Issues in implementation

Amer Diwan

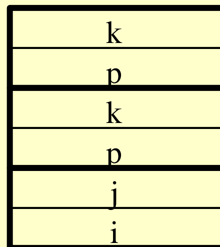
What we already know

- Each procedure puts its local variables and parameters in its activation record
- At each call the activation record for the callee is put on the stack
- At each return the activation record of the returning function is popped off the stack

An example

```
main() {
 int i = ...; int j = ...;
 foo(i)
}
```

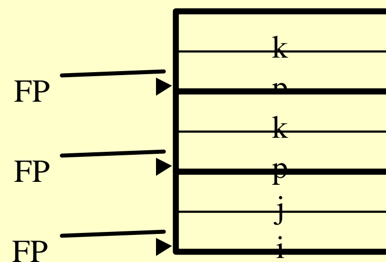
```
f(int p) {
  int k;
  if (p>10) {
    foo(p-1)
  }
```



How does a procedure access its variables?

```
main() {
 int i = ...;
 int j = ...;
 foo(i /* FP + 0 */)
}
```

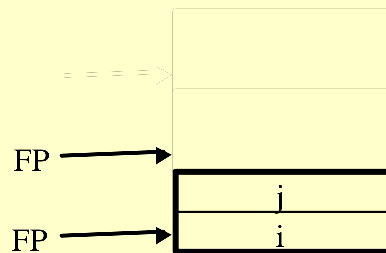
```
f(int p) {
  int k /*FP+1*/;
  if (p/*FP+0*/>10) {
    foo(p/*FP+0*/-1)
  }
```



What happens on a return?

```
main() {
 int i = ...;
 int j = ...;
 foo(i /* FP + 0 */)
}
```

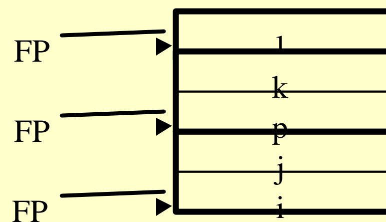
```
f(int p) {
  int k /*FP+1*/;
  if (p/*FP+0*/>10) {
    foo(p/*FP+0*/-1)
  }
```



Nesting scoping

```
main() {
 int i = ...;
 int j = ...;
 foo(i /* FP + 0 */)
}
```

```
f(int p) {
 int k /*FP+1*/;
 g() {
  int l;
  l /*FP+0*/ =
   k /*???*/ + 5;
 }
 k/*FP+1*/ =
   p/*FP+0*/ + 1;
 g();
}
```



The problem

- **g** needs access to a variable, **k**, in its outer scope
  - But **k** is not a global variable
  - **k** belongs to another procedure (and thus **g** doesn't even know where it is on the stack)

Environments

- An environment for a procedure contains all the names that a code needs

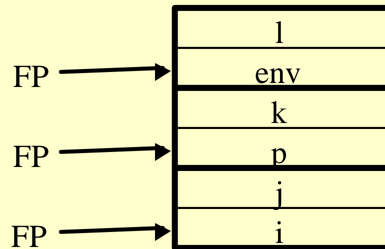
```
f(int p) {
 int k;
 g() {
  int j;
  l = k + 5;
 }
 k = p + 1;
 g(); }
```

What is the environment of procedure g?

(k, ...), (p, ...), (f, ...), ...

Solution

```
main() {
 int i = ...;
 int j = ...;
 foo(i /* FP + 0 */)
}
```




```
f(int p) {
 int k /*FP+1*/;
 g() {
  int j;
  l /*FP+0*/ =
   k /*env(k)*/ + 5;
 }
 k/*FP+1*/ =
   p/*FP+0*/ + 1;
 g();
}
```

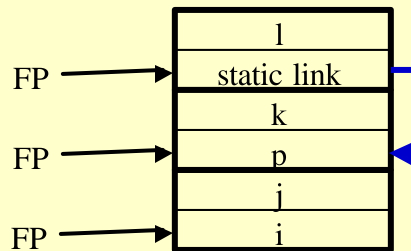
So, what goes in an environment?

- In functional languages (when there aren't any assignments), an environment can be a table:
  - [(name, value), (name, value), ...]
- Does this work for programs with mutation?

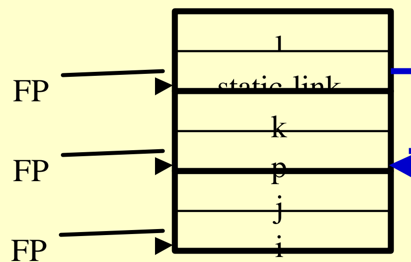
An implementation of environments

```
main() {
 int i = ...;
 int j = ...;
 foo(i /* FP + 0 */)
}
```

```
f(int p) {
 int k /*FP+1*/;
 g() {
  int j;
  l /*FP+0*/ =
   k /*env(k)*/ + 5;
 }
 k/*FP+1*/ =
   p/*FP+0*/ + 1;
 g();
}
```



How to use the “environment”

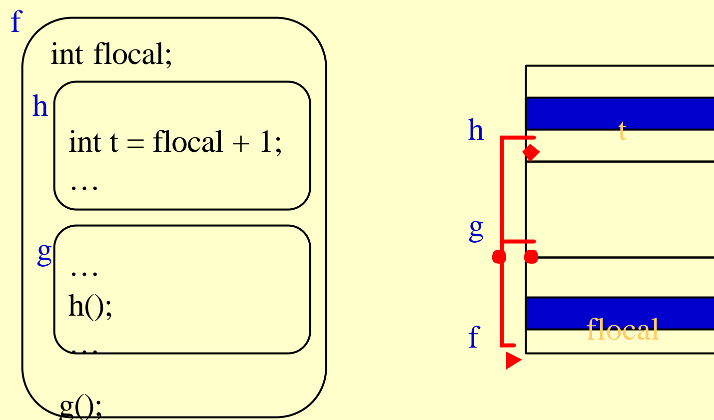


env(k) => static_link+1

What is a static link?

