```solidity
// SPDX-License-Identifier: MIT

pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/utils/EnumerableSet.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "./SushiToken.sol";

interface IMigratorChef {
    // Perform LP token migration from legacy UniswapV2 to SushiSwap.
    // Take the current LP token address and return the new LP token address.
    // Migrator should have full access to the caller's LP token.
    // Return the new LP token address.
    //
    // XXX Migrator must have allowance access to UniswapV2 LP tokens.
    // SushiSwap must mint EXACTLY the same amount of SushiSwap LP tokens or
    // else something bad will happen. Traditional UniswapV2 does not
    // do that so be careful!
    function migrate(IERC20 token) external returns (IERC20);
}

// MasterChef is the master of Sushi. He can make Sushi and he is a fair guy.
//
// Note that it's ownable and the owner wields tremendous power. The ownership
// will be transferred to a governance smart contract once SUSHI is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract MasterChef is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    // Info of each user.
    struct UserInfo {
        uint256 amount; // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
        //
        // We do some fancy math here. Basically, any point in time, the amount of SUSHIs
        // entitled to a user but is pending to be distributed is:
        //
        //   pending reward = (user.amount * pool.accSushiPerShare) - user.rewardDebt
        //
        // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accSushiPerShare` (and `lastRewardBlock`) gets updated.
        //   2. User receives the pending reward sent to his/her address.
        //   3. User's `amount` gets updated.
        //   4. User's `rewardDebt` gets updated.
    }
    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken; // Address of LP token contract.
        uint256 allocPoint; // How many allocation points assigned to this pool. SUSHIs to distribute per block.
        uint256 lastRewardBlock; // Last block number that SUSHIs distribution occurs.
        uint256 accSushiPerShare; // Accumulated SUSHIs per share, times 1e12. See below.
    }
    // The SUSHI TOKEN!
    SushiToken public sushi;
    // Dev address.
    address public devaddr;
    // Block number when bonus SUSHI period ends.
    uint256 public bonusEndBlock;
    // SUSHI tokens created per block.
    uint256 public sushiPerBlock;
    // Bonus muliplier for early sushi makers.
    uint256 public constant BONUS_MULTIPLIER = 10;
    // The migrator contract. It has a lot of power. Can only be set through governance (owner).
    IMigratorChef public migrator;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when SUSHI mining starts.
    uint256 public startBlock;
    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(
        address indexed user,
        uint256 indexed pid,
        uint256 amount
    );

    constructor(
        SushiToken _sushi,
        address _devaddr,
        uint256 _sushiPerBlock,
        uint256 _startBlock,
        uint256 _bonusEndBlock
    ) public {
        sushi = _sushi;
        devaddr = _devaddr;
        sushiPerBlock = _sushiPerBlock;
        bonusEndBlock = _bonusEndBlock;
        startBlock = _startBlock;
    }

    function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accSushiPerShare: 0
            })
        );
    }

    // Update the given pool's SUSHI allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }

    // Migrate lp token to another lp contract. Can be called by anyone. We trust
    that migrator contract is good.
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "migrate: no migrator");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
    }

    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to)
        public
        view
        returns (uint256)
    {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from).mul(BONUS_MULTIPLIER);
        } else if (_from >= bonusEndBlock) {
            return _to.sub(_from);
        } else {
            return
                bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER).add(
                    _to.sub(bonusEndBlock)
                );
        }
    }

    // View function to see pending SUSHIs on frontend.
    function pendingSushi(uint256 _pid, address _user)
        external
        view
        returns (uint256)
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accSushiPerShare = pool.accSushiPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier =
                getMultiplier(pool.lastRewardBlock, block.number);
            uint256 sushiReward =
                multiplier.mul(sushiPerBlock).mul(pool.allocPoint).div(
                    totalAllocPoint
                );
            accSushiPerShare = accSushiPerShare.add(
                sushiReward.mul(1e12).div(lpSupply)
            );
        }
        return user.amount.mul(accSushiPerShare).div(1e12).sub(user.rewardDebt);
    }

    // Update reward vairables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 sushiReward =
            multiplier.mul(sushiPerBlock).mul(pool.allocPoint).div(
                totalAllocPoint
            );
        sushi.mint(devaddr, sushiReward.div(10));
        sushi.mint(address(this), sushiReward);
        pool.accSushiPerShare = pool.accSushiPerShare.add(
            sushiReward.mul(1e12).div(lpSupply)
        );
        pool.lastRewardBlock = block.number;
    }

    // Deposit LP tokens to MasterChef for SUSHI allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount.mul(pool.accSushiPerShare).div(1e12).sub(
                    user.rewardDebt
                );
            safeSushiTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending =
            user.amount.mul(pool.accSushiPerShare).div(1e12).sub(
                user.rewardDebt
            );
        safeSushiTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
    }

    // Safe sushi transfer function, just in case if rounding error causes pool to not have enough SUSHIs.
    function safeSushiTransfer(address _to, uint256 _amount) internal {
        uint256 sushiBal = sushi.balanceOf(address(this));
        if (_amount > sushiBal) {
            sushi.transfer(_to, sushiBal);
        } else {
            sushi.transfer(_to, _amount);
        }
    }

    // Update dev address by the previous dev.
    function dev(address _devaddr) public {
        require(msg.sender == devaddr, "dev: wut?");
        devaddr = _devaddr;
    }
}

```