1. listdates(StudentID, L) :- students(StudentID, _, CList),
    exams(CList, [], L).
listdates(_, []).
exams([], L, L).
exams([H|T], L1, L2) :- exams2(H, L),
    append(L1, L, L3),
    exams(T, L3, L2).
exams2(Course, L) :- examdates(Course, L).
exams2(_, []).
append([], L, L).
append([H|T], L2, [H|L3]) :- append(T, L2, L3).

2. (DEFINE (Suitable-Rooms CId)
    (Suitable-Rooms1 Rooms (Count Students CId)))

  (DEFINE (Suitable-Rooms1 Rooms Number)
    (COND ((NULL? Rooms) '())
      ((>= (CADDAR Rooms) Number) (CONS (CAAR Rooms)
          (Suitable-Rooms1 (CDR Rooms) Number)))
      (ELSE (Suitable-Rooms1 (CDR Rooms) Number))))

  (DEFINE (Count Students CId)
    (IF (NULL? Students) 0
      (+ (Process (CAR Students) CId) (Count (CDR Students) CId))))

  (DEFINE (Process Student CId)
    (IF (= (CADR Student) 1)
      (IF (Exists (CADDR Student) CId) 1 0)
      0))

  (DEFINE (Exists Courses CId)
    (COND ((NULL? Courses) #F)
      ((EQ? (CAR Courses) CId) #T)
      (ELSE (Exists (CDR Courses) CId))))

3. Main starts execution

   ARI of
   Main
   display
   0

   For each static depth, there is only one entry in the display.
   When a subprogram begins execution, it is easy to set the
   corresponding entry in the display: just set this entry to SP.
   Here, display[0] is set to SP before ARI of Main is created, then SP is
   incremented to the top of the ARI.
   display[0] points to ARI of Main, because static depth of Main is 0.
Main calls Sub1

- ARI of Sub1
- ARI of Main

display[1] points to ARI of Sub1, because static depth of Sub1 is 1.

As we can see, the contents of the display serve the function of static links; so there is no static link field in ARIs.

Sub1 calls Sub4

- ARI of Sub4
- ARI of Sub1
- ARI of Main

display[2] points to ARI of Sub4, because static depth of Sub4 is 2.

Sub4 calls Sub2

- ARI of Sub2
- ARI of Sub4
- ARI of Sub1
- ARI of Main

Static depth of Sub2 is 2, so display[2] must point to ARI of Sub2. When display[2] is changed to ARI of Sub2, its old content (which points to ARI of Sub4) must be saved. Because, when Sub2 terminates, we must restore display[2] to point to ARI of Sub4. The best place to save it is ARI of Sub2 (currently executing subprogram).

Now Sub4 is not in the display. This is the desired effect. Static ancestors of Sub2 are Sub1 and Main. In fact, only these are in the display (i.e. only these are visible to Sub2).

If there is a nonlocal reference in Sub2, we can find the variable directly. If the static depth of the declaration is 0, we go to display[0]. If the static depth of the declaration is 1, we go to display[1].

Sub2 terminates

- ARI of Sub4
- ARI of Sub1
- ARI of Main

display[2] is restored, i.e. its old content saved in ARI of Sub2 is assigned to display[2] before ARI of Sub2 is deleted from stack.

Sub4 calls Sub5

- ARI of Sub5
- ARI of Sub4
- ARI of Sub1
- ARI of Main

display[3] points to ARI of Sub5, because static depth of Sub5 is 3.
Sub5 calls Sub2

ARI of Sub2
ARI of Sub5
ARI of Sub4
ARI of Sub1
ARI of Main

display[2] points to ARI of Sub2, because static depth of Sub2 is 2.

display[3] remains pointing to Sub5, but it has no effect. Because only the display entries below that of the currently executing subprogram (in this case, display[2], display[1] and display[0]) are in effect. Because these are the static ancestors of the currently executing subprogram.

Old value of display[2] is saved in ARI of Sub2.

Sub2 calls Sub3

ARI of Sub3
ARI of Sub2
ARI of Sub5
ARI of Sub4
ARI of Sub1
ARI of Main

display[3] points to ARI of Sub3, because static depth of Sub3 is 3.

Old value of display[3] is saved in ARI of Sub3.

There are no pointers from display to Sub4 and Sub5, so they are not visible (i.e. they are not static ancestors of Sub3).

When a nonlocal variable is referenced in a statement in a subprogram, the static depth of the subprogram containing the declaration of this variable is found by the compiler. Then the compiler generates the necessary machine code to access “display[static depth of the subprogram containing the declaration of the variable] + offset within the ARI”. Thus, the variable can be accessed directly.

4.

<table>
<thead>
<tr>
<th>z</th>
<th>Sub3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>dynamic link</td>
<td></td>
</tr>
<tr>
<td>ret adr. (to Sub2)</td>
<td></td>
</tr>
</tbody>
</table>

There is no static link field ARIs, because the only purpose of static link is for nonlocal access in statically-scoped languages.

The differences from the static chain method:

i) As in the static chain method, the compiler cannot determine how many links must be followed for a nonlocal reference during execution. Because the compiler cannot know the calling sequence during execution.

   e.g. If Main calls Sub3, then for a reference to u in Sub3, one link must be followed. But, if Main calls Sub1 and Sub1 calls Sub3, then for the same reference, two links must be followed.

   So, there must be a search within the ARI after each link is followed.

ii) Within the ARIs, the names of local variables (in addition to their values/addresses) must be stored. Because we search for the names of variables. As in the static chain method, the offset logic cannot be used.

   e.g. If Sub1 calls Sub3 and Sub3 refers to w, then w will be found at offset 3 (within ARI of Sub1). But, if Sub2 calls Sub3 and Sub3 refers to w, then w will be found at offset 2 (within ARI of Sub2).

When a variable is referenced in a statement in a subprogram, the dynamic chain is followed to find it in one of the ARIs in the stack. At each ARI we reach, we search for the variable.