In [155]:

```
import jax
import numpy

from numpy.linalg import inv
from jax import numpy as jnp
from jax import grad
```

## Problem №1

You will work with the following function for exercise, $f(x, y) = e^{-(sin(x)+cos(y))^2}$

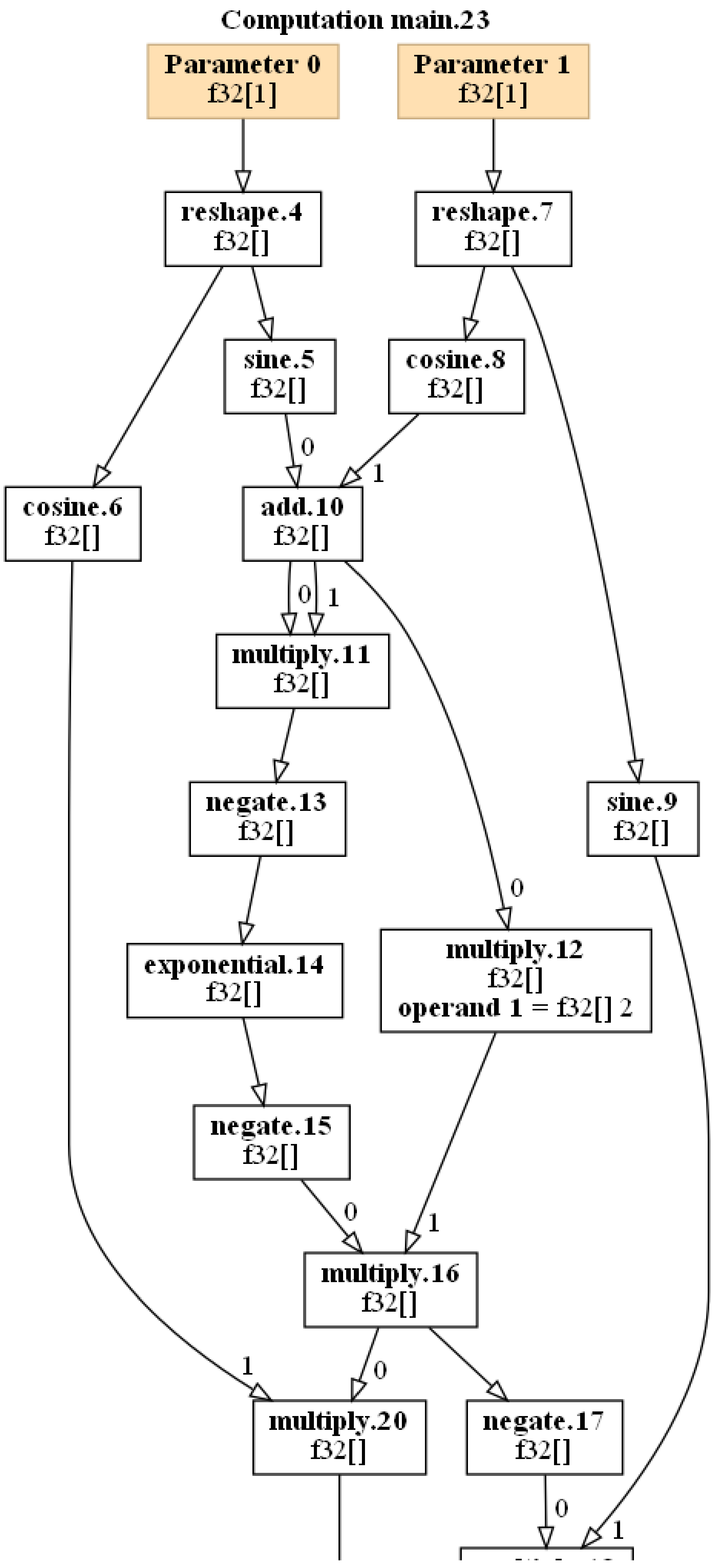
Draw the computational graph for the function. Note, that it should contain only primitive operations - you need to do it automatically.

In [156]:

