CSE 340 Spring 2011

Access links

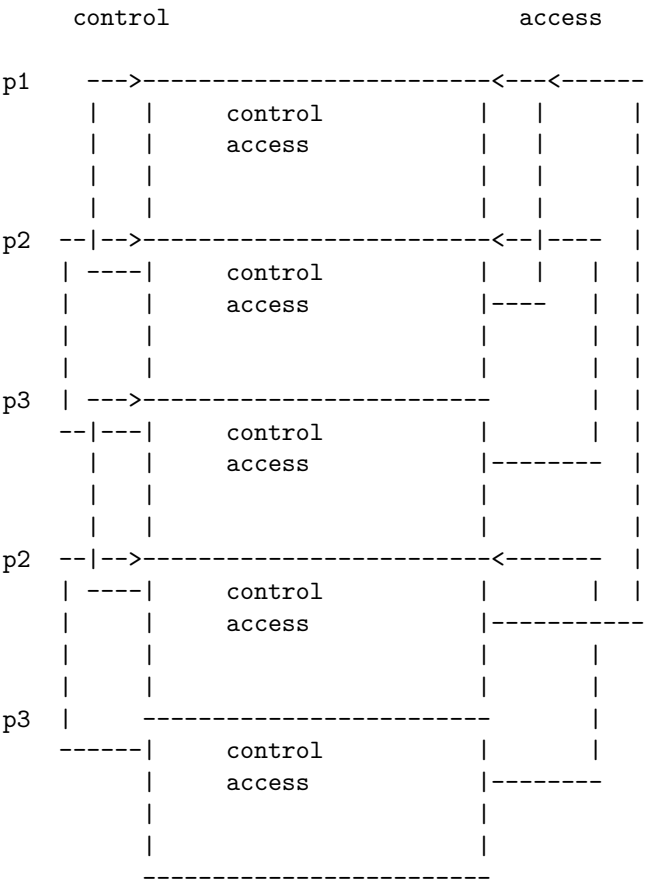
Warning! these notes are not meant to replace your notes from class. They are not meant to be exhaustive. If you see a topic that is mentioned here and not elaborated upon, you should go back to your class notes or the book for details. Also, if a topic is not here, it does not mean that I did not cover it.

1 Following the access link

In a language that supports nested procedures, we need a way to find at runtime the location of non-local variables that are defined in a enclosing procedure. For example:

```
 proc p1

      x : integer;

      proc p2

           proc p3


           begin // p3

                x = x+1;
                p2
           end // p3

      begin // p2
           p3;
      end // p2

begin // p1

   p2;
end // p1
```

The stack after the second call to `p3` looks like this:



As you can see in the figure, the access link points to the most recent instance of the defining environment. The control link points to the caller's environment.

The defining environment is determined statically for static scoping (see notes 4).

Now, we explain how `x` is accessed. To access `x` at runtime, the compiler needs to generate the code (at compile time naturally) that at runtime will access `x`. At compile time, the compiler know the following:

1. The offset of `x` in the activation record of `p1`. We call this offset offset$_x$.
2. the fact that `x` is declared in a scope that is two scopes (function scopes) removed from where it is referenced. In otehr words, the compiler know that the scope where `x` is declared encloses a function scope (`p2`) that encloses the scope (`p3`) where `x` is used.
3. the fixed offset $\texttt{offset}_{AL}$ where the access link is stored in every activation record

At runtime, the location of `x` is found by starting in `p3` and then following the access links twice. I explain how that can be done. In the code that follows I use `mem[p]` where `p` is a pointer as the content of the location pointed to by `p`. This is the same as `*p`.

We get the code for that in an incremental fashion:

- address of $\text{AL}_{p3} = \text{fp}_{p3} + \texttt{offset}_{AL}$
- value of AL_{p3} = frame pointer of defining environment of `p3` = frame pointer of most recent activation of `p2` = $\texttt{mem}[\texttt{fp}_{p3} + \texttt{offset}_{AL}]$
- address of access link of most recent activation of `p2` = $\texttt{mem}[\texttt{fp}_{p3}+\texttt{offset}_{AL}]+\texttt{offset}_{AL}$
- frame pointer of defining environment for most recent activation of `p2` = $\texttt{mem}[\texttt{mem}[\texttt{fp}_{p3} + \texttt{offset}_{AL}] + \texttt{offset}_{AL}]$
- address of `x` = $\texttt{mem}[\texttt{mem}[\texttt{fp}_{p3} + \texttt{offset}_{AL}] + \texttt{offset}_{AL}] + \texttt{offset}_{x}$

In general, if we want to follow n access links and then access `x` at offset $\texttt{offset}_x$, the code is

a = fp
for i = 1 **to** n
    $a = \texttt{mem}[a + \texttt{offset}_{AL}]$
address of `x` = $a + \texttt{offset}_x$

In the code, I assume `fp` is the environment pointer of the environment where the reference to `x` occurs.

We can write the same code using the dereference * as follows:

a = fp
for i = 1 **to** n
    $a = *(a + \texttt{offset}_{AL})$
address of `x` = $a + \texttt{offset}_x$

Note: In class today, I used both * and `mem` to dereference a pointer. This is redundant because they are equivalent.