Diamondback

```
int x;
int foo() { return 4;}
int foo2() {
    int x;
    int foo3() { return 8; }
    int x = 5;
    int x = if (y) { ... } ;
```

```
let x = def f() {....}
```

```
<program> ::= <declList> <expr>
            | <expr>

<declList> ::= <declaration>
             | <declaration> <declList>

<declaration> ::= def <identifier> (<paramList>) <expr> end
                | def <identifier> ( ) <expr> end
<paramList> ::= <identifier>
              | <identifier> , <paramList>

<expr> ::= .....
         | <identifier> (<exprList>)
```

```
def f(x)
    x+1
end
```

```
4
---------
def f(x)
    x+1
end
4
-------
def f(x)
    x+1
end
f(5)
```

```
def f(x)
    f(x)
end
f(0)
-------------
def f(x)
    g(x)
end
def g(x)
    f(x)
end
f(0)
```

```
type expr =
    ....
  | EInt of int
  | EAppl of string * expr list

type declaration =
  | EFunction of string * string list * expr

type program =
  | EProgram of declaration list * expr
```

```
i_expr
c_expr
a_expr
```

```
EAppl("f", EUnaryOp(OpInc, EInt 1))
```

```
def f(x, y)
```

```
f(inc(1))
```

```
let $1 = 1 in
let $2 = inc($1) in
f($2)
```

```
type c_expr =
  | CAppl of string * i_expr list
```

```
i_expr
c_expr
a_expr
a_decl
a-program
```

```
let rec compile_a_expression (env :environment) (ae:a_expr) =


;;
let rec compile_declaration (env :environment) (ad:a_decl) =


;;
let rec compile_a_program(env:environment) (ap :a_prog) =
```

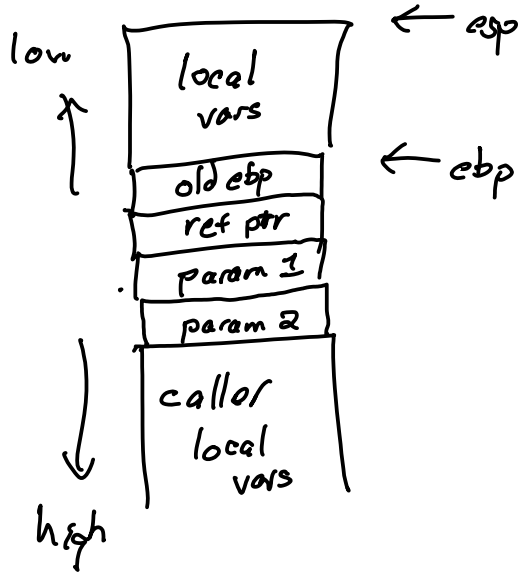
compile decl of f

```
;;
```

```
{x:+8}
x+1
```

```
def f(x):
    x+1
end
```

```
_f:
    push ebp
    mov ebp, esp
    sub esp, ??
    mov eax, [ebp+8]
    add eax, 1
    mov esp, ebp
    pop ebp
    ret
```




```
push eax
call _f
add esp, 4
```

compile call to f