/\*\*

\*Submitted for verification at BscScan.com on 2021-05-23

\*/

/\*\*

\*Submitted for verification at BscScan.com on 2021-02-26

\*/

// SPDX-License-Identifier: MIT

pragma solidity >=0.6.0 <0.8.0;

/\*\*

\* @dev Wrappers over Solidity's arithmetic operations with added overflow

\* checks.

\*

\* Arithmetic operations in Solidity wrap on overflow. This can easily result

\* in bugs, because programmers usually assume that an overflow raises an

\* error, which is the standard behavior in high level programming languages.

\* `SafeMath` restores this intuition by reverting the transaction when an

\* operation overflows.

\*

\* Using this library instead of the unchecked operations eliminates an entire

\* class of bugs, so it's recommended to use it always.

\*/

library SafeMath {

/\*\*

\* @dev Returns the addition of two unsigned integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `+` operator.

\*

\* Requirements:

\*

\* - Addition cannot overflow.

\*/

function add(uint256 a, uint256 b) internal pure returns (uint256) {

uint256 c = a + b;

require(c >= a, "SafeMath: addition overflow");

return c;

}

/\*\*

\* @dev Returns the LOTUStraction of two unsigned integers, reverting on

\* overflow (when the result is negative).

\*

\* Counterpart to Solidity's `-` operator.

\*

\* Requirements:

\*

\* - LOTUStraction cannot overflow.

\*/

function LOTUS(uint256 a, uint256 b) internal pure returns (uint256) {

return LOTUS(a, b, "SafeMath: LOTUStraction overflow");

}

/\*\*

\* @dev Returns the LOTUStraction of two unsigned integers, reverting with custom message on

\* overflow (when the result is negative).

\*

\* Counterpart to Solidity's `-` operator.

\*

\* Requirements:

\*

\* - LOTUStraction cannot overflow.

\*/

function LOTUS(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

require(b <= a, errorMessage);

uint256 c = a - b;

return c;

}

/\*\*

\* @dev Returns the multiplication of two unsigned integers, reverting on

\* overflow.

\*

\* Counterpart to Solidity's `\*` operator.

\*

\* Requirements:

\*

\* - Multiplication cannot overflow.

\*/

function mul(uint256 a, uint256 b) internal pure returns (uint256) {

// Gas optimization: this is cheaper than requiring 'a' not being zero, but the

// benefit is lost if 'b' is also tested.

// See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522

if (a == 0) {

return 0;

}

uint256 c = a \* b;

require(c / a == b, "SafeMath: multiplication overflow");

return c;

}

/\*\*

\* @dev Returns the integer division of two unsigned integers. Reverts on

\* division by zero. The result is rounded towards zero.

\*

\* Counterpart to Solidity's `/` operator. Note: this function uses a

\* `revert` opcode (which leaves remaining gas untouched) while Solidity

\* uses an invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function div(uint256 a, uint256 b) internal pure returns (uint256) {

return div(a, b, "SafeMath: division by zero");

}

/\*\*

\* @dev Returns the integer division of two unsigned integers. Reverts with custom message on

\* division by zero. The result is rounded towards zero.

\*

\* Counterpart to Solidity's `/` operator. Note: this function uses a

\* `revert` opcode (which leaves remaining gas untouched) while Solidity

\* uses an invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

require(b > 0, errorMessage);

uint256 c = a / b;

// assert(a == b \* c + a % b); // There is no case in which this doesn't hold

return c;

}

/\*\*

\* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

\* Reverts when dividing by zero.

\*

\* Counterpart to Solidity's `%` operator. This function uses a `revert`

\* opcode (which leaves remaining gas untouched) while Solidity uses an

\* invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function mod(uint256 a, uint256 b) internal pure returns (uint256) {

return mod(a, b, "SafeMath: modulo by zero");

}

/\*\*

\* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

\* Reverts with custom message when dividing by zero.

\*

\* Counterpart to Solidity's `%` operator. This function uses a `revert`

\* opcode (which leaves remaining gas untouched) while Solidity uses an

\* invalid opcode to revert (consuming all remaining gas).

\*

\* Requirements:

\*

\* - The divisor cannot be zero.

\*/

function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

require(b != 0, errorMessage);

return a % b;

}

}

pragma solidity >=0.6.4;

interface IBEP20 {

/\*\*

\* @dev Returns the amount of tokens in existence.

\*/

function totalSupply() external view returns (uint256);

/\*\*

\* @dev Returns the token decimals.

\*/

function decimals() external view returns (uint8);

/\*\*

\* @dev Returns the token symbol.

\*/

function symbol() external view returns (string memory);

/\*\*

\* @dev Returns the token name.

\*/

function name() external view returns (string memory);

/\*\*

\* @dev Returns the bep token owner.

\*/

function getOwner() external view returns (address);

/\*\*

\* @dev Returns the amount of tokens owned by `account`.

