):n)$$

Now, in a way similar to identical sets, we can also eliminate the ordered set entirely. The use of ! contributes a multiplication by 1, so there is no need to keep it around. By doing this, we end up with the following program:

```
kids = nonreusable {a,b,c,d,e}
count (?kid:3)
```

Observe that this boils down to choosing 3 kids with order. This is a new way of looking at the setting. We abstracted away the gifts again. The result is obvious: $5 \times 4 \times 3$.

Here's another program for the setting of unlimited gifts/kid.

```
kids = reusable {a,b,c,d,e}
gifts = ordered {1,2,3}
count ((?kid,!gift):3)
```

and its equivalent counterpart by eliminating the ordered set.

```
kids = reusable {a,b,c,d,e}
count (?kids:3)
```

Again, the result is obvious: $5 \times 5 \times 5$.

Note: It is tempting to think that we can always eliminate identical and ordered sets. This is not true. There are scenarios where this cannot be done. A trivial scenario is when the count is 0 because one of these sets have been exhausted. Eliminating the set in this case will change the count. Here's an example:

```
s = identical 3
t = nonreusable 4
count ((?s,?t):4)
```

With ordered sets, it is even more tricky. By eliminating an ordered set, we can sometimes transform choices from being distinct to not distinct. This changes the count. Here's an example:

```
s = ordered 3
t = reusable 3
count {(?s,?t):3}
```

Chapter 5

Program Transformations II

We will continue with examples of using different kinds of sets, i.e. nonreusable, reusable, identical, and ordered, to come up with different representations. Those representations provide us with different flavors of looking at the given problem. As it turns out, in many cases, we can reproduce a simple program from a more complex one, and this exposes the kind of abstraction that we sometimes naturally perform without giving too much thought.

5.1 All binary patterns

Consider the problem of counting the number of 10 bit binary patterns. The task of generating a pattern consists of 10 phases, each chooses a bit from the set $\{0, 1\}$ for a given position in the pattern. Since we can choose the same bit (a 0 or a 1) multiple times, this set is reusable. Therefore, we have the following program:

```
bit = reusable {0,1}
count (?bit:10)
```

Answering the two golden questions reveals that the above representation can generate all possible patterns, and no patterns are equivalent (similarly, no permutations are possible). Therefore, the representation is good. The answer to this counting problem is $2 \times 2 \ldots \times 2$ (10 times), which is $2^{10} = 1024$.

Let us now try to be more elaborate about generating a pattern; we will do this by explicitly considering the set of positions in the pattern. One way is to repeatedly select a position and a bit, to form a pair. We then make 10 of these pairs. The order by which we make the pairs is irrelevant. Therefore, we are looking at something like:

```
bit = reusable {0,1}
pos = nonreusable 10
count {(?pos,?bit):10}
```

It is important to understand why the pairs are unordered. Again, one should ask the two golden questions and make sure that permuting the unordered choices corresponds to equivalent configurations and vice-versa. But what if we wanted to only use tuples? Consider for instance the following variation:

```
bit = reusable {0,1}
pos = nonreusable 10
count ((?pos,?bit):10)
```

If we run this program, we discover that we overcount a lot (well, by a factor of 10!, the factorial of 10). Is there a way to fix this while keeping the tuple representation? Indeed, there is:

```
bit = reusable {0,1}
pos = nonreusable 10
count ((!pos,?bit):10)
```

By using *next* instead of *any* we assign the positions in order. What this accomplishes is that there are no more equivalent configurations, because equivalent patterns inevitably become the same. In addition, we have no unordered choices to permute. So the representation is good.

Observe that *any* is never used in making choices from the set *pos*. Therefore, this set may as well be an ordered set. In fact, it is actually a good idea to make it ordered to explicitly express the fact that we must select positions in order. Doing so will disable the use of *any* with this set.

```
bit = reusable {0,1}
pos = ordered 10
count ((!pos,?bit):10)
```

Observe now that being an ordered set, *pos* can be eliminated entirely from the program. And once we do that, we retrieve the original simple form:

```
bit = reusable {0,1}
count (?bit:10)
```

Therefore, this form of the program tells us that we had mentally abstracted away the notion of the position to start with. And we did that precisely by assuming an implicit order on the positions. While this may seem a natural thing to do, the above transformation in the program makes that mental process explicit, and provides us with an understanding of what was going on in our mind.

5.2 Some binary patterns

Consider now the problem of counting the number of 10 bit binary patterns with 3 ones. We have seen this problem before in the setting of the donut problem, where we wanted to count the number of 16 bit patterns with 4 ones. Therefore, we can adapt our previous solution:

```
pos = nonreusable 10
cout {?pos:3}
```

The answer to this counting problem is $(10 \times 9 \times 8)/3! = 120$. We will now make things more explicit. One way to do this is by considering, in addition to the set of positions, a set of 3 ones and a set of 7 zeros. A natural choice for the property of these two sets is to use the identical kind. Consider the following program:

```
pos = nonreusable 10
onebit = identical 3
zerobit = identical 7
count ({(?pos,?onebit):3},{(?pos,?zerobit):7})
```

The above representation can generate all possible patterns of 10 bits with exactly 3 ones (and 7 zeros). Also, permuting the unordered choices (either the ones or the zeros) leads to equivalent patterns. Similarly, equivalent patterns correspond to permutations of those unordered choices. So the representation is good.

Finally, knowing that *onebit* and *zerosbit* are identical sets, we can eliminate them entirely from the program, to reveal this form:

```
pos = nonreusable 10
count ({?pos:3},{?pos:7})
```

While this form seems a bit different than our original attempt, observe that it is actually equivalent. For instance, after choosing 3 ones, there is only one way we can choose 7 zeros (we only have 10 bits). Similarly, after choosing 7 zeros, there is only one way we can choose 3 ones. Therefore, one of the two unordered choices is indeed redundant.

Our original solution dropped one of them. In fact, we could have also done the following:

```
pos = nonreusable 10
count {?pos:7}
```

Again, this transformation in the program exposes the kind of abstraction that was naturally occurring in our mind. Our original solution precisely reflects the idea that the following representations are equivalent for a nonreusable set x of size n.

$$\{?x : k\}$$
$$\{?x : n - k\}$$
$$(\{?x : k\}, \{?x : n - k\})$$

Finally, to verify the math: $(\{?pos : 3\}, \{?pos : 7\})$ corresponds to $(10 \times 9 \times 8 \times 7 \times \ldots \times 1)/(3!7!)$ and $\{?pos : 3\}$ corresponds to $(10 \times 9 \times 8)/3!$. Observe that the second part of the numerator in the first expression is 7! which simplifies with the 7! in the denominator to give the second expression.

