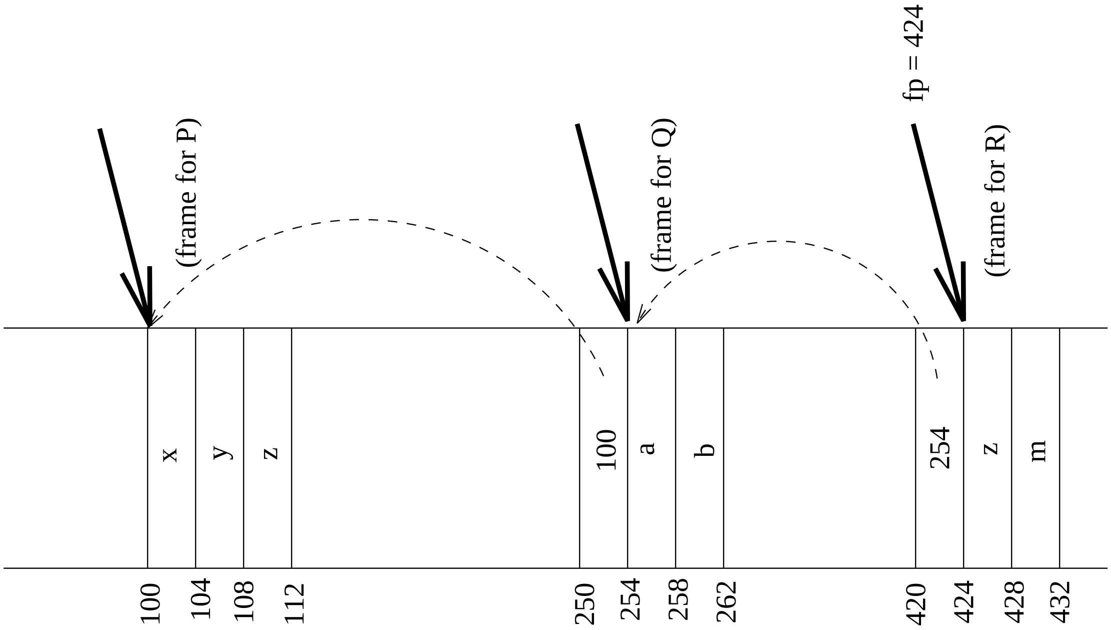
Static Links



Compilers for many high-level languages use a stack for runtime storage management. In the simplest case, every procedure invocation is allocated a portion of the stack called a *frame*, which holds the local variables defined by the procedure. In more advanced languages where procedures may be nested, each frame has a pointer, called a *static link*, which points to the frame of the parent procedure. The static link is used to access variables defined in outer scopes. For the purposes of this problem, we assume that the static link is always stored at offset `-4` of the frame pointer, and that the local variables are stored at offsets 0, 4, 8, etc.

For example, consider the situation depicted in the figure. A procedure P defines three variables x, y, and z. Within P, a procedure Q defines two variables a and b. As the stack organization shows, Q may access the variables defined in P by following the static link (the dotted line). Within Q, another procedure R defines two more variables z and m. If procedure R is currently active, a global variable `fp` points to its frame. The variables local to R are stored at addresses `(fp+0)` and `(fp+4)`. To access these variables we write `(*(fp+0))` and `(*(fp+4))` respectively. The variable a can be accessed by following the static link to reach the frame for Q and then using offset 0. We express this computation as `(*((*(fp-4))+0))`. The variable y can be accessed by following two static links, and then using offset 4. We express this computation as `(*((*((*(fp-4))-4))+4))`. The variable z is procedure P cannot be accessed from R since the declaration of z in R shadows the one in P.

You are to implement the part of the compiler that generates intermediate code for accessing variables. The input will be of the form:

```
3
x y z
a b
```

```
z m
z m a y
```

The first line gives the number of nested procedure declarations that follow. The corresponding number of following lines list the variables declared at each level. The final line gives you the variables for which you should generate code. In this case, your output should be:

```
(* (fp + 0))
(* (fp + 4))
(* ((* (fp - 4)) + 0))
(* ((* ((* (fp - 4)) - 4)) + 4))
```