\*/

function balanceOf(address account) external view returns (uint256);

/\*\*

\* @dev Moves `amount` tokens from the caller's account to `recipient`.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* Emits a {Transfer} event.

\*/

function transfer(address recipient, uint256 amount) external returns (bool);

/\*\*

\* @dev Returns the remaining number of tokens that `spender` will be

\* allowed to spend on behalf of `owner` through {transferFrom}. This is

\* zero by default.

\*

\* This value changes when {approve} or {transferFrom} are called.

\*/

function allowance(address \_owner, address spender) external view returns (uint256);

/\*\*

\* @dev Sets `amount` as the allowance of `spender` over the caller's tokens.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* IMPORTANT: Beware that changing an allowance with this method brings the risk

\* that someone may use both the old and the new allowance by unfortunate

\* transaction ordering. One possible solution to mitigate this race

\* condition is to first reduce the spender's allowance to 0 and set the

\* desired value afterwards:

\* https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

\*

\* Emits an {Approval} event.

\*/

function approve(address spender, uint256 amount) external returns (bool);

/\*\*

\* @dev Moves `amount` tokens from `sender` to `recipient` using the

\* allowance mechanism. `amount` is then deducted from the caller's

\* allowance.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* Emits a {Transfer} event.

\*/

function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

/\*\*

\* @dev Emitted when `value` tokens are moved from one account (`from`) to

\* another (`to`).

\*

\* Note that `value` may be zero.

\*/

event Transfer(address indexed from, address indexed to, uint256 value);

/\*\*

\* @dev Emitted when the allowance of a `spender` for an `owner` is set by

\* a call to {approve}. `value` is the new allowance.

\*/

event Approval(address indexed owner, address indexed spender, uint256 value);

}

pragma solidity >=0.6.0 <0.8.0;

/\*

\* @dev Provides information about the current execution context, including the

\* sender of the transaction and its data. While these are generally available

\* via msg.sender and msg.data, they should not be accessed in such a direct

\* manner, since when dealing with GSN meta-transactions the account sending and

\* paying for execution may not be the actual sender (as far as an application

\* is concerned).

\*

\* This contract is only required for intermediate, library-like contracts.

\*/

abstract contract Context {

function \_msgSender() internal view virtual returns (address payable) {

return msg.sender;

}

function \_msgData() internal view virtual returns (bytes memory) {

this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691

return msg.data;

}

}

pragma solidity >=0.6.0 <0.8.0;

//import "../GSN/Context.sol";

/\*\*

\* @dev Contract module which provides a basic access control mechanism, where

\* there is an account (an owner) that can be granted exclusive access to

\* specific functions.

\*

\* By default, the owner account will be the one that deploys the contract. This

\* can later be changed with {transferOwnership}.

\*

\* This module is used through inheritance. It will make available the modifier

\* `onlyOwner`, which can be applied to your functions to restrict their use to

\* the owner.

\*/

abstract contract Ownable is Context {

address private \_owner;

event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

/\*\*

\* @dev Initializes the contract setting the deployer as the initial owner.

\*/

constructor () internal {

address msgSender = \_msgSender();

\_owner = msgSender;

emit OwnershipTransferred(address(0), msgSender);

}

/\*\*

\* @dev Returns the address of the current owner.

\*/

function owner() public view returns (address) {

return \_owner;

}

/\*\*

\* @dev Throws if called by any account other than the owner.

\*/

modifier onlyOwner() {

require(\_owner == \_msgSender(), "Ownable: caller is not the owner");

\_;

}

/\*\*

\* @dev Leaves the contract without owner. It will not be possible to call

\* `onlyOwner` functions anymore. Can only be called by the current owner.

\*

\* NOTE: Renouncing ownership will leave the contract without an owner,

\* thereby removing any functionality that is only available to the owner.

\*/

function renounceOwnership() public virtual onlyOwner {

emit OwnershipTransferred(\_owner, address(0));

\_owner = address(0);

}

/\*\*

\* @dev Transfers ownership of the contract to a new account (`newOwner`).

\* Can only be called by the current owner.

\*/

function transferOwnership(address newOwner) public virtual onlyOwner {

require(newOwner != address(0), "Ownable: new owner is the zero address");

emit OwnershipTransferred(\_owner, newOwner);

\_owner = newOwner;

}

}

pragma solidity >=0.6.0 <0.8.0;

//import "./IBEP20.sol";

//import "@openzeppelin/contracts/math/SafeMath.sol";

//import "@openzeppelin/contracts/utils/Address.sol";

/\*\*

\* @title SafeBEP20

\* @dev Wrappers around BEP20 operations that throw on failure (when the token

\* contract returns false). Tokens that return no value (and instead revert or

\* throw on failure) are also supported, non-reverting calls are assumed to be

\* successful.

\* To use this library you can add a `using SafeBEP20 for IBEP20;` statement to your contract,

\* which allows you to call the safe operations as `token.safeTransfer(...)`, etc.