5.3 Anagrams

Consider the word "cat". How many words can be obtained by shuffling the letters of "cat". We call these anagrams. It should not be hard to see that this is similar to the problem of counting the number of ways we can seat 3 people on 3 chairs. Simply call the people by the letters of the given word. So we have person c, person a, and person t.

```
person = nonreusable {c,a,t}
chair = nonreusable 3
```

In light of Section 2, we could solve this problem using the following representations (the last being obtained by changing the person set to ordered and eventually eliminating it):

$$\{(?person, ?chair) : 3\}$$

$$((!person, ?chair) : 3)$$

$$(?chair : 3)$$

The count is therefore, $3 \times 2 \times 1 = 3!$. One can then conclude that the number of anagrams, also the number of permutations, is generally obtained as the factorial of the number of objects (letters in this case).

For instance, with the word cat, we have $3! = 6$ anagrams. However, we will start to have problems when letters repeat. This can never happen in the seating problem because people are always distinct. Take for instance, the word "bob". It has only 3 anagrams (find them). How can we take into consideration that some letters repeat?

Consider the word "mathematics". Let us attempt to first create a set for these letters:

```
letter = nonreusable {m,a,t,h,e,m,a,t,i,c,s}
```

When running this program, we will notice that the set thus created has only 8 elements! This is because we cannot have duplicates in a set. However, we have a special kind of set where duplication is implicit: the identical set. Let us try to rewrite our program by thinking that each letter can be represented by an identical set with the appropriate number of occurrences. We can also think about positions (11 of them).

```
m = identical 2
a = identical 2
t = identical 2
h = identical 1
e = identical 1
i = identical 1
c = identical 1
s = identical 1
pos = nonreusable 11
```

The task is now to choose for every position a letter, we need to exhaust all the letters that are available to us. Something like:

$$((?pos, ?m), (?pos, ?m), (?pos, ?a), (?pos, ?a), (?pos, ?t), (?pos, ?t),$$

$$(?pos, ?h), (?pos, ?e), (?pos, ?i), (?pos, ?c), (?pos, ?s))$$

Not all pairs have the same type, because letters are now represented as sets. But some of these pairs do have the same type, precisely those that correspond to the same letter. For such pairs, the order is irrelevant. For instance if we put an m is position 1 and an m in position 2, switching the order of these pairs results in the same outcome. Let's finish this idea:

```
m = identical 2
a = identical 2
t = identical 2
h = identical 1
e = identical 1
i = identical 1
c = identical 1
s = identical 1
pos = nonreusable 11
count ({(?pos, ?m):2}, {(?pos, ?a):2}, {(?pos, ?t):2},
       (?pos, ?h), (?pos, ?e), (?pos, ?i), (?pos, ?c), (?pos, ?s))
```

Using our knowledge of counting so far, we have

$$\frac{10 \times 1 \times 9 \times 1 \times 8 \times 1 \ldots \times 1 \times 1}{2!2!2!} = \frac{11!}{8}$$

Observe that 11! is the total number of permutation had the 11 letters been distinct.

Now, we can eliminate the identical sets entirely. We obtain:

```
pos = nonreusable 11
count ({?pos:2}, {?pos:2}, {?pos:2}, ?pos, ?pos, ?pos, ?pos, ?pos)
```

We can see now that the number of anagrams depends only on the letter count in the word. The word "mathematics" can be described as $(2, 2, 2, 1, 1, 1, 1, 1)$. This technique can be generalize to any word. For instance, the number of anagrams for the word "mississippi", which can be described using the letter count $(1, 4, 4, 2)$, can be obtained using the following program:

```
pos = nonreusable 11
count (?pos, {?pos:4}, {?pos:4}, {?pos:2})
```

Chapter 6

Split Counting

We will continue with the idea of thinking about problems in different ways and how this may lead to different programs. In particular, we will explore the idea of "splitting" the count. By analyzing the programs, we discover pitfalls and eventually ways to adjust the programs to reflect the correct interpretation of the problem. Our main theme will be playing with cards.

6.1 Warm up

Assume he have one regular deck of cards, and consider the following problem: In how many ways can we pick J, Q, and K? We will assume that the order of picking those cards is not important. Therefore, we focus on the suits.

This represents another problem in disguise. For instance, if we call the cards gifts, and the suits kids, we will recognize that this is the problem of distributing gifts to kids, where every kid may receive multiple gifts. Therefore, we can make the set of suits reusable, and end up with the following program:

```
card = nonreusable {J,Q,K}
suit = reusable {diamond, heart, spade, club}
count {(?card,?suit):3}
```

The answer to this count is $(3 \times 4 \times 2 \times 4 \times 1 \times 4)/3! = 4^3$. In light of the previous lecture, we can also modify the program by choosing cards in some order:

```
card = nonreusable {J,Q,K}
suit = reusable {diamond, heart, spade, club}
count ((!card,?suit):3)
#question: instead, can we use ‘‘!’’ with suit and ‘‘?’’ with card?
#why or why not?
```

Obviously, card may as well be an ordered set, which can eventually be eliminated as usual, to finally obtain:

```
suit = reusable {diamond, heart, spade, club}
count (?suit:3)
```

6.2 Spice it up

Let us now make it a bit interesting: In how many ways can we pick J, Q, and K with exactly 2 suits? One way to approach this problem is by relying on our previous solution. Let's assume that we only have two suits, and we will worry about this aspect later. But given 2 suits, in how many ways can we pick J, Q, and K? Isn't that exactly the previous problem with a smaller set of suits? Let's give it a try:

```
card = nonreusable {J,Q,K}
selectedsuit = reusable 2
#we will worry later about how we
#actually end up with just 2 suits
count {(?card,?selectedsuit):3}
```

The count is $2^3 = 8$, but this is wrong. Assume, for the sake of illustration, that our set of suits is $\{d, h\}$. We have 8 possibilities for J, Q, and K:

J	Q	K
d	d	d
d	d	h
d	h	d
d	h	h
h	d	d
h	d	h
h	h	d
h	h	h

As we can see above, we do not guarantee that we end up with **exactly** 2 suits, but certainly at most 2 suits. In other words, it is possible to choose the same suit for all cards, because the set of suits is reusable. How can we avoid this problem? On the one hand, we need suit to be reusable because we want the ability to choose a suit multiple times. On the other hand, we want to avoid the possibility of choosing the same suit for all the cards.

