#P: A Language for Counting

Saad Mneimneh, Alexey Nikolaev

1 About the Name

The name #P is chosen based on the following three facts:

- This is the name of a complexity class associated with counting the number of solutions
- the # symbols is often used to represent a number or a count
- P is the first letter of "Product", which is the operator involved in the Product Rule, the fundamental principle of counting used in #P

2 Sets and Tuples

A set is an unordered (unless otherwise specified) collection of elements of the same type, usually expressed using the following notation:

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

A tuple is an ordered collection of elements, not necessarily of the same type, expressed using the following notation:

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

Most of the times, we don't give too much attention to the actual elements themselves, rather, when it comes to sets for instance, we want to make sure that all elements have the same type. To enforce this, we have a special syntax when constructing representations of sets. This will be described later.

3 Counting with the Product Rule

The basic mechanism for counting in #P is based on the product rule:

Product Rule: if an activity consists of k stages, and each stage i can be carried out in α_i ways regardless of other stages, then the entire activity can be carried out in $\alpha_1\alpha_2\cdots\alpha_k$ ways.

Each stage will consist of a choice. So for a stage i, α_i will be the number of ways we can carry out that choice.

4 Choice

A choice consists of choosing an element from a set. There will be two modes of making that choice: *Any* and *Next*, denoted by ? and ! respectively. For instance, given a set S, $?S$ means choose *any* element from S, while $!S$ means choose the *next* available element from S. Therefore, with ! we assume an order on the elements of S. Given a choice, we care about the effect of ? and ! on counting the number of ways we can carry out that choice (the actual order does not affect the count). This effect will primarily depend on the kind of set. In addition, and depending on the kind of set, making a choice from set S can remove an element from S and, therefore, will decrease its "effective" size by 1. This is maintained through u_S (the usage of S) which is initially 0 (no elements taken). We have four kinds of sets (listed below).

	modifier allowed			
set kind	?	!	count	side effect
identical	yes	no	$\min[1, (\lvert S\rvert - u_S)^+]$	$u_S = u_S + 1$
reusable	yes	no	$\lvert S\rvert$	none
nonreusable	yes	yes	$(\lvert S\rvert - u_S)^+$ with ?	$u_S = u_S + 1$
			$\min[1, (\lvert S\rvert - u_S)^+]$ with !	$u_S = u_S + 1$
ordered	no	yes	$\min[1, (\lvert S\rvert - u_S)^+]$	$u_S = u_S + 1$

4.1 Identical

A set is called identical if all of its elements are the same. Making any choice always results in the same outcome. In addition, a choice removes an element from the set. Therefore, the count is always 1, unless the set has been exhausted, in which case the count will be 0.

4.2 Reusable

When a set is reusable, making a choice from this set does not remove any element. In this case, a subsequent choice can use the same element again. Therefore, the count is always equal to the size of the set.

4.3 Nonreusable

A nonreusable set is the typical one. Here, the count is equal to the effective size of the set when used with ? (choosing *Any*), and 1 when used with ! (choosing *Next*). A choice removes an element from a nonreusable set, so its effective size decreases by 1 with every choice (regardless whether it was performed with ? or !), and it can eventually reach 0.

4.4 Ordered

An ordered set is nonreusable but one that forces the choice to be done in order. The count is, therefore, always 1, unless the set has been exhausted, in which case the count will be 0.

Based on the above description, choices taken from nonreusable and ordered sets are always distinct. Sets can be created either explicitly (except for identical sets), or implicitly by simply specifying their kind and size.

5 Constructing the Count

In order to count, we need to construct the count by nesting our choices in sets and tuples to serve as a representation (not necessarily unique) of the objects we are counting. For instance, consider a set of people and a set of chairs (both nonreusable):

$$person = \{a, b, c\}$$

$$chair = \{1, 2, 3\}$$

To count the number of ways we can rank people, we can imagine the following activity: choose any person, then choose any person, then choose any person, in that order. The activity can be carried out in $3 \times 2 \times 1$ ways. Observe the decrease in the effective size of the set, an element is removed with every choice. This can be represented by a tuple:

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

The order here is desired because that's what a ranking is. On the other hand, if we wanted to simply choose two people, we can imagine the following activity: choose any person and choose any person, in no specific order. The activity can be carried out in $(3 \times 2)/2!$ ways (the division by 2! is due to the order of the two choices being irrelevant). This can be represented by a set:

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

Sets are constructed this way to ensure they are homegeneous, i.e. all elements have the same type. For instance, representations like $\{?person, ?chair\}$ are not allowed.