\*/

library SafeBEP20 {

using SafeMath for uint256;

using Address for address;

function safeTransfer(IBEP20 token, address to, uint256 value) internal {

\_callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));

}

function safeTransferFrom(IBEP20 token, address from, address to, uint256 value) internal {

\_callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));

}

/\*\*

\* @dev Deprecated. This function has issues similar to the ones found in

\* {IBEP20-approve}, and its usage is discouraged.

\*

\* Whenever possible, use {safeIncreaseAllowance} and

\* {safeDecreaseAllowance} instead.

\*/

function safeApprove(IBEP20 token, address spender, uint256 value) internal {

// safeApprove should only be called when setting an initial allowance,

// or when resetting it to zero. To increase and decrease it, use

// 'safeIncreaseAllowance' and 'safeDecreaseAllowance'

// solhint-disable-next-line max-line-length

require((value == 0) || (token.allowance(address(this), spender) == 0),

"SafeBEP20: approve from non-zero to non-zero allowance"

);

\_callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));

}

function safeIncreaseAllowance(IBEP20 token, address spender, uint256 value) internal {

uint256 newAllowance = token.allowance(address(this), spender).add(value);

\_callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));

}

function safeDecreaseAllowance(IBEP20 token, address spender, uint256 value) internal {

uint256 newAllowance = token.allowance(address(this), spender).LOTUS(value, "SafeBEP20: decreased allowance below zero");

\_callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));

}

/\*\*

\* @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxing the requirement

\* on the return value: the return value is optional (but if data is returned, it must not be false).

\* @param token The token targeted by the call.

\* @param data The call data (encoded using abi.encode or one of its variants).

\*/

function \_callOptionalReturn(IBEP20 token, bytes memory data) private {

// We need to perform a low level call here, to bypass Solidity's return data size checking mechanism, since

// we're implementing it ourselves. We use {Address.functionCall} to perform this call, which verifies that

// the target address contains contract code and also asserts for success in the low-level call.

bytes memory returndata = address(token).functionCall(data, "SafeBEP20: low-level call failed");

if (returndata.length > 0) { // Return data is optional

// solhint-disable-next-line max-line-length

require(abi.decode(returndata, (bool)), "SafeBEP20: BEP20 operation did not succeed");

}

}

}

pragma solidity >=0.6.2 <0.8.0;

/\*\*

\* @dev Collection of functions related to the address type

\*/

library Address {

/\*\*

\* @dev Returns true if `account` is a contract.

\*

\* [IMPORTANT]

\* ====

\* It is unsafe to assume that an address for which this function returns

\* false is an externally-owned account (EOA) and not a contract.

\*

\* Among others, `isContract` will return false for the following

\* types of addresses:

\*

\* - an externally-owned account

\* - a contract in construction

\* - an address where a contract will be created

\* - an address where a contract lived, but was destroyed

\* ====

\*/

function isContract(address account) internal view returns (bool) {

// This method relies on extcodesize, which returns 0 for contracts in

// construction, since the code is only stored at the end of the

// constructor execution.

uint256 size;

// solhint-disable-next-line no-inline-assembly

assembly { size := extcodesize(account) }

return size > 0;

}

/\*\*

\* @dev Replacement for Solidity's `transfer`: sends `amount` wei to

\* `recipient`, forwarding all available gas and reverting on errors.

\*

\* https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost

\* of certain opcodes, possibly making contracts go over the 2300 gas limit

\* imposed by `transfer`, making them unable to receive funds via

\* `transfer`. {sendValue} removes this limitation.

\*

\* https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].

\*

\* IMPORTANT: because control is transferred to `recipient`, care must be

\* taken to not create reentrancy vulnerabilities. Consider using

\* {ReentrancyGuard} or the

\* https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects-interactions-pattern[checks-effects-interactions pattern].

\*/

function sendValue(address payable recipient, uint256 amount) internal {

require(address(this).balance >= amount, "Address: insufficient balance");

// solhint-disable-next-line avoid-low-level-calls, avoid-call-value

(bool success, ) = recipient.call{ value: amount }("");

require(success, "Address: unable to send value, recipient may have reverted");

}

/\*\*

\* @dev Performs a Solidity function call using a low level `call`. A

\* plain`call` is an unsafe replacement for a function call: use this

\* function instead.

\*

\* If `target` reverts with a revert reason, it is bubbled up by this

\* function (like regular Solidity function calls).

\*

\* Returns the raw returned data. To convert to the expected return value,

\* use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.decode#abi-encoding-and-decoding-functions[`abi.decode`].

\*

\* Requirements:

\*

\* - `target` must be a contract.

\* - calling `target` with `data` must not revert.

\*

\* \_Available since v3.1.\_

\*/

function functionCall(address target, bytes memory data) internal returns (bytes memory) {

return functionCall(target, data, "Address: low-level call failed");

}

/\*\*

\* @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with

\* `errorMessage` as a fallback revert reason when `target` reverts.