It is not impossible to solve this problem. It is simply an issue of representation. If we carefully think about the problem, we discover that the only way it can be realized is by assigning two cards the same suit, and one card another. So we can make the set of suits nonreusable. We choose a suit, and two cards for that suit, and another suit, and the last card for that suit. Something like this:

$$(?suit, (?card : 2), ?suit, ?card)$$

But observe that the cards with the same suit must be unordered. For instance $(d, (J, Q), h, K)$ is equivalent to $(d, (Q, J), h, K)$. This is easy to fix:

$$(?suit, \{?card : 2\}, ?suit, ?card)$$

Now we can finalize this part of the program, assuming we have already determined the two suits in question:

```
card = nonreusable {J,Q,K}
selectedsuit = reusable 2
#we will worry later about how we
#actually end up with just 2 suits
count (?selectedsuit,{?card:2},?selectedsuit,?card)
```

To finish, we need to account for the number of ways we can actually choose 2 suits out of 4. This can be done as follows:

```
card = nonreusable {J,Q,K}
suit = nonreusable {diamond, heart, spade, club}
#number of ways we can choose 2 suits
a = count {?suit:2}
#now make suit a set of 2
selectedsuit = nonreusable 2
b = count (?selectedsuit,{?card:2},?selectedsuit,?card)
a*b
```

The result of the first count is $4 \times 3/2! = 6$ and the result of the second count is $(2 \times 3 \times 2 \times 1 \times 1)/2! = 6$. Therefore, the final result is 36.

This kind of "split" in counting is a bit tricky. One has to make sure that the entire process makes sense. We are dealing with two representations. Each is good for its own part of the problem. But how do we know that both are good together? Here's a example:

```
card = nonreusable {J,Q,K}
suit = nonreusable {diamond, heart, spade, club}
#number of ways we can choose 2 suits
#observe the change in the count below
a = count (?suit:2)
#now make suit a set of 2
selectedsuit = nonreusable 2
b = count (?selectedsuit,{?card:2},?selectedsuit,?card)
a*b
```

In this version of the program we made the choice of 2 suits ordered (we used a tuple instead of a set). This will obviously change the overall result. The first count is now 12, making the final result 72 (twice as before). Which version is correct? The first version is.

The second part of the program did not make any assumption about the order by which the first part made the selection of the two suits. It was simply treating those suits as a set of 2. Therefore, an unordered choice in the first part is appropriate. If the choice in the first part was ordered, like in the above version, then we should use "!" instead of "?" in the second part to conform

with that ordered choice. In summary, the two parts must be compatible. The whole problem is due to the fact that selectedsuit has technically nothing to do with suit and the way we select from it. It is simply an intermediate set that exists in our mind. Therefore, the first part and the second part of the program are not coupled in any way. In the future, we will discuss ways of "splitting" the count that will mitigate this inconsistencies. For now, one way to make sure things are good is the following: the two representations can be combined into one tuple if they use no sets in common. For example:

$$((?suit : 2), (?selectedsuit, \{?card : 2\}, ?selectedsuit, ?card))$$

Using this combined representation, we can ask the golden questions to make sure that we can generate all possible configurations and that equivalent configurations correspond to permutations of unordered choices, and vice-versa. We fail on the second condition. For instance, the following configurations are equivalent but cannot be obtained by permuting unordered choices:

$$((d, h), (d, \{J, Q\}, h, K))$$

$$((h, d), (d, \{J, Q\}, h, K))$$

In the first version of the program, the tuple (h, d) is replaced by $\{h, d\}$, which eliminates the problem.

We could have also done this:

```
card = nonreusable {J,Q,K}
suit = nonreusable {diamond, heart, spade, club}
#number of ways we can choose 2 suits
#observe the change in the count below
a = count (?suit:2)
#now make suit an ordered set of 2
selectedsuit = ordered 2
b = count (!selectedsuit,{?card:2},!selectedsuit,?card)
a*b
```

Combining the two representations in one tuple gives:

$$((?suit : 2), (!selectedsuit, \{?card : 2\}, !selectedsuit, ?card))$$

There are no more equivalent configurations. For one thing,

$$((h, d), (d, \{J, Q\}, h, K))$$

is not an allowed configuration because it does not obey the order of the first set in selecting the suits. Observe that in both cases we have to keep an implicit connection between the choices we make from suit and the elements of selectedsuit.

Both programs could have replaced the splitting by a combined tuple:

```
card = nonreusable {J,Q,K}
suit = nonreusable {diamond, heart, spade, club}
selectedsuit = nonreusable 2
count ({?suit:2},(?selectedsuit,{?card:2},?selectedsuit,?card))
```

```
card = nonreusable {J,Q,K}
suit = nonreusable {diamond, heart, spade, club}
selectedsuit = ordered 2
count ((?suit:2),(!selectedsuit,{?card:2},!selectedsuit,?card))
```

On a different note, we did not have to split the count in two parts. The fact that we have exactly 2 suits could have been expressed directly with one representation in this case (eliminate selectedsuit from the above program):

```
card = nonreusable {J,Q,K}
suit = nonreusable {diamond, heart, spade, club}
count (?suit,{?card:2},?suit,?card)
```

6.3 Make it tricky

Now we will make the problem even more tricky. In how many ways can we select J, J, and K? Let's adapt the solution to the warm up problem and see if it works; after all, changing one of the Js to Q retrieves the exact same problem.

```
#observe that we must use J1 and J2
#if they are in the same set
card = nonreusable {J1,J2,K}
suit = nonreusable {diamond, heart, spade, club}
count {{?card,?suit):3}
```

This will of course reproduce the count 4^3 as before. This time, however, it is wrong. Intuitively, we expect to see a smaller count since the two Js have the same meaning (unlike a J and a Q). Let's us explore the two golden questions. Can we generate all possible configurations? The answer is yes. Simply make the choices you want for your J, J, and K. Are there equivalent configurations? Well, let's see. Fix a configuration, say:

$$\{(J1, d), (J2, h), (K, s)\}$$

Here's a configuration that is equivalent, but cannot be obtained by permuting the unordered choices.

$$\{(J2, d), (J1, h), (K, s)\}$$

Actually, this is not the only problem. This is the first time we see this, but this representation, though it can generate all possible configurations, it can also generate more. It can generate configurations that are not valid. For instance, two Js with the same suit:

$$\{(J1, d), (J2, d), (K, s)\}$$

In light of this, one must modify the first question to make sure that a representation can generate all possible configurations and only those. Generating configurations that are not valid (garbage) is not a good thing.