More sophisticated nesting of sets and tuples can be done. For instance, to assign people to chairs, we can imagine the following activity: choose a person, then choose a chair, in that order to make a pair, and do this three times where the order of the pairs is irrelevant. This activity can be carried out in $(3 \times 3) \times (2 \times 2) \times (1 \times 1)/3!$ ways (again, the division by 3! is due to the order of the three pairs being irrelevant). This can be represented as follows:

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

This set is also homogeneous, it contains three elements of the same type. The type of an element is described by the representation making it. For instance, $\{?person : 2\}$ contains two elements of type $person$. The above set contains three elements of type $(person, chair)$. The syntax for declaring sets will not allow for the possibility of creating sets of elements of different types. For tuples, we sometimes require that their elements have the same type. In this case, we call the tuple *homogeneous*.

The representation can be defined recursively:

$$Rep \ = ?S \mid !S \mid (Rep_1, \ldots, Rep_k) \mid \{Rep : k\}$$

and the equivalence of types as follows:

$$same\ Rep_1\ Rep_2 = \begin{cases} true & Rep_1 = ?S, Rep_2 = ?S \\ true & Rep_1 = !S, Rep_2 = !S \\ true & Rep_1 = ?S, Rep_2 = !S \\ true & Rep_1 = !S, Rep_2 = ?S \\ true & Rep_1 = \{Rep'_1 : 0\}, Rep_2 = \{Rep'_2 : 0\} \\ true & Rep_1 = (Rep'_1 : 0), Rep_2 = (Rep'_2 : 0) \\ true & Rep_1 = (Rep : 0), Rep_2 = () \\ true & Rep_1 = (), Rep_2 = (Rep : 0) \\ same\ Rep'_1\ Rep'_2 & Rep_1 = \{Rep'_1 : k\}, Rep_2 = \{Rep'_2 : k\}, k > 0 \\ \wedge_i(same\ Rep_{1i}\ Rep_{2i}) & Rep_1 = (Rep_{11}, \ldots, Rep_{1k}), \\ & Rep_2 = (Rep_{21}, \ldots, Rep_{2k}), k > 0 \\ false & \text{otherwise} \end{cases}$$

We can also imagine a different way for assigning people to chairs: choose the next chair, then choose any person, then choose the next chair, then choose any person, then choose the next chair, then choose any person, in that order. This activity can be carried out in $(1 \times 3) \times (1 \times 2) \times (1 \times 1)$ ways. Observe that the count for the choice of a chair is always 1 because we choose chairs in a specific order (regardless what that order is). This can be represented with the use of ! as follows:

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

or by making pairs more explicit:

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

The former is not a homogeneous tuple (it has elements of type $person$ and elements of type $chair$), the latter is (all its elements have type $(person, chair)$). Whatever your representation is, ! should never occur within { } because ! corresponds to an ordered choice and { } signifies an unordered one.

Since representations are not unique, how do we know that our representation is faithful to what we are counting? For instance, we could have constructed the following:

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

which would be wrong. Why? The reason is overcounting. For instance, the above representation can generate:

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

and can also generate:

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

While these seem to be different, they are "equivalent" because they both assign the same people to the same chairs. But this equivalence cannot be corrected because all the parts are ordered. Fortunately, to avoid such problem, we only have to verify two things:

- does the representation generate all possible configurations?
- do "equivalent" (generated) configurations correspond to permutations of the unordered choices?

If the answers to the above questions are Yes and Yes, then the representation is correct. For the above representation, we can easily argue that it can generate any valid assignment of people to chairs. However, some generated equivalent ones cannot be obtained by permutations, simply because the representation has no sets, just tuples. On the other hand both

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

and

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

are compliant. They can both generate all possible configurations (assignment of people to chairs). In addition, in the former case, equivalent configurations can be obtained by permutations. In the latter case, no generated configurations are equivalent because the chairs are always generated in a specific order (due to the use of !).