\*

\* \_Available since v3.1.\_

\*/

function functionCall(address target, bytes memory data, string memory errorMessage) internal returns (bytes memory) {

return functionCallWithValue(target, data, 0, errorMessage);

}

/\*\*

\* @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],

\* but also transferring `value` wei to `target`.

\*

\* Requirements:

\*

\* - the calling contract must have an ETH balance of at least `value`.

\* - the called Solidity function must be `payable`.

\*

\* \_Available since v3.1.\_

\*/

function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) {

return functionCallWithValue(target, data, value, "Address: low-level call with value failed");

}

/\*\*

\* @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValue`], but

\* with `errorMessage` as a fallback revert reason when `target` reverts.

\*

\* \_Available since v3.1.\_

\*/

function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {

require(address(this).balance >= value, "Address: insufficient balance for call");

require(isContract(target), "Address: call to non-contract");

// solhint-disable-next-line avoid-low-level-calls

(bool success, bytes memory returndata) = target.call{ value: value }(data);

return \_verifyCallResult(success, returndata, errorMessage);

}

/\*\*

\* @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],

\* but performing a static call.

\*

\* \_Available since v3.3.\_

\*/

function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {

return functionStaticCall(target, data, "Address: low-level static call failed");

}

/\*\*

\* @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],

\* but performing a static call.

\*

\* \_Available since v3.3.\_

\*/

function functionStaticCall(address target, bytes memory data, string memory errorMessage) internal view returns (bytes memory) {

require(isContract(target), "Address: static call to non-contract");

// solhint-disable-next-line avoid-low-level-calls

(bool success, bytes memory returndata) = target.staticcall(data);

return \_verifyCallResult(success, returndata, errorMessage);

}

function \_verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) private pure returns(bytes memory) {

if (success) {

return returndata;

} else {

// Look for revert reason and bubble it up if present

if (returndata.length > 0) {

// The easiest way to bubble the revert reason is using memory via assembly

// solhint-disable-next-line no-inline-assembly

assembly {

let returndata\_size := mload(returndata)

revert(add(32, returndata), returndata\_size)

}

} else {

revert(errorMessage);

}

}

}

}

pragma solidity >=0.4.0;

//import '@openzeppelin/contracts/access/Ownable.sol';

//import '@openzeppelin/contracts/GSN/Context.sol';

//import './IBEP20.sol';

//import '@openzeppelin/contracts/math/SafeMath.sol';

/\*\*

\* @dev Implementation of the {IBEP20} interface.

\*

\* This implementation is agnostic to the way tokens are created. This means

\* that a supply mechanism has to be added in a derived contract using {\_mint}.

\* For a generic mechanism see {BEP20PresetMinterPauser}.

\*

\* TIP: For a detailed writeup see our guide

\* https://forum.zeppelin.solutions/t/how-to-implement-BEP20-supply-mechanisms/226[How

\* to implement supply mechanisms].

\*

\* We have followed general OpenZeppelin guidelines: functions revert instead

\* of returning `false` on failure. This behavior is nonetheless conventional

\* and does not conflict with the expectations of BEP20 applications.

\*

\* Additionally, an {Approval} event is emitted on calls to {transferFrom}.

\* This allows applications to reconstruct the allowance for all accounts just

\* by listening to said events. Other implementations of the EIP may not emit

\* these events, as it isn't required by the specification.

\*

\* Finally, the non-standard {decreaseAllowance} and {increaseAllowance}

\* functions have been added to mitigate the well-known issues around setting

\* allowances. See {IBEP20-approve}.

\*/

contract BEP20 is Context, IBEP20, Ownable {

using SafeMath for uint256;

mapping(address => uint256) private \_balances;

mapping(address => mapping(address => uint256)) private \_allowances;

uint256 private \_totalSupply;

string private \_name;

string private \_symbol;

uint8 private \_decimals;

/\*\*

\* @dev Sets the values for {name} and {symbol}, initializes {decimals} with

\* a default value of 18.

\*

\* To select a different value for {decimals}, use {\_setupDecimals}.

\*

\* All three of these values are immutable: they can only be set once during

\* construction.

\*/

constructor(string memory name, string memory symbol) public {

\_name = name;

\_symbol = symbol;

\_decimals = 0;

}

/\*\*

\* @dev Returns the bep token owner.

\*/

function getOwner() external override view returns (address) {

return owner();

}

/\*\*

\* @dev Returns the name of the token.

\*/

function name() public override view returns (string memory) {

return \_name;

}

/\*\*

\* @dev Returns the symbol of the token, usually a shorter version of the

\* name.

\*/

function symbol() public override view returns (string memory) {

return \_symbol;

}

/\*\*

\* @dev Returns the number of decimals used to get its user representation.

\*/

function decimals() public override view returns (uint8) {

return \_decimals;

}

/\*\*

\* @dev See {BEP20-totalSupply}.

\*/

function totalSupply() public override view returns (uint256) {

return \_totalSupply;

}

/\*\*

\* @dev See {BEP20-balanceOf}.