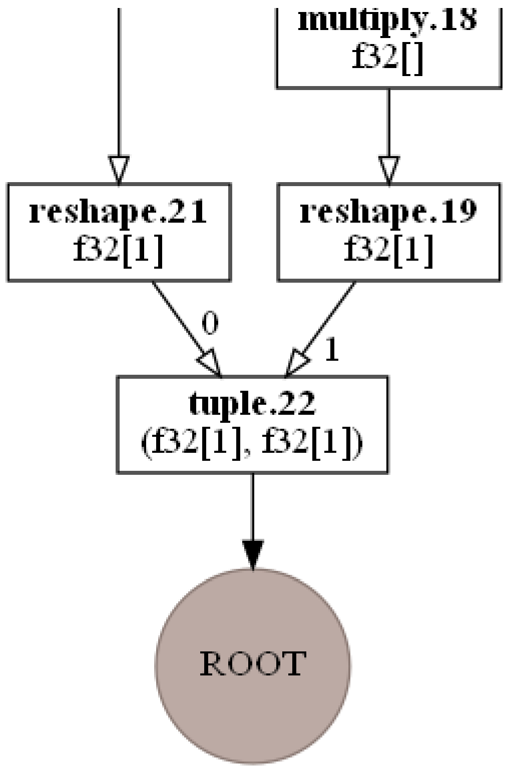
```
#Function of first problem
def func_p1(x, y):
  return jnp.exp(- jnp.power((jnp.sin(x[0]) + jnp.cos(y[0])), 2))

def dfunc_p1(x, y):
  return grad(func_p1, argnums=(0, 1))(x, y)
```

In [157]:

```
z=jax.xla_computation(dfunc_p1)(numpy.random.rand(1), numpy.random.rand(1))

with open("t1.txt", "w") as f:
  f.write(z.as_hlo_text())

with open("t1.dot", "w") as f:
  f.write(z.as_hlo_dot_graph())
```



Computation main.23
Parameter 0
f32[1]
Parameter 1
f32[1]
reshape.4
f32[]
reshape.7
f32[]
sine.5
f32[]
cosine.8
f32[]
cosine.6
f32[]
add.10
f32[]
multiply.11
f32[]
negate.13
f32[]
sine.9
f32[]
exponential.14
f32[]
multiply.12
f32[]
operand 1 = f32[] 2
negate.15
f32[]
multiply.16
f32[]
multiply.20
f32[]
negate.17
f32[]
0
1



## Problem №2

Compare analytic and autograd approach for the hessian of: $f(x) = \frac{1}{2}x^T Ax + b^T x + c$

In [158]:

```
from jax import jacfwd, jacrev
```

In [159]:

```
A = numpy.random.rand(100, 100)
b = numpy.random.rand(100)
c = 1

def func_p2(x):
 return 0.5 * x.T @ A @ x + b @ x + c

def hessian(f):
  return jax.jacfwd(jax.grad(f))

def d2func_p2(x):
  return hessian(func_p2)(x)

hessian_auto2 = d2func_p2(numpy.random.rand(100))
hessian_anal2 = (A + A.T) / 2
```

Difference between autograde and analytical solution:

In [160]:

```
numpy.linalg.norm(hessian_anal2 - hessian_auto2)
```

Out[160]:

```
2.1366172e-06
```

Cringe moment for visualising it

In [161]:

```
z = jax.xla_computation(d2func_p2)(numpy.random.rand(100))

with open("t2.txt", "w") as f:
   f.write(z.as_hlo_text())

with open("t2.dot", "w") as f:
  f.write(z.as_hlo_dot_graph())
```

## Problem №3

Suppose we have the following function $f(x) = \frac{1}{2}||x||^2$, select a random point $x_0 \in \mathbb{B}^{1000} = \{0 \leq x_i \leq 1 | \forall i\}$. Consider 10 steps of the gradient descent starting from the point $x_0$:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k).$$

Your goal in this problem to write the function, that takes 10 scalar values $\alpha_i$ and return the result of the gradient descent on function $L = f(x_{10})$. And optimize this function using gradient descent on $\alpha \in \mathbb{R}^{10}$. Suppose, $\alpha_0 = 1$.

$$\alpha_{k+1} = \alpha_k - \beta \cdot \frac{\partial L}{\partial \alpha}$$

Choose any $\beta$ and the number of steps your need. Describe obtained results.

In [162]:

```
import numpy as np
```

In [163]:

```
def dldalpha_p3(x, alpha):
    N = len(alpha) - 1
    dl = np.zeros(N + 1)
    cur_alpha = 1 - alpha[N]

    for k in range(N):
        dl[N - k ] = -x[N - k] @ x[N - k - 1] * cur_alpha
        cur_alpha *= (1-alpha[N - k - 1])

    return dl

def my_grad_p3(alpha, iterations):
    x = np.array([np.random.rand(1000) for i in range(len(alpha) + 1)])
    beta = 3e-3

    for l in range(iterations):
        for i in range(len(alpha)):
            x[i+1] =  x[i] - alpha[i] * x[i]

        alpha = alpha - beta * dldalpha_p3(x, alpha)

    return x
```

