題目 E
Dynamic Programming

執行時間限制: 3 秒

古可魚語被發現後已經過了一年。可魚國的考古學家們雖然成功的將其翻譯為**現代可魚語**，但因為現代可魚語太複雜，無法直接使用電腦分析*魚法結構*。因此他們求助於你，希望你能將其翻譯成**簡化可魚語**。

以下分別定義現代可魚語、簡化可魚語的魚法。其中「$\langle name \rangle$」定義了魚法中「$name$」類型的 non-terminal（中間變數），它總是需要被其他定義換掉；「abc」這種字型的字則代表普通的字串；$[defn]^*$ 代表 $defn$ 這個 * 定義重複**零次或多次**（如 $[,\ \langle variable \rangle]^*$ 代表「」（空字串）、「, $\langle variable \rangle$」、「, $\langle variable \rangle$, $\langle variable \rangle$」、...）。

1. $\langle constant \rangle$ 是十進位整數，$\langle variable \rangle$ 是 `%`n（$n = 0, 1, 2, \ldots$）或任意 C 識別字（identifier）如 `%0`、`%514`、`npsc2014`，但**不能**是 `let`, `in`, `if`, `then`, `else`, `fn`。
2. 在現代可魚語只有 $\langle expr_0 \rangle$ 一種 non-terminal，而一個現代可魚語句子是一個符合 $\langle expr_0 \rangle$ 魚法的字串。$\langle expr_0 \rangle$ 可以被以下任何一種取代：
   - $\langle constant \rangle$、$\langle variable \rangle$、`(` $\langle expr_0 \rangle$ `)` 或 `fn (` $\langle variable \rangle$ `[,` $\langle variable \rangle$`]*` `) =>` $\langle expr_0 \rangle$
   - `let` $\langle variable \rangle$ `=` $\langle expr_0 \rangle$ `in` $\langle expr_0 \rangle$
   - `if` $\langle expr_0 \rangle$ `then` $\langle expr_0 \rangle$ `else` $\langle expr_0 \rangle$
   - $\langle expr_0 \rangle$ `(` $\langle expr_0 \rangle$ `[,` $\langle expr_0 \rangle$`]*` `)`

   以上的定義中，`fn (...) => ...` 代表「建立一個函數」，就像 javascript 的 `function(...){...}` 構造。詳細執行方法之後會以**可魚直譯器**解釋。
3. 簡化可魚語則有 $\langle expr_1 \rangle$ 和 $\langle value_1 \rangle$ 兩種 non-terminal。一個簡化可魚語句子是一個符合 $\langle expr_1 \rangle$ 魚法的字串，而 $\langle expr_1 \rangle$ 可以是
   - $\langle value_1 \rangle$
   - $\langle variable \rangle$ `(` $\langle value_1 \rangle$ `[,` $\langle value_1 \rangle$`]*` `)`
   - `let` $\langle variable \rangle$ `=` `fn (` $\langle variable \rangle$ `[,` $\langle variable \rangle$`]*` `) =>` $\langle expr_1 \rangle$ `in` $\langle expr_1 \rangle$
   - `let` $\langle variable \rangle$ `=` $\langle variable \rangle$ `(` $\langle value_1 \rangle$ `[,` $\langle value_1 \rangle$`]*` `)` `in` $\langle expr_1 \rangle$
   - `if` $\langle value_1 \rangle$ `then` $\langle expr_1 \rangle$ `else` $\langle expr_1 \rangle$

   $\langle value_1 \rangle$ 可以是 $\langle constant \rangle$、$\langle variable \rangle$ 或「`fn (` $\langle variable \rangle$ `[,` $\langle variable \rangle$`]*` `) =>` $\langle expr_1 \rangle$」

在現代可魚語中，$\langle expr_0 \rangle$ 的長度延伸到盡量遠處，所以 `fn (w)=> w(0)(1)` 等同 `(fn (w)=> (w(0)(1)))`。當然，簡化可魚語句子都是現代可魚語句子。翻譯過程中，對任意未知的集合 S^*，我們必須保證翻譯前與翻譯後的句子以**可魚直譯器**解釋的結果永遠一樣：

```
set insert(S_new, S) {
    shadow ← { (var,v) | ∀var,v,w s.t. (var,v) ∈ S ∧ (var,w) ∈ S_new }
    return (S \ shadow) ∪ S_new
}
// S, S_e, S* ⊆ variable × value
value translate(expr) { return translateHelper(expr,some fixed S*); }
value translateHelper(expr, S) {
    switch (expr) {
    case( c ): return c; // if c is a ⟨constant⟩
    case( var ): return v if (var,v) ∈ S // if var is a ⟨variable⟩
    case( (e) ): return translateHelper(e, S);
    case( fn (x1,x2,...,xk) => e ): return ⟨S,e,x1,x2,...,xk⟩;
    case( display(e) ): return printf(translateHelper(e, S));
    case( let var = e1 in e2 ):
        v ← translateHelper(e1, S);
        return translateHelper(e2, insert({(var,v)}, S));
    case( if e1 then e2 else e3 ):
        v ← translateHelper(e1, S);
        return translateHelper(v? e2 : e3, S);
    case( e0(e1,e2,...,ek) ):
        vk ← translateHelper(ek, S);
        ...
        v2 ← translateHelper(e2, S);
        v1 ← translateHelper(e1, S);
        ⟨Se,e,x1,x2,...,xk⟩ ← translateHelper(e0, S);
        return translateHelper(e, insert({(x1,v1),(x2,v2),...,(xk,vk)}, Se);
    }
}
```

