Contents

1 Background

1.1 Procedural macros

Procedural macros allow creating syntax extensions as execution of a function. Procedural macros come in one of three flavors: • Function-like macros - custom!(. . .) • Derive macros - #[derive(CustomDerive)] • Attribute macros - #[CustomAttribute] Procedural macros allow you to run code at compile time that operates over Rust syntax, both consuming and producing Rust syntax. You can sort of think of procedural macros as functions from an AST to another AST.

1.2 Derive macros

Derive macros define new inputs for the derive attribute. These macros can create new items given the token stream of a struct, enum, or union. They can also define derive macro helper attributes. Custom derive macros are defined by a public function with the $\text{proc}_{\text{macroderive}}$ attribute and a signature of (TokenStream) -> TokenStream. The input TokenStream is the token stream of the item that has the derive attribute on it. The output TokenStream must be a set of items that are then appended to the module or block that the item from the input TokenStream is in.

2 NFT

For create NFT need to create struct:

```
#[derive(Icrc7)]
#[icrc7(max_query_batch_size = 100)]/default 100
```

```
#[icrc7(max_update_batch_size = 20)] //default 20
#[icrc7(default_take_value = 10)] //default 10
#[icrc7(max_take_value = 100)] //default 100
#[icrc7(max_memo_size = 32)] //defaut 32
#[icrc7(atomic_batch_transfers = false)] //default false
#[icrc7(tx_window = 2 * 60 * 60)] //default 2 hours
#[icrc7(permitted_drift = 2*60)] // default 2 min
pub struct Token {
    pub name: String,
    pub description: Option<String>,
    pub asset_name: String,
    pub asset_content_type: String,
    pub asset_hash: [u8; 32],
    pub metadata: Metadata,
    pub author: Principal,
    pub supply_cap: Option<u32>,
    pub total_supply: u32,
    pub created_at: u64,
    pub updated_at: u64,
}
```

with `Metadata` as any `struct` or `enum` or `custom type`

```
type Metadata = std::collections::HashMap<String,String>
```

this will generate all memory code for storage and all Icrc7 necessary functions

2.1 query

```
pub fn icrc7_collection_metadata() -> Metadata;
pub fn icrc7_symbol() -> String;
pub fn icrc7_name() -> String;
pub fn icrc7_description() -> Option<String>;
pub fn icrc7_logo() -> Option<String>;
pub fn icrc7_total_supply() -> Nat;
pub fn icrc7_supply_cap() -> Option<Nat>;
pub fn icrc7_max_query_batch_size() -> Option<Nat>;
pub fn icrc7_max_update_batch_size() -> Option<Nat>;
pub fn icrc7_default_take_value() -> Option<Nat>;
pub fn icrc7_max_take_value() -> Option<Nat>;
```

```
pub fn icrc7_max_memo_size() -> Option<Nat>;
pub fn icrc7_atomic_batch_transfers() -> Option<bool>;
pub fn icrc7_tx_window() -> Option<Nat>;
pub fn icrc7_permitted_drift() -> Option<Nat>;
pub fn icrc7_token_metadata(token_ids: Vec<Nat>) -> Vec<Option<Metadata>>;
pub fn icrc7_owner_of(token_ids: Vec<Nat>) -> Vec<Option<Account>>;
pub fn icrc7_balance_of(accounts: Vec<Account>) -> Vec<Nat>;
pub fn icrc7_tokens(prev: Option<Nat>, take: Option<Nat>) -> Vec<Nat>;
pub fn icrc7_tokens_of(account: Account, prev: Option<Nat>, take: Option<Nat>) -> Vec<
```

2.2 update

```
pub fn icrc7_transfer(args: Vec<TransferArg>) -> Vec<Option<TransferResult>>;
pub fn mint(args: MintArg) -> MintResult;
pub fn create_token(args: CreateTokenArg) -> Result<Nat, String>;
pub fn update_token(args: UpdateTokenArg) -> Result<(), String>;
```

2.2.1 with

```
pub struct TransferArg {
    pub from_subaccount: Option<Subaccount>,
    pub to: Account,
    pub token_id: Nat,
    pub memo: Option<Memo>,
    pub created_at_time: Option<u64>,
}

pub enum TransferError {
    NonExistingTokenId,
    InvalidRecipient,
    Unauthorized,
    TooOld,
    CreatedInFuture { ledger_time: u64 },
    Duplicate { duplicate_of: Nat },
    GenericError { error_code: Nat, message: String },
    GenericBatchError { error_code: Nat, message: String },
}

pub type TransferResult = Result<Nat, TransferError>;

pub struct CreateTokenArg {
```

```
    pub name: String,
    pub description: Option<String>,
    pub asset_name: String,
    pub asset_content_type: String,
    pub asset_content: ByteBuf,
    pub metadata: Metadata,
    pub supply_cap: Option<u32>,
    pub author: Principal,
    pub challenge: Option<ByteBuf>,
}

pub struct UpdateTokenArg {
    pub id: Nat,
    pub name: Option<String>,
    pub description: Option<String>,
    pub asset_name: Option<String>,
    pub asset_content_type: Option<String>,
    pub asset_content: Option<ByteBuf>,
    pub metadata: Option<Metadata>,
    pub supply_cap: Option<u32>,
    pub author: Option<Principal>,
}

pub struct MintArg {
    pub token_id: Nat,
    pub holders: BTreeSet<Principal>,
}

pub enum MintError {
    NonExistingTokenId,
    SupplyCapReached,
    GenericBatchError { error_code: Nat, message: String },
}

pub type MintResult = Result<Nat, MintError>;
```