\*/

function balanceOf(address account) public override view returns (uint256) {

return \_balances[account];

}

/\*\*

\* @dev See {BEP20-transfer}.

\*

\* Requirements:

\*

\* - `recipient` cannot be the zero address.

\* - the caller must have a balance of at least `amount`.

\*/

function transfer(address recipient, uint256 amount) public override returns (bool) {

\_transfer(\_msgSender(), recipient, amount);

return true;

}

/\*\*

\* @dev See {BEP20-allowance}.

\*/

function allowance(address owner, address spender) public override view returns (uint256) {

return \_allowances[owner][spender];

}

/\*\*

\* @dev See {BEP20-approve}.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\*/

function approve(address spender, uint256 amount) public override returns (bool) {

\_approve(\_msgSender(), spender, amount);

return true;

}

/\*\*

\* @dev See {BEP20-transferFrom}.

\*

\* Emits an {Approval} event indicating the updated allowance. This is not

\* required by the EIP. See the note at the beginning of {BEP20};

\*

\* Requirements:

\* - `sender` and `recipient` cannot be the zero address.

\* - `sender` must have a balance of at least `amount`.

\* - the caller must have allowance for `sender`'s tokens of at least

\* `amount`.

\*/

function transferFrom (address sender, address recipient, uint256 amount) public override returns (bool) {

\_transfer(sender, recipient, amount);

\_approve(

sender,

\_msgSender(),

\_allowances[sender][\_msgSender()].LOTUS(amount, 'BEP20: transfer amount exceeds allowance')

);

return true;

}

/\*\*

\* @dev Atomically increases the allowance granted to `spender` by the caller.

\*

\* This is an alternative to {approve} that can be used as a mitigation for

\* problems described in {BEP20-approve}.

\*

\* Emits an {Approval} event indicating the updated allowance.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\*/

function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {

\_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender].add(addedValue));

return true;

}

/\*\*

\* @dev Atomically decreases the allowance granted to `spender` by the caller.

\*

\* This is an alternative to {approve} that can be used as a mitigation for

\* problems described in {BEP20-approve}.

\*

\* Emits an {Approval} event indicating the updated allowance.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\* - `spender` must have allowance for the caller of at least

\* `LOTUStractedValue`.

\*/

function decreaseAllowance(address spender, uint256 LOTUStractedValue) public returns (bool) {

\_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender].LOTUS(LOTUStractedValue, 'BEP20: decreased allowance below zero'));

return true;

}

/\*\*

\* @dev Creates `amount` tokens and assigns them to `msg.sender`, increasing

\* the total supply.

\*

\* Requirements

\*

\* - `msg.sender` must be the token owner

\*/

function mint(uint256 amount) public onlyOwner returns (bool) {

\_mint(\_msgSender(), amount);

return true;

}

/\*\*

\* @dev Moves tokens `amount` from `sender` to `recipient`.

\*

\* This is internal function is equivalent to {transfer}, and can be used to

\* e.g. implement automatic token fees, slashing mechanisms, etc.

\*

\* Emits a {Transfer} event.

\*

\* Requirements:

\*

\* - `sender` cannot be the zero address.

\* - `recipient` cannot be the zero address.

\* - `sender` must have a balance of at least `amount`.

\*/

function \_transfer (address sender, address recipient, uint256 amount) internal {

require(sender != address(0), 'BEP20: transfer from the zero address');

require(recipient != address(0), 'BEP20: transfer to the zero address');

\_balances[sender] = \_balances[sender].LOTUS(amount, 'BEP20: transfer amount exceeds balance');

\_balances[recipient] = \_balances[recipient].add(amount);

emit Transfer(sender, recipient, amount);

}

/\*\* @dev Creates `amount` tokens and assigns them to `account`, increasing

\* the total supply.

\*

\* Emits a {Transfer} event with `from` set to the zero address.

\*

\* Requirements

\*

\* - `to` cannot be the zero address.

\*/

function \_mint(address account, uint256 amount) internal {

require(account != address(0), 'BEP20: mint to the zero address');

\_totalSupply = \_totalSupply.add(amount);

\_balances[account] = \_balances[account].add(amount);

emit Transfer(address(0), account, amount);

}

/\*\*

\* @dev Destroys `amount` tokens from `account`, reducing the

\* total supply.

\*

\* Emits a {Transfer} event with `to` set to the zero address.

\*

\* Requirements

\*

\* - `account` cannot be the zero address.

\* - `account` must have at least `amount` tokens.

\*/

function \_burn(address account, uint256 amount) internal {

require(account != address(0), 'BEP20: burn from the zero address');

\_balances[account] = \_balances[account].LOTUS(amount, 'BEP20: burn amount exceeds balance');

\_totalSupply = \_totalSupply.LOTUS(amount);

emit Transfer(account, address(0), amount);

}

/\*\*

\* @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.