In [164]:

```
alpha = np.array([float(np.random.rand(1)) for i in range(10)])
x = my_grad_p3(alpha, 3)
print(x[-1])
# The method converges very quickly, despite the number of iterations (3 or 30),
# and then e-5 is already a machine zero hinders.
# Having tested it several times, everything strongly depends on the initial alpha.
# The closer to one, the better
```

```
 4.73998639e-04 1.09616949e-04 1.41240217e-04 4.28934702e-05
 1.21931998e-04 2.55045152e-04 2.57784649e-04 6.62678159e-05
 3.57227559e-04 3.54347994e-04 5.05101070e-04 4.57211362e-04
 2.12416394e-05 2.65531852e-04 2.75066628e-04 4.96563818e-04
 2.42761458e-04 6.84828172e-05 3.54887245e-04 1.75792293e-04
 4.96970409e-04 3.40873064e-04 1.14075589e-04 2.57406378e-04
 4.46789599e-04 2.32420815e-04 2.56337498e-04 2.01395761e-04
 8.06003241e-05 1.72717746e-04 9.82050641e-05 2.73584191e-04
 8.09531208e-05 2.11171733e-04 1.74160573e-04 7.33927375e-05
 4.80571989e-04 3.52904366e-04 4.45467706e-04 4.53448745e-05
 2.54705059e-04 2.70298146e-04 3.09885902e-04 1.72345172e-04
 4.60182349e-04 5.03076812e-04 1.58222722e-04 1.56601982e-04
 3.01523796e-04 3.47146878e-04 5.10606641e-04 3.21960333e-04
 2.58708699e-04 5.11791452e-04 4.37511632e-04 3.89106132e-05
 5.16222657e-05 5.31700141e-04 3.76305500e-04 3.32173200e-04
 6.28505105e-05 1.91547715e-05 1.46589186e-04 6.56351555e-05
 2.96961921e-05 3.34641570e-04 3.21975249e-04 1.70076498e-04
 3.23673641e-05 5.77914347e-05 2.63653661e-04 4.60466515e-04
 1.74001251e-05 2.56137355e-04 2.19101837e-04 2.08307824e-04]
```

## Problem №4

Compare analytic and autograd approach for the gradient of: $f(X) = -log(det(X))$

Analytical gradient: $df = -\frac{1}{det(X)} \cdot det(X)\langle X^{-T}, dX\rangle$

$$df = -\langle X^{-T}, dX\rangle$$

$$\nabla f = -X^{-T}$$

In [165]:

```
X = numpy.random.rand(100, 100)

func_p4 = lambda X: -jnp.log(jnp.linalg.det(X))
dfunc_p4 = lambda X: grad(func_p4)(X)

grad_auto4 = dfunc_p4(X)
grad_anal4 = -(inv(X)).T

print("Difference between analytical and autograde methods:",
      numpy.linalg.norm(grad_auto4 - grad_anal4))
```

```
Difference between analytical and autograde methods: 0.00013053074
```

## Problem №5

Compare analytic and autograd approach for the gradient and hessian of: $f(x) = x^T x x^T x$

In [166]:

```
def func_p5(x):
  return jnp.dot(x.T, x)* jnp.dot(x.T, x)

def dfunc_p5(x):
  return grad(func_p5)(x)

def d2func_p5(x):
  return hessian(func_p5)(x)
```

Analytical gradient: $df = 4\langle x, x\rangle \cdot \langle x, dx\rangle$

$$\nabla f = 4\langle x, x\rangle \cdot x$$

Analytical hessian: $d^2 f = 4 \cdot \Big(\langle dx_2, x\rangle \cdot \langle x, dx_1\rangle + \langle x, dx_2\rangle \cdot \langle x, dx_1\rangle + \langle x, x\rangle \cdot \langle dx_2, dx_1\rangle\Big)$

$$d^2 f = 8 \cdot (x, dx_1)(x, dx_2) + 4(x, x)(dx_2, dx_1) = (8x(x, dx_2), dx_1) + (4(x, x)dx_2, dx_1) =$$
$$= (8xx^T dx_2, dx_1) + (4(x, x)dx_2, dx_1) = ((8xx^T + 4(x, x)I)dx_2, dx_1)$$

$$hessian(f) = 8xx^T + 4(x, x)I$$

In [167]:

```
x = numpy.random.rand(10)
grad_auto5 = grad(func_p5)(x)
grad_anal5 = 4 * jnp.dot(x, x) * x

print("Difference between analytic and auto gradient",
      numpy.linalg.norm(grad_auto5 - grad_anal5))
```

```
Difference between analytic and auto gradient 0.0
```

In [168]:

```
hessian_auto5 = d2func_p5(x)
hessian_anal5 = 8*jnp.outer(x, x.T) + 4 * x @ x * jnp.eye(len(x))

print("Difference between analytic and auto hessian:",
      numpy.linalg.norm(hessian_auto5 - hessian_anal5))
```

```
Difference between analytic and auto hessian: 6.031566e-06
```