- Points to the activation record of its immediately enclosing procedure
  - When **`f`** calls **`g`**, it puts a pointer to **`f`**'s activation record into the static link slot in **`g`**'s activation record

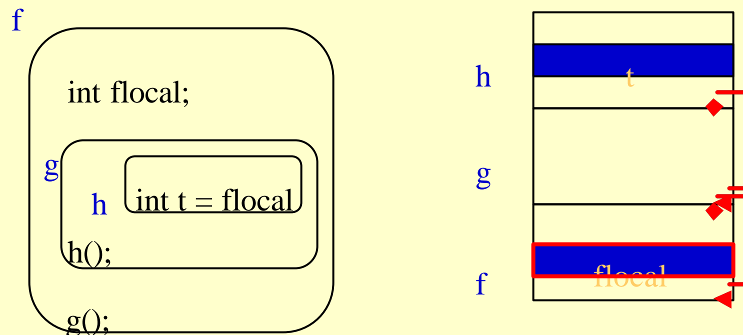
Another example



Accessing non-local variables

- To access a non-local variable (say flocal) declared in the immediately enclosing scope:
  - *(static_link + offset_of_flocal)
- To access a non-local variable declared n scopes away:
  - follow static links "n" times

An example



Passing functions as pointers

Assume that C has nested functions…

```
void do_something(i, l) {
      int n;
      int incrn(int elem) {
            return elem + n;
      }
      n = i;
      map(l, incrn);
}
```

```
int_list *map(lst, fcn) {
      if (lst == NULL) return NULL
      list *rest = map(list->tail, fcn);
      return cons(fcn(list->head), rest);
}
```

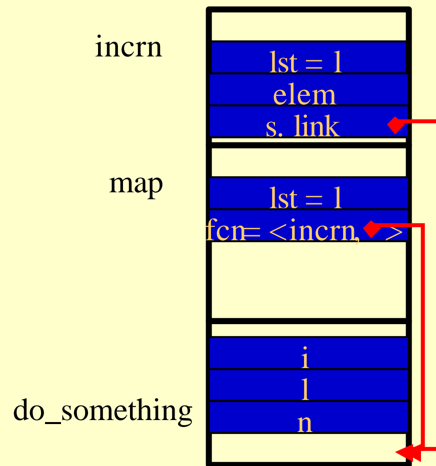
How will incrn access n?

Closures

- A <code, environment> pair
- code can get to variables in outer scopes through the environment
- Example of environment: the static link

When you need to pass a function or return a function, pass a closure

Example with closure



What's in a name?

- The closure of a function is the closed form of the function
  - no free variables
  - to access previously free variables, must go through the environment component of closure

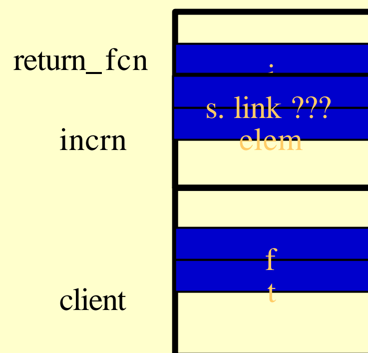
Escaping functions

```
void client() {
      f = return_fcn(10);
      t = f(5)
}
```

```
return_fcn(int i) {
int n;
int incrn(int elem) {
      return elem+n;
}
n = i;
return incrn;
```

What's the problem here?

Example continued



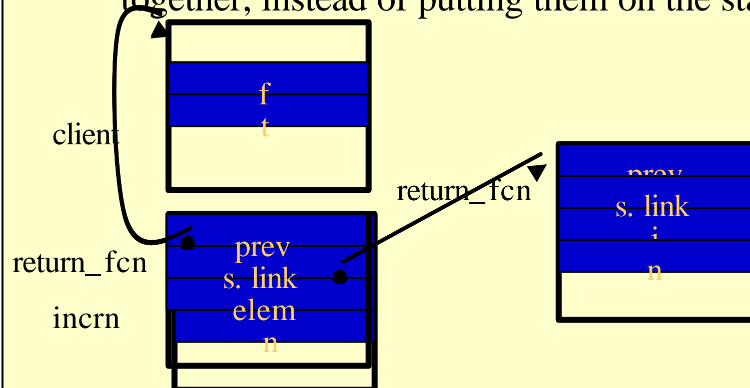
We would like incrn's static link to point to return_fcn's activation record, but that has been popped off the stack!

Solutions

- Don't have nested functions: C/C++/Java
- Have nested functions but don't allow them to escape: Modula-3, Pascal
- Keep activation records around even after they have been popped off (if needed): SML

Keeping activation records around

- Put activation records on the heap and link them together, instead of putting them on the stack



Pros and cons of closures

- Expressiveness
  - Ability to pass functions around is useful (as is evident from project 2!)
- Simplicity
  - If language supports it fully, it is more systematic and easier to keep straight
- Ease of implementation
  - Efficient implementation can be hard
- Efficiency
  - May have significant performance degradation: not dissimilar from virtual functions
  - It is a ripe area of research!

Next topic: Exception handling

- Readings: Section 8.5