Writinga

Postgres allows you to write queries of the type

```
retrieve (emp.all) where overpaid(emp)
```

where emp is a instance (or record) in the emp class (or table). Because of this, Postgres has a notion of the "current instance" which is being examined at a particular time when processing a query. Your function will not be passed a instance - rather it can make calls to the library function "GetAttribute" which will return requested attributes out of the current instance.

In the query above, a reasonable "overpaid" function would be:

```
bool
overpaid()  /* note that there are no arguments */

{
   extern Datum GetAttribute();
   short age, salary, performance;


   salary = (short) GetAttribute("salary");
   age = (short) GetAttribute("age");
   performance = (short) GetAttribute("performance");

   return (salary > (age * performance));
}
```

GetAttribute is the Postgres system function that returns attributes out of the current instance. It has one argument which is the name of the desired attribute, and its return type is a type "Datum" which is defined as being large enough to hold pointer values and all other types - currently it is defined as a "long". GetAttribute will align data properly so you can cast its return value into the desired form. So if you have an attribute "name" which is of the Postquel type "char16", the GetAttribute call would look like:

```
char *str;

str = (char *) GetAttribute("name")
```

Now suppose we have a variable length array of longs called "stats", that a more sophisticated "overpaid2()" function may be using. All variable length attributes in Postgres are returned in the following structure:

```
struct
{
   long len;
   unsigned char string[len];
}
VAR_LEN_ATTR;
```

so for our "stats" array, we do the following:

```
overpaid2()

{
   long *stats, nitems, i;
   VAR_LEN_ATTR *retval;

   retval = (VAR_LEN_ATTR *) GetAttribute("stats");
```

```
        stats = (long *) retval->string;
        nitems = retval->len / sizeof(long);

        for (i = 0; i < nitems; i++)
        {
            process the stats array...
        }
    }
```

Functions can have "normal" arguments as well as access to instances, so the following is quite legal (assuming the function overpaid3 takes these arguments)

```
retrieve (emp.all) where overpaid3(emp, emp.age)
```

In queries, the instance argument must be the first argument.

To let Postgres know about the first "overpaid" function, just do the following:

```
define C function overpaid(file = "/usr/postgres/tutorial/overpaid.o",
                           returntype = bool) arg is (RELATION)
```

The special flag "relation" in the argument list tells Postgres that this function will be processing a instance.

To define the second overpaid function (the one with a non-instance argument), run the following query:

```
define function overpaid (language = C,
    file = "/usr/postgres/tutorial/circle.o",
    returntype = bool) arg is (RELATION, uint1)
```

where emp.age is defined as an Postgres "uint1" type (one byte unsigned int).