6 Counting: The Algorithm

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

$$\mathit{distinct}\ \mathit{Rep} = \begin{cases} \mathit{true} & \mathit{Rep} = ?S \text{ or } \mathit{Rep} = !S, \\ & S \text{ is nonreusable or ordered} \\ \mathit{distinct}\ \mathit{Rep}_1 & \mathit{Rep} = \{\mathit{Rep}_1 : k\}, k > 0 \\ \vee_i(\mathit{distinct}\ \mathit{Rep}_i) & \mathit{Rep} = (\mathit{Rep}_1, \ldots, \mathit{Rep}_k), k > 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

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

- *count* $?S$ and *count* $!S$ are done as described in the above table 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).
  - 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 $n =$ *count Rep* is enough since in this case it does not produce any side effects.

$$
\textit{count Rep} = \begin{cases}
\min[1, (|S| - u_S)^+] & Rep = ?S, S \text{ is identical }^* \\
|S| & Rep = ?S, S \text{ is reusable} \\
(|S| - u_S)^+ & Rep = ?S, S \text{ is nonreusable }^* \\
\min[1, (|S| - u_S)^+] & Rep = !S, S \text{ is nonreusable }^* \\
\min[1, (|S| - u_S)^+] & Rep = !S, S \text{ is ordered }^* \\
1 & Rep = (\ ) \text{ or } \{Rep_1 : 0\} \\
\prod_{i=1}^{k} n_i & Rep = (Rep_1, \ldots, Rep_k), n_i = \textit{count } Rep_i \\
\prod_{i=1}^{k} n_i/i & Rep = \{Rep_1 : k\}, Rep \text{ is distinct,} \\
 & n_i = \textit{count } Rep_1 \\
C_k^{n+k-1} & Rep = \{Rep_1 : k\}, Rep \text{ is not distinct,} \\
 & n = \min_{i=1\ldots k} n_i, n_i = \textit{count } Rep_1
\end{cases}
$$

* side effect $u_S = u_S + 1$

7 The premise

The premise of #P is that if you can construct correct representations for the objects you wish to count, then you have learned counting using the product rule. The following section provides examples to guide you through this process:

8 Examples

8.1 People and Chairs

8.1.1 Example 1

Count the number of ways we can we seat three people on three chairs.

Here's one way to count it:

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

The count will be $(3 \times 3) \times (2 \times 2) \times (1 \times 1)$ divided by 3!. Another way we could have done this is by explicitly naming the elements of our sets. By detault, those sets will be nonreusable:

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

8.1.2 Example 2

Count the number of ways we can seat five people on three chairs (only three get to sit).

The same strategy of Example 1 will work here too:

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

The count will be $(5 \times 3) \times (4 \times 2) \times (3 \times 1)$ divided by 3!. We could have also explicitly mentioned those who do not get to sit:

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

Make sure you understand why this works and verify that the count is the same as before. Here's another way for a change:

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

The count will be $(1 \times 5) \times (1 \times 4) \times (1 \times 3)$. Yet another way is to force the set of chairs to be ordered (which will force us to use !):

```
person = {a, b, c, d, e}
chair = ordered 3
rep = ((!chair, ?person):3)
count rep
```

Try to figure out why the following program is wrong (think about questions 1 and 2 described in Section 5).

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

8.1.3 Example 3

Count the number of ways we can seat five people on three chairs (all must sit).

Again, the same type of program should work for this case:

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

Obviously, the answer should be zero because the set of chairs will be exhausted after making the third choice from that set. Explicitly, the count will be $(5 \times 3) \times (4 \times 2) \times (3 \times 1) \times (2 \times 0) \times (1 \times 0)$ divided by 5!.

8.1.4 Example 4

Count the number of ways we can seat five people on three chairs (not all get to sit) if all the chairs are identical.

Now we have the chance to use an identical set:

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

The count will be $(5 \times 1) \times (4 \times 1) \times (3 \times 1)$ divided by 3!.

8.2 People and Gifts

8.2.1 Example 1

Count the number of ways we can give ten gifts to five people if all the gifts must go.