How can we solve this problem. Again, thinking about the problem reveals that the two Js must have different suits. While J and K are unconstrained, the second J must be different than the first. Let's deal with that by splitting the problem again in two parts. In the first part, we handle the case of J and K in the standard way by adapting the solution to the warm up problem. In the second part, we worry about the second J.

```
card1 = nonreusable {J,K}
card2 = nonreusable {J}
suit = reusable {diamond, heart, spade, club}
a = count {(?card1,?suit):2}
#now change suit to contain only
#the 3 allowed for the second J
allowedsuit = reusable 3
b = count (?card2,?allowedsuit)
a*b
```

Each representation is good for its own part. How do we know if they are compatible? Again we observe that the two representations use no common sets. So we can combine them into one tuple:

$$(\{(?card1, ?suit) : 2\}, (?card2, ?allowedsuit))$$

Consider the following two configurations:

$$(\{(J, d), (K, h)\}, (J, h))$$

$$(\{(J, h), (K, h)\}, (J, d))$$

These two configurations are equivalent but do not correspond to permutations of unordered choices. We can easily see that every configuration has exactly another one that is equivalent but cannot be obtained by permutations (we exchange the suits of the two jacks). So we conclude that we are overcounting by a factor of 2. In fact, the count is $(2 \times 4 \times 1 \times 4 \times 1 \times 3)/2! = 48$. So we know that the answer should be 24. Let's try to fix the program. The problem is that we cannot exchange the suits for the two Js because they do not belong to unordered choices. So let's handle the Js together, then worry about the K. In doing so, we make the set of suits nonreusable to make sure the two Js receive different suits:

```
card1 = nonreusable {J1,J2}
card2 = nonreusable {K}
suit = nonreusable {diamond, heart, spade, club}
#the following representation is not good
a = count {(?card1,?suit):2}
```

```
#now choose anything for the K
b = count (?card2,?suit)
a*b
```

As indicated in the program itself, the first representation is not good. Consider for instance the following configuration:

$$\{(J1, d), (J2, h)\}$$

This is equivalent to

$$\{J2, d), (J1, h)$$

But the two cannot be obtained by permutation of unordered choices. The only way to fix this is by using the representation (verify):

$$(\{?card1 : 2\}, \{?suit : 2\})$$

So the correct program becomes:

```
card1 = nonreusable {J1,J2}
card2 = nonreusable {K}
suit = nonreusable {diamond, heart, spade, club}
#the following representation is now good
a = count ({?card1:2},{?suit:2})
#now choose anything for the K
b = count (?card2,?suit)
a*b
```

Since the two representations have a set in common (suit in this case), we cannot combine them into one tuple. But we could if we create a duplicate set of suits with a different name. We can then verify the two questions and make sure that the representations are compatible. In fact, it is easy to verify that the result of this program is 24.

Why did we have a problem with the representation $\{?card, ?suit) : 2\}$ above? The reason is that J1 and J2 are identical. Maybe it is a better option to use an identical set. It turns out, this will work nicely:

```
card1 = identical 2
card2 = nonreusable {K}
suit = nonreusable {diamond, heart, spade, club}
a = count {(?card,?suit):2}
#now choose anything for the K
b = count (?card2,?suit)
a*b
```

Observe that the two configurations that gave us problems before are now both the same:

$$\{(J, d), (J, h)\}$$

which alleviates the problem. Computing the answer gives 24 again.

As a twist, what if we had the cards J, J, Q, and K? The same solution can be applied in principle, but we have one tiny problem. In the second part for choosing suits for Q and K, we need the set of suits to be reusable. We could either create a new set, or we can change the property of suit. Here's how:

```
card1 = identical 2
card2 = nonreusable {Q,K}
suit = nonreusable {diamond, heart, spade, club}
a = count {(?card,?suit):2}
#now choose anything for the Q and K
suit = makereusable suit
b = count (?card2,?suit)
a*b
```

The makereusable command creates a new set with the reusable property from a given set. Thus, we are creating a set called suit that is reusable from suit itself. Therefore, we change suit to reusable. There are two other commands that work in a similar way: makenonreusable and makeordered. There is no makeidentical command (because it does not make much sense). Similarly, we cannot create a nonreusable, reusable, or ordered set from an identical set. So the following commands cause errors:

```
s = identical n
# all three lines below produce an error
t = makenonreusable s
t = makereusable s
t = makeordered s
```

As a final exercise, try this problem: in how many ways we can select J, J, and K with exactly two suits?

Chapter 7

Split Counting with *one* and *all*

We will now focus on the idea of how to better handle a split count. As we have seen in the examples of the previous lecture, special care has to be given when the count is split into two representations. Here's an scenario that is now familiar: We have 5 people and 3 chairs, in how many ways can we seat the people? A typical solutions that we have seen is the following:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
count {(?person,?chair):3}
```

However, one could think about first choosing the 3 people who will set, then count the number of ways we can seat 3 people on 3 chairs. Here's the program based on the split count:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
a = count {?person:3}
selected = nonreusable 3
b = count {(?selected,?chair):3}
a*b
```

There is no obvious advantage of splitting in this particular example because the second part looks pretty much the same as our original attempt. But this example is used for the sake of illustration. What are the problems?

- Coupling: the first and second parts are completely separate except in our mind. In fact, the creation of an arbitrary new set makes no connection to the original set. As such, changes to the first part do not affect the second, which is problematic. For instance, changing the first representation will produce the wrong count:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
#observe the change to the representation
#which does not affect the second part
a = count (?person:3)
selected = nonreusable 3
b = count {(?selected,?chair):3}
a*b
```

- Multiple representations: counting is done by the use of two representations in this case. Is there a way to combine them into one?

7.1 The *one* function

Our goal in the second part is to create a set that represents the choice made in the first part. In other words, the first part counts the number of ways we can select 3 people out of 5. A set of 3 people is therefore our choice. The second part should act on this set. Using the *one* function, we can create such a set. The statement $s = one\ \{rep : n\}$ creates a nonreusable set s of size n, while the statement $s = one\ (rep : n)$ (thus the tuple must be homogeneous) creates an ordered set s of size n. We could verify this claim by using the function *size*, which gives the size of a set. In both cases, the representation *rep* must contain a choice from a nonreusable or ordered set to make the elements distinct.

We could rewrite the above program in this way:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
rep = {?person:3}
a = count rep
selected = one rep
#try size
size selected
b = count {(?selected,?chair):3}
a*b
```

At the first glance, nothing much has changed. But a careful examination will reveal that making changes to rep affects the selected set. In particular, if we make the following change:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
#observe the change to rep
rep = (?person:3)
a = count rep
selected = one rep
b = count {(?selected,?chair):3}
a*b
```

Then the second count statement will produce an error because selected is now an ordered set and, therefore, we cannot use *any* on it. This leads us to think about a new representation that is compatible and, obviously, we have to change *any* to *next*. Observe also that "! cannot be used within { }. Therefore, one might consider

$$((!selected, ?chair) : 3)$$

While this correctly counts the number of ways we can seat 3 people on 3 chairs, it is still not compatible with the first representation. Let's combine the tuples, and we can do so because they have no sets in common.

$$((?person : 3), ((!selected, ?chair) : 3))$$

Consider the following two configurations (observe that selected must follow the order given by the first choice):

$$((a, b, c), ((a, 1), (b, 2), (c, 3))$$

$$((b, c, a), ((b, 2), (c, 3), (a, 1))$$

The two configurations are equivalent and cannot be obtained by permutations of unordered choices. One way to fix this is by using the representation:

$$((!selected : 3), \{?chair : 3\})$$

which we can interpret by ignoring the chairs, i.e. regardless of our choice of chairs, the first person sits on chair 1, the second on chair 2, and the third on chair 3. In fact, we can easily show that in the following program $a \times b = c \times d$ if the tuple contains nonreusable or ordered sets other than s.