\*

\* This is internal function is equivalent to `approve`, and can be used to

\* e.g. set automatic allowances for certain LOTUSsystems, etc.

\*

\* Emits an {Approval} event.

\*

\* Requirements:

\*

\* - `owner` cannot be the zero address.

\* - `spender` cannot be the zero address.

\*/

function \_approve (address owner, address spender, uint256 amount) internal {

require(owner != address(0), 'BEP20: approve from the zero address');

require(spender != address(0), 'BEP20: approve to the zero address');

\_allowances[owner][spender] = amount;

emit Approval(owner, spender, amount);

}

/\*\*

\* @dev Destroys `amount` tokens from `account`.`amount` is then deducted

\* from the caller's allowance.

\*

\* See {\_burn} and {\_approve}.

\*/

function \_burnFrom(address account, uint256 amount) internal {

\_burn(account, amount);

\_approve(account, \_msgSender(), \_allowances[account][\_msgSender()].LOTUS(amount, 'BEP20: burn amount exceeds allowance'));

}

}

pragma solidity 0.6.12;

//import "./libs/BEP20.sol";

// LOTUS with Governance.

contract LOTUS\_Token is BEP20('LOTUS Token', 'LOTUS') {

/// @notice Creates `\_amount` token to `\_to`. Must only be called by the owner (MasterChef).

function mint(address \_to, uint256 \_amount) public onlyOwner {

\_mint(\_to, \_amount);

\_moveDelegates(address(0), \_delegates[\_to], \_amount);

}

// Copied and modified from YAM code:

// https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernanceStorage.sol

// https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernance.sol

// Which is copied and modified from COMPOUND:

// https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/Comp.sol

/// @notice A record of each accounts delegate

mapping (address => address) internal \_delegates;

/// @notice A checkpoint for marking number of votes from a given block

struct Checkpoint {

uint32 fromBlock;

uint256 votes;

}

/// @notice A record of votes checkpoints for each account, by index

mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

/// @notice The number of checkpoints for each account

mapping (address => uint32) public numCheckpoints;

/// @notice The EIP-712 typehash for the contract's domain

bytes32 public constant DOMAIN\_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");

/// @notice The EIP-712 typehash for the delegation struct used by the contract

bytes32 public constant DELEGATION\_TYPEHASH = keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");

/// @notice A record of states for signing / validating signatures

mapping (address => uint) public nonces;

/// @notice An event thats emitted when an account changes its delegate

event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

/// @notice An event thats emitted when a delegate account's vote balance changes

event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

/\*\*

\* @notice Delegate votes from `msg.sender` to `delegatee`

\* @param delegator The address to get delegatee for

\*/

function delegates(address delegator)

external

view

returns (address)

{

return \_delegates[delegator];

}

/\*\*

\* @notice Delegate votes from `msg.sender` to `delegatee`

\* @param delegatee The address to delegate votes to

\*/

function delegate(address delegatee) external {

return \_delegate(msg.sender, delegatee);

}

/\*\*

\* @notice Delegates votes from signatory to `delegatee`

\* @param delegatee The address to delegate votes to

\* @param nonce The contract state required to match the signature

\* @param expiry The time at which to expire the signature

\* @param v The recovery byte of the signature

\* @param r Half of the ECDSA signature pair

\* @param s Half of the ECDSA signature pair

\*/

function delegateBySig(

address delegatee,

uint nonce,

uint expiry,

uint8 v,

bytes32 r,

bytes32 s

)

external

{

bytes32 domainSeparator = keccak256(

abi.encode(

DOMAIN\_TYPEHASH,

keccak256(bytes(name())),

getChainId(),

address(this)

)

);

bytes32 structHash = keccak256(

abi.encode(

DELEGATION\_TYPEHASH,

delegatee,

nonce,

expiry

)

);

bytes32 digest = keccak256(

abi.encodePacked(

"\x19\x01",

domainSeparator,

structHash

)

);

address signatory = ecrecover(digest, v, r, s);

require(signatory != address(0), "LOTUS::delegateBySig: invalid signature");

require(nonce == nonces[signatory]++, "LOTUS::delegateBySig: invalid nonce");

require(now <= expiry, "LOTUS::delegateBySig: signature expired");

return \_delegate(signatory, delegatee);

}

/\*\*

\* @notice Gets the current votes balance for `account`

\* @param account The address to get votes balance

\* @return The number of current votes for `account`

\*/

function getCurrentVotes(address account)

external

view

returns (uint256)

{

uint32 nCheckpoints = numCheckpoints[account];

return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;

}

/\*\*

\* @notice Determine the prior number of votes for an account as of a block number

\* @dev Block number must be a finalized block or else this function will revert to prevent misinformation.