Obviously, someone has to receive multiple gifts. Here's one way to encode this observation using reusable sets.

```
gift = nonreusable 10
person = reusable 5
rep = {(?gift, ?person):10}
count rep
```

This time, the count will not reach zero because person is a reusable set. In other words, making a choice from person does not decrease the effective size of the set (see above table). The count will be $(10\times 5)\times(9\times 5)\times(8\times 5)\cdots\times(1\times 5)$ divided by 10!, which is 5^{10}.

8.2.2 Example 2

Count the number of ways we can give ten gifts to five people if all the gifts must go and they are identical, e.g. pennies.

Now our sets are identical and reusable, respectively:

```
gift = identical 10
person = reusable 5
rep = {(?gift, ?person):10}
count rep
```

The count will be C_k^{n+k-1}, where $n = 1 \times 5$ and $k = 10$ (see Section 6).

8.3 Making Teams

Count the number of ways we can make teams of two out of six people.

We need to divided people into groups of two:

```
person = nonreusable 6
rep = {{?person:2}:3}
count rep
```

Observe that we have only used unordered combinations. Make sure you understand why. The count will be $(6 \times 5)/2! \times (4 \times 3)/2! \times (2 \times 1)/2!$ divided by 3!. Here's a non-obvious way of counting the same thing:

```
person = nonreusable 6
rep = ((!person, ?person), (!person, ?person), (!person, ?person))
count rep
```

The count will be $(1 \times 5) \times (1 \times 3) \times (1 \times 1) = 15$ (you can verify that this is the same count in the previous example). While this may be tricky to understand, again there are only two questions you have to consider:

- does the representation generate all possible configurations?
- do "equivalent" (generated) configurations correspond to permutations of the unordered choices?

To answer the two questions, let's change the program slightly:

```
person = {a, b, c, d, e, f}
rep = ((!person, ?person), (!person, ?person), (!person, ?person))
count rep
```

Now, let's consider the teams:

$$((a, c), (d, b), (e, f))$$

Obviously, this configuration cannot be generated because after choosing (a, c), the representation says that we have to choose the next available person (using !), which cannot be d since b appears ahead of it in a fixed order. However, the configuration

$$((a, c), (b, d), (e, f))$$

is possible, which to us, is equivalent to the previous one. So that's good. In fact, given any configuration (team assignment), we can sort the people within each team, and then sort the teams by their first person. The equivalent configuration thus obtained can definitely be generated given the rules of the representation. So every configuration can be generated.

To answer the second question: no generated configurations are equivalent because the first person of every team is generated in a specific order. Try to figure out why $(?person, !person, ?person, !person, ?person, !person)$ is not a good representation for this problem.

8.4 Playing Cards

8.4.1 Example 1

Count the number of ways we can select J, Q, K from a deck of cards and end up with exactly two suits.

Here's a first attempt:

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

Two cards get the same suit, one card gets the other one. Notice that the order of choosing the first two cards is irrelevant. It might be more intuitive if we explicitly group our choices:

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

In both cases the count will be the same, $4 \times (3 \times 2)/2! \times 3 \times 1 = 36$.

Another way we could have done this is by first counting the number of ways we can choose two suits. Then incorporate this count into another choice involving the cards. For this, we can create an intermediate set consisting of all the possible configuration, i.e. choices of two suits, the function *all* does that.

```
suit = {diamond, heart, spade, club}
card = {J, Q, K}
rep1 = (?suit, ?suit)
allpairs = all rep1 #this is now a nonreusable set
                    #consisting of all 12 pairs of suits
rep2 = (?allpairs, {?card:2}, ?card)
count rep2
```

The count will be $12 \times (3 \times 2)/2! \times 1 = 36$.