為了確保翻譯的品質，考古學家們決定限制只能使用以下四種重寫規則，且規則 1、2、3 總是必須優先於規則 4 使用。

1. 一 *expr* 語句依其前後語句的條件（敘述於底下）及種類，在不影響解釋結果的前提下，可以做如下數種變換：

```
    ... // part A
    ... e0(e1,...,ei,...,ek) ... // part B
⇒ //  ( 0 ≤ i ≤ k 、 part B 不在「 fn (...) => 」中、前面沒有 let )
    ... // part A
    let %m = ei in ... e0(e1,...,%m,...,ek) ... // part B
```

```
    ... // part A
    ... (fn (x1,...,xk) => e0)(v1,...,vk) ... // part B
⇒ //  ( vi 都是 ⟨value1⟩ ，其餘條件同上)
    ... // part A
    let %m = fn (x1,...,xk) => e0 in ... %m(v1,...,vk) ... // part B
```

```
    ... // part A
    ... if e1 then e2 else e3 ... // part B
⇒ //  (其餘條件同上)
    ... // part A
    let %m = e1 in ... if %m then e2 else e3 ... // part B
```

「不在「fn (...) =>」中、前面沒有 let」意味以下都不可行，即使 S* 可能無法分辨

```
   (fn (x) => let z = h(w) in z)(0)
⇒ let %0 = let z = h(w) in z in (fn(x) => %0)(0)
```

```
   let k = fn (x) => x in f(k,g(z))
⇒ let %0 = g(z) in let k = fn(x) => x in f(k,%0)
```

2. 在不影響解釋結果的前提下，調整任意「let *var* = ⟨*expr*$_0$⟩ in」語句的**內外** 嵌套順序，如

```
    let x = let y = let z = w in z in y in x
⇒
    let x = let z = w in let y = z in y in x
```

3. 對於「let *var* = ⟨*constant*⟩ in *e*」或「let *var* = ⟨*variable*⟩ in *e*」語句，我們可以把 *e* 中**所有的** *var* 都替換成等號右邊的值，例如經過兩次變換後，

```
    let x = %1 in let w = h(x) in let z = 7 in mul(x,w,z)
⇒
    let w = h(%1) in mul(%1,w,7)
```

4. 若一「if e_1 then e_2 else e_3」語句不是在 ⟨$expr_0$⟩ 最尾端的位置，在不影響解釋結果的前提下，可以做如下的變換

```
    ... // part A
    ... if e1 then e2 else e3 ... // part B
⇒ //  (因為 if 不在最尾巴， part B 後面一定有 ... )
    ... // part A
    let %n = fn (%m) => ... %m ... // part B
                            ... in // part C
    if e1 then %n(e2) else %n(e3)
```

注意在這個變換中，「//part C」總是必須把所有外面的語句都抓進來。所以假如

```
    f( let flag = if x then y else z in g(flag) )
```

使用規則 4 重寫的話，結果是

```
    let %0 = fn (%1) => f( let flag = %1 in g(flag) ) in
    if x then %0(y) else %0(z)
```

而**不是**

```
    f( let %0 = fn (%1) => let flag = %1 in g(flag)
       in       if x then %0(y) else %0(z) )
```

■ 輸入說明

輸入的第一行有一個正整數 T，代表測試資料的筆數。

每一筆測試資料為一句現代可魚語句子，輸入皆以空白或換行隔開，且後面都有單獨一行「`-----`」。排版僅供參考。

- $T \leq 30$
- 保證輸入的句子都正確。
- 字串/符號個數不超過 21000 個。
- 變數名稱長度不超過 31 個字元，且不會重複，且一定不是 `%` 開頭。

■ 輸出說明

對於每筆測試資料請輸出翻譯後的簡化可魚語句子，並在其後輸出一行「`=====`」。翻譯過程中所新增的 $\%n$ 變數不可重複，且編號由上到下必須依序是 $0, 1, \ldots$。

遇到「`let` x `=` e_1 `in` e_2」語句時，必須在「`in`」之後換行；遇到「`fn (...) =>` e」語句時，若 e 是 `let ...` 或 `if ...`，則必須在「`=>`」之後換行，並於輸出 e 時內縮兩個空格；遇到「`if` e_1 `then` e_2 `else` e_3」時，必須在 `then` 後換行、`else` 前後換行，且輸出 e_2, e_3 時都內縮兩個空格。除此之外不可換行，且字串、符號之間都需空一個空白。

■ 範例輸入

```
2
let x =
  let z = 514 in
  let w = ( let v = 31 in add1 ( v , z , y ) ) in
  w in
( fn ( u ) => u ( u ( u ) ) ) ( fn ( k ) => k ) ( sub1 ( x ) )
-----
let x = pow ( 2 , n ) in
let v = if greater ( x , 8 ) then false else true in
if v then 7 else 29
-----
```

■ 範例輸出

```
let w = add1 ( 31 , 514 , y ) in
let %0 = sub1 ( w ) in
let %1 = fn ( u ) =>
  let %2 = u ( u ) in
  u ( %2 ) in
let %3 = %1 ( fn ( k ) => k ) in
%3 ( %0 )
=====
let x = pow ( 2 , n ) in
let %0 = greater ( x , 8 ) in
let %1 = fn ( %2 ) =>
  if %2 then
    7
  else
    29 in
if %0 then
  %1 ( false )
else
  %1 ( true )
=====
```