```
s = one {rep:n}
a = count {rep:n}
b = count {(?s,...):n}
s = one (rep:n)
c = count (rep:n)
d = count ((!s:n),{(...):n})
```

Observe that $a = (count(rep : n))/n!$ and $b = (n!(count((...) : n))/n!$. Also, $c = count(rep : n)$ and $d = (count((...) : n))/n!$.

A similar analysis shows that $a \times b$ is always the same as $c \times d$ in the following program:

```
s = one {rep:n}
a = count {rep:n}
b = count ((?s:n),...)
s = one (rep:n)
c = count (rep:n)
d = count ((!s:n),...)
```

The point of this section is that by using *one* we can connect the first and the second part in some way to prevent certain errors.

7.2 The *all* function

The statement $s = all\ rep$ creates a nonreusable set s of size equal to *count rep*. Conceptually, it creates a set of all possible choices given by the representation *rep*. The use of *all* can help in combining multiple representations into one. Here's our example revisited:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
rep = {?person:3}
choice = all rep
selected = one rep
count (?choice, {(?selected,?chair):3})
```

Could we have simply combined the representations without introducing a new set, like this?

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
rep = {?person:3}
selected = one rep
count ({?person:3}, {(?selected,?chair):3})
```

The answer is yes. But if we have decided for clarity to keep the same names for our sets, i.e. use person instead of selected, our program will look like this:

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
rep = {?person:3}
#change person to the set of 3 selected
person = one rep
count ({?person:3}, {(?person,?chair):3})
```

In this case, we are actually choosing 6 people from the set person, which now contains 3 elements; therefore, we exhaust the set and produce a count of 0. Again, we cannot combine representations if they have sets in common. One way around this (other than changing names), is the use of the *all* function to indicate a milestone and count up to a certain point.

```
person = nonreusable {a,b,c,d,e}
chair = nonreusable {1,2,3}
rep = {?person:3}
choice = all rep
#now size choice is equal to count rep
#change person to the set of 3 selected
person = one rep
count (?choice, {(?person,?chair):3})
```

We can apply the various program transformations on this example too. It is not hard to see that the following the general programs are equivalent:

```
rep = {?s:n}
t = one rep
choice = all rep
count (?choice, ((?t,...):n), ...)
```

```
rep = (?s:n)
t = one rep
choice = all rep
count (?choice, ((!t,...):n), ...)
```

```
count ((?s:n), ((...):n), ...)
```

Also, if $(?t, ...)$ has nonreusable or ordered sets other than t, the following three general programs are equivalent:

```
rep = {?s:n}
t = one rep
choice = all rep
count (?choice, {(?t,...):n}, ...)
```

```
rep = (?s:n)
t = one rep
choice = all rep
count (?choice, (!t:n), {(...):n}, ...)
```

```
count ((?s:n), {(...):n}, ...)
```

7.3 A committee with leader

We have 12 people, and we would like to form a committee of 4, one of them is a leader. In how many ways can we do that? One approach is to first choose 4 people, then given such a set, we can elect a leader among them.

```
person = nonreusable 12
rep = {?person:4}
choice = all rep
committee = one rep
count (?choice, ?committee)
```

But what if we decide to do it the other way. First elect a leader. Then select three among the remaining people. The problem here is that in the second part, the set that we are acting on is not the one that we have selected in the first part. Here's the attempt:

```
person = nonreusable 12
leader = {?person:1}
choice = all leader
remaining = nonreusable 11
count (?choice, {?remaining:3})
```

We had to create an arbitrary set called remaining to alleviate the problem. A better way to approach this scenario would be to choose the remaining set of people instead of choosing a leader. Both amount to the same outcome:

```
person = nonreusable 12
#11 remaining people also define the leader
#choosing 1 or 11 from 12 has the same count
rep = {?person:11}
choice = all rep
remaining = one rep
count (?choice, {?remaining:3})
```

Sometimes we need to act on both sets, the set of choice and its complement. In such a case, we must explicitly define both of them. The following section provides an example:

7.4 Coloring blocks

Assume we have 5 blocks, 3 light colors, and 3 dark colors. We want to color the blocks in such a way that 2 must be light and 3 must be dark. First, we can choose which blocks are light. Then we can worry about coloring them. But, we also need to color the remaining blocks. So we need to act on both sets, the set of choice, and its complement. Here's a program to do that.

```
block = nonreusable 5
lightcolor = reusable 3
darkcolor = reusable 2
rep1 = {?block:2}
rep2 = {?block:3}
lightblock = one rep1
darkblock = one rep2
#use one choice only not both
#because the choice of light
#determines also what is dark
choice = all rep1
count (?choice, {(?lightcolor, ?lightblock):2}, {(?darkcolor, ?darkblock):3})
```

Chapter 8

Pitfalls In Transformations

We will consider common pitfalls in making program transformations, in particular when moving replacing *any* "?" with *next* "!", and when changing sets from being nonreusable to ordered.

8.1 Legal transformations

These are the three types of general transformations we have encountered:

8.1.1 Moving from *any* to *next*

If s is a nonreusable set of size n, then the following two representations give the same count:

```
{(?s,...):n}
```

```
((!s,...):n)
```

The reason for this is that $(?s : n)$ will contribute $n!$ to the count, while $\{... : n\}$ will contribute an adjustment of (division by) $n!$ to account for the unordered choices. Therefore, since $(!s : n)$ contributes 1, the two representations give the same count. Observe that it is important that s is nonreusable and that it has size n.

8.1.2 Unordered/ordered split counts

The following three programs produce the same count:

```
rep = {?s:n}
t = one rep
choice = all rep
count (?choice, ((?t,...):n),...)
```

```
rep = (?s,n)
t = one rep
choice = all rep
count (?choice, ((!t,...):n),...)

count ((?s:n), ((...):n),...)
```

The reason for this is that in the first program $(?t : n)$ contributes $n!$, and in the second program $?choice$ contributes an extra $n!$ while $(!t : n)$ contributes 1. The third program follows from the second by elimination of "!".

For a similar reason, if $\{(?t, ...) : n\}$ contains nonreusable and/or ordered sets other than t, the following three programs produce the same count.

```
rep = {?s:n}
t = one rep
choice = all rep
count (?choice, {(?t,...):n},...)

rep = (?s,n)
t = one rep
choice = all rep
count (?choice, (!t:n), {(...):n},...)

count ((?s:n), {(...):n},...)
```

This is because $\{(...) : n\}$ would still contains nonreusable and/or ordered sets after the removal of t and, hence, still adjusts by a factor of $n!$.

Observe that in both cases, it is important that n is the same in $rep = \{?s : n\}$ and in the count expression.

We now present examples of pitfalls using these types of transformations.