Almost the same can be done using the *size* function, which gives the size of a set. In fact, *size* (*all rep*) is equivalent to *count rep*.

```
suit = {diamond, heart, spade, club}
card = {J, Q, K}
rep1 = (?suit, ?suit)
allpairs = all rep1 #this is now a nonreusable set
                    #consisting of all 12 pairs of suits
rep2 = ({?card:2}, ?card)
(size allpairs)*(count rep2)
```

Try to work out the count and verify that it is the same as above. We cab also make two sets, one for all the pairs, and another for the card choices:

```
suit = {diamond, heart, spade, club}
card = {J, Q, K}
rep1 = (?suit, ?suit)
allpairs = all rep1 #this is now a nonreusable set
                    #consisting of all 12 pairs of suits
rep2 = ({?card:2}, ?card)
cardchoices = all rep2 #this is now a nonreusable set
                       #of all 3 card choices
rep = (?allpairs, ?cardchoices)
count rep
```

Yet another way is to first count the number of ways we can choose two suits, then given such a choice, count the number of ways we can assign cards to suits. Then multiply the two counts. Analogous to *all*, the function *one rep* makes a set representing a single configuration (an element of *all rep*).

```
suit = {diamond, heart, spade, club}
card = {J, Q, K}
rep1 = {?suit:2}
pair = one rep1 #this makes a nonreusable set because rep1 is a set
                #the set contains two suits
rep2 = (?pair, ?pair, {?card:2}, ?card)
(count rep1)*(count rep2)
```

The count for rep1 is $(4\times3)/2! = 6$. The count for rep2 is $2\times1\times(3\times2)/2!\times1 = 6$. So the result is $6 \times 6 = 36$ as before.

A similar way to above can make rep1 ordered:

```
suit = {diamond, heart, spade, club}
card = {J, Q, K}
rep1 = (?suit, ?suit)
pair = one rep1 #this makes an ordered set because rep1 is a tuple
                #the set contains two suits
                #we are forced to use ! with ordered sets
rep2 = (!pair, !pair, {?card:2}, ?card)
(count rep1)*(count rep2)
```

The count for rep1 is $4 \times 3 = 12$. The count for rep2 is $1 \times 1 \times (3 \times 2)/2! \times 1 = 3$. So the result is $12 \times 3 = 36$ as before.

Since *one rep* creates a set that is an element of *all rep*, it will produce an error if *count rep* is zero (the set *all rep* is empty). Similarly, *one rep* will produce an error if *rep* is not distinct or is a tuple that is not homogeneous.

8.4.2 Example 2

Count the number of ways we can select from a deck of cards three aces with exactly two suits.

The answer is obviously 0, but we can work it out explicitly.

```
suit = {diamond, heart, spade, club}
ace = identical 3
rep1 = {?suit:2}
pair = one rep1 #this is now a nonreusable set of size 2
rep2 = {(?pair, ?ace):3}
(count rep1)*(count rep2)
```

Since we are making three choices from a nonreusable set of size 2, the count will reach 0.

8.4.3 Example 3

Count the number of ways we can choose J, J, K with exactly two suits (assume one deck of cards).

The important observation here is that one jack must be different than the king; otherwise, all three will have the same suit. Since jacks cannot have the same suit (one deck), then one jack must have the same suit as the king. Let's ignore that jack. The problem becomes that of assigning two suits to two cards, a jack and a king.

```
card = {J, K}
suit = {diamond, heart, spade, club}
rep = ((?suit, !card), (?suit, !card))
count rep
```

We essentially choose a suit for the first card, then a suit for the second card. The count will be $(4 \times 1) \times (3 \times 1)$.

We could have also done this:

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

Verify that the count remains the same.

Another way is to explicitly use a set of three card, like this:

```
card = ordered {K, J1, J2} #elements must be unique
suit = {diamond, heart, spade, club}
rep = ((?suit, !card, !card), (?suit, !card))
count rep
```

The set of cards is explicitly made an ordered set. In our mind, it is ordered like this: K, J1, J2. We choose a suit, and assign it to the first two cards, then choose another suit, and assign it to remaining card. There are other ways of doing this, try to come up with one on your own. The count is still as before $(4 \times 1 \times 1) \times (3 \times 1) = 12$.

8.4.4 Example 4

Count the number of ways we can choose J, J, K with exactly two suits (assume two decks of cards).

Let's drop one of the jacks because it can be simply assigned the suit that is not assigned to the king. This time, however, the remaining jack can have the same suit as the king.

```
card = {J, K}
suit = {diamond, heart, spade, club}
rep1 = {?suit:2}
pair = one rep1 # this is now nonreusable
pair = makereusable pair # now pair is reusable
rep2 = {(?pair, ?card):2}
(count rep1)*(count rep2)
```

The count for rep1 is 6. Since the set pair is reusable, the count for rep2 is $2 \times 2 \times 2 \times 1$ divided by 2!, which is 4. The result is $6 \times 4 = 24$.