\* @param account The address of the account to check

\* @param blockNumber The block number to get the vote balance at

\* @return The number of votes the account had as of the given block

\*/

function getPriorVotes(address account, uint blockNumber)

external

view

returns (uint256)

{

require(blockNumber < block.number, "LOTUS::getPriorVotes: not yet determined");

uint32 nCheckpoints = numCheckpoints[account];

if (nCheckpoints == 0) {

return 0;

}

// First check most recent balance

if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {

return checkpoints[account][nCheckpoints - 1].votes;

}

// Next check implicit zero balance

if (checkpoints[account][0].fromBlock > blockNumber) {

return 0;

}

uint32 lower = 0;

uint32 upper = nCheckpoints - 1;

while (upper > lower) {

uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow

Checkpoint memory cp = checkpoints[account][center];

if (cp.fromBlock == blockNumber) {

return cp.votes;

} else if (cp.fromBlock < blockNumber) {

lower = center;

} else {

upper = center - 1;

}

}

return checkpoints[account][lower].votes;

}

function \_delegate(address delegator, address delegatee)

internal

{

address currentDelegate = \_delegates[delegator];

uint256 delegatorBalance = balanceOf(delegator); // balance of underlying LOTUS (not scaled);

\_delegates[delegator] = delegatee;

emit DelegateChanged(delegator, currentDelegate, delegatee);

\_moveDelegates(currentDelegate, delegatee, delegatorBalance);

}

function \_moveDelegates(address srcRep, address dstRep, uint256 amount) internal {

if (srcRep != dstRep && amount > 0) {

if (srcRep != address(0)) {

// decrease old representative

uint32 srcRepNum = numCheckpoints[srcRep];

uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;

uint256 srcRepNew = srcRepOld.LOTUS(amount);

\_writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);

}

if (dstRep != address(0)) {

// increase new representative

uint32 dstRepNum = numCheckpoints[dstRep];

uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;

uint256 dstRepNew = dstRepOld.add(amount);

\_writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);

}

}

}

function \_writeCheckpoint(

address delegatee,

uint32 nCheckpoints,

uint256 oldVotes,

uint256 newVotes

)

internal

{

uint32 blockNumber = safe32(block.number, "LOTUS::\_writeCheckpoint: block number exceeds 32 bits");

if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {

checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;

} else {

checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);

numCheckpoints[delegatee] = nCheckpoints + 1;

}

emit DelegateVotesChanged(delegatee, oldVotes, newVotes);

}

function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {

require(n < 2\*\*32, errorMessage);

return uint32(n);

}

function getChainId() internal pure returns (uint) {

uint256 chainId;

assembly { chainId := chainid() }

return chainId;

}

}

pragma solidity 0.6.12;

//import "@openzeppelin/contracts/math/SafeMath.sol";

//import "./libs/IBEP20.sol";

//import "./libs/SafeBEP20.sol";

//import "@openzeppelin/contracts/access/Ownable.sol";

//import "./LOTUS\_Token.sol";

// MasterChef is the master of LOTUS. He can make LOTUS 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 LOTUS 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 SafeBEP20 for IBEP20;

// 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 LOTUS

// entitled to a user but is pending to be distributed is:

//

// pending reward = (user.amount \* pool.accLOTUSPerShare) - user.rewardDebt

//

// Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:

// 1. The pool's `accLOTUSPerShare` (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 {

IBEP20 lpToken; // Address of LP token contract.

uint256 allocPoint; // How many allocation points assigned to this pool. LOTUS to distribute per block.

uint256 lastRewardBlock; // Last block number that LOTUS distribution occurs.

uint256 accLOTUSPerShare; // Accumulated LOTUS per share, times 1e12. See below.

uint16 depositFeeBP; // Deposit fee in basis points

}

// The LOTUS TOKEN!

LOTUS\_Token public LOTUS;

// Dev address.

address public devaddr;

// LOTUS tokens created per block.

uint256 public LOTUSPerBlock;

// Bonus muliplier for early LOTUS makers.

uint256 public constant BONUS\_MULTIPLIER = 1;

// Deposit Fee address

address public feeAddress;

// Info of each pool.

PoolInfo[] public poolInfo;

// Info of each user that stakes LP tokens.

mapping (uint256 => mapping (address => UserInfo)) public userInfo;

// Total allocation points. Must be the sum of all allocation points in all pools.

uint256 public totalAllocPoint = 0;

// The block number when LOTUS 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(

LOTUS\_Token \_LOTUS,

address \_devaddr,

address \_feeAddress,

uint256 \_LOTUSPerBlock,

uint256 \_startBlock

) public {

LOTUS = \_LOTUS;

devaddr = \_devaddr;

feeAddress = \_feeAddress;

LOTUSPerBlock = \_LOTUSPerBlock;

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, IBEP20 \_lpToken, uint16 \_depositFeeBP, bool \_withUpdate) public onlyOwner {

require(\_depositFeeBP <= 10000, "add: invalid deposit fee basis points");

if (\_withUpdate) {

massUpdatePools();

}

uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;

totalAllocPoint = totalAllocPoint.add(\_allocPoint);

poolInfo.push(PoolInfo({

lpToken: \_lpToken,

allocPoint: \_allocPoint,

lastRewardBlock: lastRewardBlock,

accLOTUSPerShare: 0,

depositFeeBP: \_depositFeeBP

}));