8.2 Pitfall 1

Consider the problem of seating 5 people on 3 chairs.

```
person = nonreusable 5
chair = nonreusable 3
count {(?person, ?chair):3}
```

One might think about applying the first type of transformation and replacing $\{(?person, ?chair) : 3\}$ with $((!person, ?chair) : 3)$ as in the following program:

```
person = nonreusable 5
chair = nonreusable 3
count ((!person, ?chair):3)
```

This would be wrong because person does not have size 3. It has size 5. In fact, a quick examination of the new representation reveals that only the first 3 people will ever get to sit. There is no way we could seat the fourth or the fifth person. So the representation is not good. On the other hand, we could have done this (chair is reusable and has size 3)

```
person = nonreusable 5
chair = nonreusable 3
count ((?person, !chair):3)
```

8.3 Pitfall 2

Consider the problem of selecting 2 people out of 12 to form a committee, one of them is the chair. This can be done using a split count as follows:

```
person = nonreusable 12
rep = {?person:2}
committee = one rep
choice = all rep
count (?choice, ?committee)
```

One might think about applying the second type of transformation by changing the selection in rep to an ordered one.

```
person = nonreusable 12
rep = (?person:2)
committee = one rep
choice = all rep
count (?choice, !committee)
```

Upon a first glance, both programs produce the same count. However, applying this transformation is conceptually wrong. The reason is that committee has size two, but it is being used only once. In other words, a more explicit way of writing the original program would be the following:

```
person = nonreusable 12
rep = {?person:2}
committee = one rep
#committee has size 2
choice = all rep
count (?choice, (?committee:1))
#only one element is chosen from committee
```

We can exhibit a difference is the count, if we slightly change the problem to a committee of 3 one of them is chair. Now the following two programs do not produce the same count (try):

```
person = nonreusable 12
rep = {?person:3}
committee = one rep
choice = all rep
count (?choice, ?committee)

person = nonreusable 12
rep = (?person:3)
committee = one rep
choice = all rep
count (?choice, !committee)
```

We can also verify that this newly obtained representation is not good. First we replace $?choice$ with $(?person : 3)$, and we preserve the order when performing $!committee$. So one possible configuration is:

$$((a, b, c), a)$$

and another possible configuration is:

$$((a, c, b), a)$$

Obviously, these two configurations are equivalent as they define the same committee and chair, but cannot be obtained from one another by permutations of unordered choices.

8.4 Pitfall 3

Consider a word of length 7. We want to assign 3 of those letters vowels. In how many ways can we do that? Here's a solution given by first selecting the three letters to hold the vowels.

```
letter = nonreusable 7
vowel = reusable 6
rep = {?letter:3}
vowelletter = one rep
choice = all rep
count (?choice, {(?vowelletter, ?vowel):3})
```

Focusing on the second part of the count expression, we can identify that $\{(?vowelletter, ?vowel) : 3\}$ can be replaced by $((!vowelletter, ?vowel) : 3)$. This is the first type of transformation and is legitimate (we can verify that the count is preserved):

```
letter = nonreusable 7
vowel = reusable 6
rep = {?letter:3}
```

```
vowelletter = one rep
choice = all rep
count (?choice, ((!vowelletter, ?vowel):3))
```

But what if we decide to change the selection to ordered in the first part? This will make vowelletter an ordered set. Then everything that follows seems to be compatible since we are using "!" with vowelletter, and we have established that this part of the expression counts correctly.

```
letter = nonreusable 7
vowel = reusable 6
rep = (?letter:3)
vowelletter = one rep
choice = all rep
count (?choice, ((!vowelletter, ?vowel):3))
```

Obviously, this transformation changes the count, since choice is now counting an ordered selection. It is easy to see that we are overcounting because we increased the count for ?*choice* but kept the rest the same. The transformation simply does not conform to any of the three types. In fact, we can determine that the newly obtained representation is not good. Consider the configuration which assigns the first, second, and third letters, the vowels a, e, and o.

$$((1,2,3),((1,a),(2,e),(3,o)))$$

This configuration is equivalent to:

$$((2,3,1),((2,e),(3,o),(1,a)))$$

and the two cannot be obtained from one another by permuting unordered choices. Observe that in both cases, the order of !*vowelletter* followed the order given by the first choice (that's how it should be).

Coming back to the original program, one might try a transformation of the third type.

```
letter = nonreusable 7
vowel = reusable 6
rep = (?letter:3)
vowelletter = one rep
choice = all rep
count (?choice, (!vowelletter:3), {?vowel:3})
```

Now, this is wrong because after the removal of vowelletter from the set, only a nonreusable set remains. This will change the count because we no longer divide by 3!, as the elements are no longer necessarily distinct.

Chapter 9

Exclusive Representations

We will consider some examples of using the functions *one* and *all*. Most of these examples can be solved without the use of *one* and *all*, but we consider them for the sake of further illustrating these two concepts. More importantly, we will consider counting problems that require the use of the **addition rule** by solving subproblems and adding their counts. The representations corresponding to different subproblems must be exhaustive, i.e. together they generate all possible configurations, and **exclusive**, i.e. they cannot generate equivalent configurations.

9.1 Coloring blocks

Assume we have 4 blocks and 3 colors and we wish to color the blocks. In how many ways can we do that if only 2 colors must be used? One approach is to first select which two colors will be used; hence, creating a split count that will lead to the possible use of *one* and *all*. Once a set of two colors has been created using *one*, a second part of the program can act on this set to obtain the number of ways we can color the blocks.

```
color = nonreusable 3
block = nonreusable 4
rep = {?color:2}
#a set of all ways we can do this
choice = all rep
#one particular choice of two
selected = one rep
#make selected reusable, so colors can be reused
selected = makereusable selected
a=count (?choice, {(?block,?selected):4})
#since selected is now reusable, we can use the same color for
#all blocks, subtract the two possibilities of using a single color
a-2*size(choice)
```

The size of choice is $3 \times 2/2! = 3$. The count obtained for a is therefore $3(4 \times 2 \times 3 \times 2 \times 2 \times 2 \times 1 \times 2)/4! = 48$. After subtracting twice the size of choice, which is 6, we get 42.

The last subtraction is needed because of a problem that we have encountered before: on the one hand, we need the set to be reusable, but on the other hand, we do not want a single element to occupy all the choices. We have encountered this problem with the 3 cards and 2 suits example. We want the set of 2 suits to be reusable in order to assign suits to cards; however, we do not want to assign all cards the same suit. At the time, we figured out a solution by observing that the only way we could assign suits to cards is by assigning one of the suits to two cards, and the other suit to the last card. That's the only way. So by making the set of suits nonreusable, we can use the representation:

$$(?suit, \{?card : 2\}, ?suit, ?card)$$

Can we do the same here? The answer is yes, but we have to be careful. There are now multiple ways to dividing the two colors among 4 blocks. We could have one of the two colors assigned to three blocks, and the other color to the remaining block. We could also have one of the two colors assigned to two blocks, and the other color to the remaining two. These two categories are exclusive: the first category represents 3 blocks of the same color, and one different; and the second category represents 2 blocks of the same color, and another two blocks of a different color. Moreover, these two categories are exhaustive: there is no other way. Therefore, we can add the counts corresponding to these two categories. Let's redo the program (this time by keeping selected as a nonreusable set).

```
color = nonreusable 3
block = nonreusable 4
rep = {?color:2}
#a set of all ways we can do this
choice = all rep
#one particular choice of two
selected = one rep
a=count (?choice,{?block:3},?selected,?block,?selected)
b=count (?choice, {({?block:2},?selected):2})
#the two categories are exclusive, so we can add them
a+b
```

Make sure you understand why the representations make sense, e.g. ask the two golden questions.