8.5 Binary Words

Count the number of binary words we can make with ten bits.

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

The count will be $(10 \times 2) \times (9 \times 2) \times (8 \times 2) \cdots \times (1 \times 2)$ divided by 10!, which is 2^{10}.

We can also make the position implicit:

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

8.6 Anagrams

Count the number of words we can make from the letters of MATHEMATICS.

Each letter can be used a certain number of times:

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

The count will be $(1 \times 11) \times (1 \times 10)/2! \times (1 \times 9) \times (1 \times 8)/2! \times (1 \times 7) \times (1 \times 6)/2! \times (1 \times 5) \times (1 \times 4) \times (1 \times 3) \times (1 \times 2) \times (1 \times 1) = 11!/(2! \times 2! \times 2!)$.

Try another way in which the letters are implicit.

8.7 Greeting Cards

8.7.1 Example 1

You have 12 friends and 3 kinds of greeting cards. Each kind has 4 cards. Count the number of ways you can distribute the cards among your friends.

Each card can serve as a set:

```
friend = nonreusable 12
card1 = identical 4
card2 = identical 4
card3 = identical 4
rep = ({(?card1, ?friend):4}, {(?card2, ?friend):4}, {(?card3, ?friend):4})
count rep
```

The count will be $(1\times 12)\times(1\times 11)\times(1\times 10)\times(1\times 9)/4!\times(1\times 8)\times(1\times 7)\times (1\times 6)\times(1\times 5)/4!\times(1\times 4)\times(1\times 3)\times(1\times 2)\times(1\times 1)/4! = 12!/(4!\times 4!\times 4!)$.

8.7.2 Example 2

You have 12 friends and 3 kinds of greeting cards. Each kind has infinite supply. cards. Count the number of ways you can distribute the cards among your friends if each friend must receive exactly one card.

Since each card has an infinite supply, we can think of a reusable set of three elements:

```
friend = nonreusable 12
card = reusable 3 #3 kinds
rep = {(?friend, ?card):12}
count rep
```

The count will be $(12\times 3)\times(11\times 3)\times\cdots\times(1\times 3)$ divided by 12!, which is 3^{12}.

8.7.3 Example 3

You have 12 friends and 3 kinds of greeting cards. Each kind has infinite supply. cards. Count the number of ways you can distribute the cards among your friends if each friend must receive at least one card, but not the same kind twice.

Now we have several ways we could give cards to a friend: either 1 card, 2 cards, or 3 cards:

```
friend = nonreusable 12
card = nonreusable 3
rep1 = {?card:1} #ways to choose 1 card
rep2 = {?card:2} #ways to choose 2 cards
rep3 = {?card:3} #ways to choose 3 cards
```

Then we can do the following:

```
way = reusable ((count rep1) + (count rep2) + (count rep3))
rep = {(?friend, ?way):12}
count rep
```

The count for rep1 will be 3, for rep2 3, and for rep3 1, for a total of 7 ways to give a friend some cards. The count for rep will be $(12\times7)\times(11\times7)\times\cdots\times(1\times7)$ divided by 12!, which is 7^{12}.

8.8 The Empty Choice

Count the number of ways we can do nothing.

The count for an empty choice is 1.

```
count ()
```

or if *rep* is of any type, the following also represent the empty choice.

```
count {rep:0}
count (rep:0)
```

8.9 Apples and Oranges

8.9.1 Example 1

You have three apples and three oranges. You get to eat four fruits. In how many ways can you choose the fruits if the order of eating them is important?

