Summary

This document proposes a standard for fungible assets on Tezos. This standard focuses on transfer semantics and does not include often desired features like any notion of approvals and token metadata. This is a deliberate decision: such mechanisms are expected to emerge in complementary standards.

Fungible Asset Standard

The parameter of a contract implementing FA1.3 interface MUST be a tree of `Or`s and have entrypoint annotations.

The contract MUST be compatible with FA1, i.e. it MUST provide the following entrypoints:

1. `(address :from, (address :to, nat :value)) %transfer`
2. `view (address :owner) nat %getBalance`
3. `view unit nat %getTotalSupply`

`%getBalance` and `%getTotalSupply` entrypoints have the same semantics as they do in FA1. This standard specifies additional requirements for `%transfer` entrypoint, as explicitly allowed by FA1.

Additionally, the contract must support the following entrypoints:

1. `() %approveToken`
2. `() %unapproveToken`
3. `view (address :receiver) bool %acceptsTokens`
4. `(address :operator) %addOperator`
5. `(address :operator) %removeOperator`
6. `view (address :owner, address :operator) bool %isOperator`
7. `view () nat %granularity`

Unless explicitly whitelisted, the token receiver must implement the `TokenReceiver` interface:

- `(address :from, (address :to, nat :value)) %onTokensReceived`

The semantics of the on-receive hook is not limited by this standard. This hook MAY emit arbitrary operations. In particular the on-receive hook MAY initiate further token transfers by reentering the token contract. The token implementation MUST support reentract calls to `%transfer` and MUST NOT prohibit such reentrancy via internal mechanisms.

Entrypoint semantics

approveToken

This entrypoint adds the `SENDER` to the whitelist of addresses that are allowed to receive tokens regardless of whether `SENDER` implements `TokenReceiver` interface or not. Note that since implicit accounts do not run any code, **implicit accounts MUST call `%approveToken` for each token contract** in order to receive the tokens operated by the contract.

If some address is whitelisted and implements `TokenReceiver` interface, the `onTokensReceived` hook is still invoked upon transfer.

unapproveToken

This entrypoint removes the `SENDER` from the whitelist. This effectively blocks token transfers to the `SENDER` unless it implements the `TokenReceiver` interface.

acceptsTokens

Returns `true` if:

- The receiver is a contract that implements the `TokensReceiver` interface.
- The receiver is an expicitly whitelisted address, i.e. it agreed to receive tokens using `approveToken` entrypoint and the whitelisting is not revoked by `unapproveToken`.

addOperator

Let `:operator` transfer the funds of the `:owner` on behalf of the `:owner`.

In case the supplied `:operator` and `:owner` are equal, or `:operator` is already authorized for the `:owner`, the operation MUST be a no-op, i.e. it must not fail, emit any operations or modify the storage of the contract.

removeOperator

Revoke the `:operator` right to transfer the funds of the `:owner` on behalf of the `:owner`.

In case the supplied `:operator` and `:owner` are equal, the operation MUST fail with `CannotRemoveSelf`.

In case the supplied `:operator` is not authorized by the `:owner`, the operation MUST fail with `UnauthorizedOperator (address :owner, address :operator)`.

isOperator

A view that returns `true` if the `:operator` has been autorized by the `:owner`, otherwise returns `false`.

transfer

This entrypoint initiates a transfer call chain.

The transfer MUST fail if any of the following preconditions is violated:

1. `SENDER` can send tokens on behalf of the `:from` account, i.e. either:
    - `:from` account is equal to `SENDER`;
    - `SENDER` is an authorized operator of `:from`.
2. The `:from` account has at least `:value` tokens.
3. The `:to` address is either:
    - `:to` is a contract that conforms to the `TokenReceiver` interface;
    - `:to` is an expicitly whitelisted address, i.e. it agreed to receive tokens using `approveToken` entrypoint and the whitelisting is not revoked by `unapproveToken`.
4. `:value` is a multiple of the token `granularity`.

The implementation MAY add other conditions not explicitly forbidden by this standard.

Successful transfer MUST debit the address of `:to` with `:value` tokens. The amount of tokens credited from the `:from` account MAY be larger (but MUST NOT be less) than `:value`. The implementation MAY perform other transfers required according to the contract logic.

The token contract MUST support the following scenarios in case the prevalidation phase succeeded:

1. If the token receiver is a contract that conforms to the `TokenReceiver` interface, the implementation MUST emit an operation `(address :from, (address :to, nat :value)) %onTokensReceived` to the receiver contract. The implementation MUST NOT proxy this operation through another contract, i.e. the `SENDER` value in the hook MUST be equal to the token contract address.
2. If the receiver does not conform to the `TokenReceiver` interface, and the receiver is whitelisted, the transfer should continue.
3. Otherwise, the transfer MUST fail.

The following table demonstrates the required actions for each type of transfer:

Implements `TokenReceiver`	**Is whitelisted**	**Action**
Yes	Yes	Continue transfer, call `%onTokensReceived`
Yes	No	Continue transfer, call `%onTokensReceived`
No	Yes	Continue transfer
No	No	Fail with `UnsafeTransfer`

In addition to `NotEnoughBalance` error specified by FA1, this enrypoint can fail with:

- `UnsafeTransfer` – in case the receiver is neither whitelisted nor it implements the `TokenReceiver` interface.
- `UnsupportedAmount` – in case the specified amount is not a multiple of granularity.
- `UnauthorizedOperator (address :owner, address :operator)` – in case the sender is neither the owner of the tokens nor an authorized operator of the owner's tokens.

Note that the set of possible error conditions MAY be extended by the contract according to its logic.

granularity

Get the smallest part of the token that’s not divisible.

In other words, the granularity is the smallest amount of tokens (in the internal denomination) which MAY be transferred at any time.

The following rules MUST be applied regarding the granularity:

- The granularity value MUST be set at creation time.
- The granularity value MUST NOT be changed, ever.
- The granularity value MUST be greater than or equal to 1.
- All balances MUST be a multiple of the granularity.

Any amount of tokens (in the internal denomination) transferred MUST be a multiple of the granularity value.

Any operation that would result in a balance that's not a multiple of the granularity value MUST be considered invalid, and the transaction MUST revert.

NOTE: Most tokens SHOULD be fully partitionable, i.e., this function SHOULD return 1 unless there is a good reason for not allowing any fraction of the token.