Now what if we decide that the two selected colors are ordered? Consider the following variation (the second representation needs to be adjusted because "!" cannot be used within a { }. Observe that using

$$((\{?block : 2\}, !selected) : 2)$$

correctly counts the number of ways we can equally split the colors among the blocks, but is not compatible with the rest of the representation because choice is ordered. For instance the following two configurations will be equivalent:

$$((blue, red), (\{a, b\}, blue), (\{c, d\}, red))$$

$$((red, blue), (\{c, d\}, red), (\{a, b\}, blue))$$

We have seen such a scenario with people and chairs in the previous lecture. Here's a typical correct transformation by simply removing the "!" outside the { }.

```
color = nonreusable 3
block = nonreusable 4
rep = (?color:2)
#a set of all ways we can do this
choice = all rep
#one particular choice of two
selected = one rep
#now selected is an ordered set, we must use ! with it
a=count (?choice,{?block:3}, !selected, ?block, !selected)
b=count (?choice, !selected, !selected, {{?block:2}:2})
#the two categories are exclusive, so we can add them
a+b
```

Now, since being an ordered set, selected can be eliminated entirely from the program, we can obtain the following:

```
color = nonreusable 3
block = nonreusable 4
rep = (?color:2)
#a set of all ways we can do this
choice = all rep
a=count (?choice,{?block:3},?block)
b=count (?choice, {{?block:2}:2})
#the two categories are exclusive, so we can add them
a+b
```

This is basically saying that for every ordered choice of two colors, we need to count the number of ways can we split the blocks in two groups of 3 and 1, and of 2 and 2.

There is another way we could have solved the problem using "!" and three disjoint categories (representations). The idea is to first choose 2 colors, then pick which subset of the blocks is of the "first" (in some **fixed** order) color, whatever it may be. For instance, one may assume an alphabetical order on colors (and this order is irrelevant of our original choice of two colors because that choice is unordered).

```
color = nonreusable 3
block = nonreusable 4
rep = {?color:2}
#a set of all ways we can do this
choice = all rep
#one particular choice of two
selected = one rep
#we will use ! with selected to select the first color
a=count (?choice, ?block, !selected, {?block:3}, !selected)
b=count (?choice, {?block:2}, !selected, {?block:2}, !selected)
c=count (?choice, {?block:3}, !selected, ?block, !selected)
#the three categories are exclusive, so we can add them
a+b+c
```

Verify that the above representations are exhaustive and exclusive (fix an order on the colors, e.g. alphabetical). Observe that if $rep = (?color : 2)$ (i.e. ordered), the representations are no longer good. For instance, this can be checked by combining a representation with $(?color : 2)$ in one tuple, where $(?color : 2)$ replaces $?choice$. Observe that, since *selected* must be chosen in order using !, the following two configurations are equivalent and cannot be permuted (this time the order of selected is given by the original choice):

$$((blue, red), \{a, b\}, blue, \{c, d\}, red)$$

$$((red, blue), \{c, d\}, red, \{a, b\}, blue)$$

In addition, the first and last representations overlap (and account for both). For instance, the following are equivalent:

$$((blue, red), a, blue, \{b, c, d\}, red)$$

$$((red, blue), \{b, c, d\}, red, a, blue)$$

Coming back to our program, the use of "!" can be eliminated, to get:

```
color = nonreusable 3
block = nonreusable 4
rep = {?color:2}
#a set of all ways we can do this
choice = all rep
#one particular choice of two
selected = one rep
#we will use ! with selected to select the first color
a=count (?choice, ?block, {?block:3})
b=count (?choice, {?block:2}, {?block:2})
c=count (?choice, {?block:3}, ?block)
#the three categories are exclusive, so we can add them
a+b+c
```

This in turn can be simplified as follows:

```
color = nonreusable 3
block = nonreusable 4
rep = {?color:2}
#a set of all ways we can do this
choice = all rep
a=count (?choice, ?block)
b=count (?choice, {?block:2})
c=count (?choice, {?block:3})
#the three categories are exclusive, so we can add them
a+b+c
```

Finally, we could have avoided the split count that uses *one* and/or *all* by replacing the ?*choice* with an explicit choice of two colors:

```
color = nonreusable 3
block = nonreusable 4
a=count ({?color:2}, ?block)
b=count ({?color:2}, {?block:2})
c=count ({?color:2}, {?block:3})
#the three categories are exclusive, so we can add them
a+b+c
```

This tells us that the unordered choice of colors means that there is a specific (hidden) rule about how to assign the colors to the blocks. For instance, the color that occurs first alphabetically is the one used for the selected set of blocks (verify that the three categories are exclusive).

We could have also done this (based on our fourth implementation):

```
color = nonreusable 3
block = nonreusable 4
a=count ((?color:2), ?block)
b=count ((?color:2), {{?block:2}:2})
#the two categories are exclusive, so we can add them
a+b
```

Now the choices of colors are ordered, so choosing one block or 3 blocks is symmetric (so one of them is dropped).

9.2 Apples and Oranges

We have 3 apples and 3 oranges. In how many ways can we eat 4 fruits if the order of eating them is relevant? Here's one possible approach to solve this problem:

```
fruit = reusable{apple, orange}
count (?fruit:4)
```

The problem with the above program is that it allows unlimited use of apples and oranges. For instance, one possibility is to eat 4 apples, or 4 oranges. But we know that we can eat at most 3 of a kind. To fix the problem, we can subtract the two erroneous configurations.

```
fruit = reusable{apple, orange}
a=count (?fruit:4)
a-2
```

The answer is obviously $2^4 - 2 = 14$.