We can break this into categories based on how many apples we eat:

```
apple = identical 3
orange = identical 3
pos = nonreusable 4
apple0 = ({(?apple, ?pos):0}, {(?orange, ?pos):4})
apple1 = ({(?apple, ?pos):1}, {(?orange, ?pos):3})
apple2 = ({(?apple, ?pos):2}, {(?orange, ?pos):2})
apple3 = ({(?apple, ?pos):3}, {(?orange, ?pos):1})
apple4 = ({(?apple, ?pos):4}, {(?orange, ?pos):0})
(count apple0) + (count apple1) + (count apple2) +
(count apple3) + (count apple4)
```

The count for apple0 will be $(1)\times(1\times4)\times(1\times3)\times(1\times2)\times(0\times1)/4! = 0$, for apple1 $(1\times4)/1!\times(1\times3)\times(1\times2)\times(1\times1)/3! = 4$, for apple2 $(1\times4)\times(1\times3)/2!\times(1\times2)\times(1\times1)/2! = 6$, for apple3 $(1\times4)\times(1\times3)\times(1\times2)/3!\times(1\times1)/1! = 4$, and for apple4 $(1\times4)\times(1\times3)\times(1\times2)\times(0\times1)/4!\times(1) = 0$. The total is 14.

8.9.2 Example 2

You have three apples and three oranges. You get to eat four fruits. In how many ways can you choose your fruits if the order of eating them is not important?

Here it just amounts to the number of apples we eat:

```
appleNum = {1,2,3}
count ?appleNum
```

The count is obviously 3.

8.10 Example 3

You have four apples and four oranges. You get to eat four fruits. In how many ways can you choose your fruits if the order of eating them is/(is not) important?

Since there is no restriction on how many apples and/or oranges we can pick within our choice of 4, we can think of a reusable set:

```
kind = reusable {apple, orange}
count (?kind:4) #order important

or

cout {?kind:4} #order not important
```

In the first case, the count will be $2 \times 2 \times 2 \times 2 = 2^4$ (find a relation to binary words). In the second case, the count will be C_k^{n+k-1} where $n = 2$ and $k = 4$, that's 5.

9 The #P Language at a Glance

9.1 Sets

identical n	makes an implicit set of n identical elements
reusable n	makes an implicit set of n reusable elements
nonreusable n	makes an implicit set of n nonreusable elements
ordered n	makes an implicit set of n ordered elements
$\{e_1, \ldots, e_n\}$	makes a nonreusable set, $e_i \neq e_j$
reusable $\{e_1, \ldots, e_n\}$	makes a reusable set, $e_i \neq e_j$
nonreusable $\{e_1, \ldots, e_n\}$	makes a nonreusable set, $e_i \neq e_j$
ordered $\{e_1, \ldots, e_n\}$	makes an ordered set, $e_i \neq e_j$
makereusable S	makes a reusable set with the elements of S, S cannot be identical
makenonreusable S	same as above but nonreusable
makeordered S	same as above but ordered
size S	gives the size of set S

9.2 Arithmetic

n	an integer
$+$	addition operator
$-$	subtraction operator
$*$	multiplication operator
$/$	division operator, gives error when result is not an integer

9.3 Comments

# followed by anything	a comment

9.4 Representations

$?S$	chooses any element from S, and removes it if S is not reusable, works with identical, reusable, and nonreusable
$!S$	chooses the next element from S and removes it, works with nonreusable and ordered
$\mathit{Rep}\ = ?S \mid\ !S$ $\mid (\mathit{Rep}_1, \ldots, \mathit{Rep}_k)$ $\mid \{\mathit{Rep} : k\}$	representation to construct a count, a ! choice does not work within a $\{\mathit{Rep} : k\}$
count *Rep*	gives the count
all *Rep*	makes an implicit nonreusable set of all possible configurations generated by *Rep*, elements of this set have type *Rep* and the size of this set is given by count *Rep*
one *Rep*	requires that $S =$ all *Rep* is not empty (i.e. count $\mathit{Rep} > 0$), and that *Rep* is distint if $k > 1$ (below), if *Rep* is $\{\mathit{Rep}_1 : k\}$, this makes an implicit non-reusable set of k elements of type Rep_1; if *Rep* is a homogeneous tuple $(\mathit{Rep}_1, \ldots, \mathit{Rep}_k)$, this makes an implicit ordered set of k elements of type Rep_1
generate *Rep* (not implemented yet)	generates a random configuration consistent with *Rep* by recursively replacing $?S$ and $!S$ with elements from S, for the base case, if the set is declared implicitly, elements take the name of the set followed by an index starting at 0 (no index when set is identical)