}

// Update the given pool's LOTUS allocation point and deposit fee. Can only be called by the owner.

function set(uint256 \_pid, uint256 \_allocPoint, uint16 \_depositFeeBP, bool \_withUpdate) public onlyOwner {

require(\_depositFeeBP <= 10000, "set: invalid deposit fee basis points");

if (\_withUpdate) {

massUpdatePools();

}

totalAllocPoint = totalAllocPoint.LOTUS(poolInfo[\_pid].allocPoint).add(\_allocPoint);

poolInfo[\_pid].allocPoint = \_allocPoint;

poolInfo[\_pid].depositFeeBP = \_depositFeeBP;

}

// Return reward multiplier over the given \_from to \_to block.

function getMultiplier(uint256 \_from, uint256 \_to) public view returns (uint256) {

return \_to.LOTUS(\_from).mul(BONUS\_MULTIPLIER);

}

// View function to see pending LOTUS on frontend.

function pendingLOTUS(uint256 \_pid, address \_user) external view returns (uint256) {

PoolInfo storage pool = poolInfo[\_pid];

UserInfo storage user = userInfo[\_pid][\_user];

uint256 accLOTUSPerShare = pool.accLOTUSPerShare;

uint256 lpSupply = pool.lpToken.balanceOf(address(this));

if (block.number > pool.lastRewardBlock && lpSupply != 0) {

uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);

uint256 LOTUSReward = multiplier.mul(LOTUSPerBlock).mul(pool.allocPoint).div(totalAllocPoint);

accLOTUSPerShare = accLOTUSPerShare.add(LOTUSReward.mul(1e12).div(lpSupply));

}

return user.amount.mul(accLOTUSPerShare).div(1e12).LOTUS(user.rewardDebt);

}

// Update reward variables 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.allocPoint == 0) {

pool.lastRewardBlock = block.number;

return;

}

uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);

uint256 LOTUSReward = multiplier.mul(LOTUSPerBlock).mul(pool.allocPoint).div(totalAllocPoint);

LOTUS.mint(devaddr, LOTUSReward.div(10));

LOTUS.mint(address(this), LOTUSReward);

pool.accLOTUSPerShare = pool.accLOTUSPerShare.add(LOTUSReward.mul(1e12).div(lpSupply));

pool.lastRewardBlock = block.number;

}

// Deposit LP tokens to MasterChef for LOTUS 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.accLOTUSPerShare).div(1e12).LOTUS(user.rewardDebt);

if(pending > 0) {

safeLOTUSTransfer(msg.sender, pending);

}

}

if(\_amount > 0) {

pool.lpToken.safeTransferFrom(address(msg.sender), address(this), \_amount);

if(pool.depositFeeBP > 0){

uint256 depositFee = \_amount.mul(pool.depositFeeBP).div(10000);

pool.lpToken.safeTransfer(feeAddress, depositFee);

user.amount = user.amount.add(\_amount).LOTUS(depositFee);

}else{

user.amount = user.amount.add(\_amount);

}

}

user.rewardDebt = user.amount.mul(pool.accLOTUSPerShare).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.accLOTUSPerShare).div(1e12).LOTUS(user.rewardDebt);

if(pending > 0) {

safeLOTUSTransfer(msg.sender, pending);

}

if(\_amount > 0) {

user.amount = user.amount.LOTUS(\_amount);

pool.lpToken.safeTransfer(address(msg.sender), \_amount);

}

user.rewardDebt = user.amount.mul(pool.accLOTUSPerShare).div(1e12);

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];

uint256 amount = user.amount;

user.amount = 0;

user.rewardDebt = 0;

pool.lpToken.safeTransfer(address(msg.sender), amount);

emit EmergencyWithdraw(msg.sender, \_pid, amount);

}

// Safe LOTUS transfer function, just in case if rounding error causes pool to not have enough LOTUS.

function safeLOTUSTransfer(address \_to, uint256 \_amount) internal {

uint256 LOTUSBal = LOTUS.balanceOf(address(this));

if (\_amount > LOTUSBal) {

LOTUS.transfer(\_to, LOTUSBal);

} else {

LOTUS.transfer(\_to, \_amount);

}

}

// Update dev address by the previous dev.

function dev(address \_devaddr) public {

require(msg.sender == devaddr, "dev: wut?");

devaddr = \_devaddr;

}

function setFeeAddress(address \_feeAddress) public{

require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");

feeAddress = \_feeAddress;

}

//Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.

function updateEmissionRate(uint256 \_LOTUSPerBlock) public onlyOwner {

massUpdatePools();

LOTUSPerBlock = \_LOTUSPerBlock;

}

function transferERC20(address tokenAddress) external onlyOwner {

IBEP20(tokenAddress).transfer(msg.sender, IBEP20(tokenAddress).balanceOf(address(this)));

}

}