Another way to approach this is by fixing the number of apples (thus we also fix the number of oranges). For instance, let's say we are going to eat n apples (and thus $m = 4 - n$ oranges). This version of the problem looks like the anagram problem. We want to place n apples and m oranges in 4 positions. The following program will solve this version (where n and m are replaced by numbers)

```
pos = nonreusable 4
apple = identical 3
orange = identical 3
count ({(?pos, ?apple):n},{(?pos, ?orange):m})
```

Obviously, by eliminating identical sets, we end up with this version:

```
pos = nonreusable 4
count ({?pos:n},{?pos:m})
```

Now we can observe that by changing n, we obtain different solutions that are legitimate for the original problem. Solutions for different values of n are exclusive, so we can add their corresponding counts. Therefore, we can solve our original problem as follows:

```
pos = nonreusable 4
a = count ({?pos:0},{?pos:4})
b = count ({?pos:1},{?pos:3})
c = count ({?pos:2},{?pos:2})
d = count ({?pos:3},{?pos:1})
e = count ({?pos:4},{?pos:0})
a+b+c+d+e
```

For a given $0 < n < 4$ (and the corresponding m), the answer is $(4 \times 3 \times 2 \times 1)/(n!m!) = 24/(n!m!)$ (otherwise, it is 0 for $n = 4$ or $m = 4$ because the set of positions will be exhausted). Therefore, we get $0 + 24/(1!3!) + 24/(2!2!) + 24/(3!1!) + 0 = 14$.

What if the order of eating the fruits is not relevant? Then it's just a matter of how many apples we eat. So we have 3 possibilities. In fact, we can still adapt our first correct program by changing the tuple to a set.

```
fruit = reusable{apple, orange}
a=count {?fruit:4}
a-2
```

Obviously, the answer should be 3 and, therefore, $count\{?fuit : 3\}$ should be 5. We now revisit how to obtain the count manually. The formula based on the product rule and its adjustment given by $(2 \times 2 \times 2)/3!$ does not work here. This does not even give an integer for an answer. We have seen this before with the donut problem. The issue here is that $\{?fruit, ?fruit, ?fruit, ?fruit\}$ does not consist of **distinct** elements. This is because the set $fruit$ is reusable. This happens whenever the sets used within $\{rep : k\}$, where rep is some representation, do not include any nonreusable or ordered sets. In other words, all the sets are identical or reusable. Since elements are not distinct, dividing by $k!$ is not the correct adjustment, because the number of possible permutations is not $k!$ (it's less). The correct answer is C_k^{n+k-1}, where $n = count\ rep$ (but if some identical set in rep is exhausted, n should be 0), where:

$$C_k^n = \frac{n!}{k!(n-k)!}$$

In fact, we have been using this quantity without naming it. Observe that

$$C_k^n = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{k!}$$

which involves a multiplication of k terms and an adjustment by $k!$.

For our problem above, $n = count\ ?fruit$ which is 2 and $k = 4$. So we get $C_4^{2+4-1} = C_4^5 = 5$ as desired.

9.3 Greeting cards

We have 12 people and an unlimited supply of 3 kinds of greeting cards. For each person, we would like to send at least 1 and up to 3 greeting cards (but not the same card multiple times). In how many ways we can do that?

Each person can receive 1, 2, or 3 cards. These categories are exclusive. We can compute how many possible configurations of cards one person can receive. Then, we can think of these configurations are reusable, and assign one to each person.

```
card = nonreusable 3
a = count ?card
b = count {?card:2}
c = count {?card:3}
#consider a set of the different
#possibilities of sending cards
way = reusable (a+b+c)
person = nonreusable 12
count {(?person,?way):12}
```

Chapter 10

The Counting Algorithm

Given a representation, we can pragmatically obtain the count. We say that *Rep* is *distinct* iff it contains $?S$ or $!S$ where S is nonreusable or ordered.

$$distinct\ Rep = \begin{cases} true & Rep = ?S \text{ or } Rep = !S, \\ & S \text{ is nonreusable or ordered} \\ distinct\ Rep_1 & Rep = \{Rep_1 : k\}, k > 0 \\ \vee_i (distinct\ Rep_i) & Rep = (Rep_1, \ldots, Rep_k), k > 0 \\ false & \text{otherwise} \end{cases}$$

The count can be computed recursively using the following rules (assuming the representation is valid and has has no ! in { }).

- *count* $?S$ and *count* $!S$ are done as described in the table of Chapter 4 with the appropriate side effect.
- *count* $\{Rep : 0\}$ and *count* ( ) are 1 (empty choice).
- *count* $(Rep_1, \ldots, Rep_k)$ is $\prod_{i=1}^{k}$ *count* Rep_i.
- *count* $\{Rep : k\}$ can be divided into two cases. Let $n_i =$ *count* *Rep* for $i = 1 \ldots k$ (explicitly performed k times to produce the proper side effects of the set size).
  - case 1: *Rep* is distinct. In this case, *count* $\{Rep : k\}$ is $\prod_{i=1}^{k} n_i/i$, i.e. like the product of a tuple divided by $k!$ (because choices across the k branches are distinct).
  - case 2: *Rep* is not distinct. If $n = n_1 = \ldots = n_k$, *count* $\{Rep : k\}$ is C_k^{n+k-1} (this is unordered selection with replacement because choices across the k branches are not necessarily distinct); otherwise, $n_i = 0$ for some i and *count* $\{Rep : k\}$ is 0 (some identical set S has been exhausted, i.e. *Rep* contains c occurrences of $?S$ such that S is identical and $c > |S|$). Equivalently, *count* $\{Rep : k\}$ is C_k^{n+k-1}

where $n = \min_{i=1\ldots k} n_i$. If *Rep* has no occurrence of $?S$ where S is identical, then a single computation of $n = \mathit{count}\ \mathit{Rep}$ is enough since in this case it does not produce any side effects on set size.

$$\mathit{count}\ \mathit{Rep} = \begin{cases} \min[1, (|S| - u_S)^+] & \mathit{Rep} = ?S, S \text{ is identical } {}^* \\ |S| & \mathit{Rep} = ?S, S \text{ is reusable} \\ (|S| - u_S)^+ & \mathit{Rep} = ?S, S \text{ is nonreusable } {}^* \\ \min[1, (|S| - u_S)^+] & \mathit{Rep} = !S, S \text{ is nonreusable } {}^* \\ \min[1, (|S| - u_S)^+] & \mathit{Rep} = !S, S \text{ is ordered } {}^* \\ 1 & \mathit{Rep} = (\ ) \text{ or } \{\mathit{Rep}_1 : 0\} \\ \prod_{i=1}^{k} n_i & \mathit{Rep} = (\mathit{Rep}_1, \ldots, \mathit{Rep}_k), n_i = \mathit{count}\ \mathit{Rep}_i \\ \prod_{i=1}^{k} n_i / i & \mathit{Rep} = \{\mathit{Rep}_1 : k\}, \mathit{Rep} \text{ is distinct}, \\ & n_i = \mathit{count}\ \mathit{Rep}_1 \\ C_k^{n+k-1} & \mathit{Rep} = \{\mathit{Rep}_1 : k\}, \mathit{Rep} \text{ is not distinct}, \\ & n = \min_{i=1\ldots k} n_i, n_i = \mathit{count}\ \mathit{Rep}_1 \end{cases}$$

* side effect $u_S = u_S + 1 (u_S = 0$ initially for all sets)