Pymich

Release 0.9.0

PyratzLabs

Aug 01, 2023

CONTENTS:

Build Tezos smart contracts using the programming language with the largest developer community.

The goal of this project is to build a compiler for a Tezos smart contract language that is a subset of the Python language such that when the smart contract is interpreted in CPython, it outputs the same results as when its compiled-Michelson is interpreted in the Michelson VM.

That is, all Python code is not valid Python, but all Pymich code is valid Python and guarantees the same behavior. This allows to leverage the MyPy typechecker and IDE integration, autocomplete, Python test tooling (test runner, code coverage…), go to definition and everything that makes up a good Python development environment.

CHAPTER

ONE

INSTALLATION

This document describes various means of installing Pymich.

1.1 Pip

Dependencies:

- Pymich currently only supports Python 3.9.
- A hard dependency of Pymich is Pytezos, which you need to install accordingly (along with its own dependencies).

You can then simply install pymich as follows:

```
pip install pymich
```

The installation process will expose the `pymich` command if your `PATH` includes the path to your pip binary installs.

1.2 Docker

A Docker image is provided here.

```
docker run --rm -v "$PWD":"$PWD" -w "$PWD" pyratzlabs/pymich compile <contract-source>␣
↪michelson > output.tz
```

1.3 Git

```
git clone git@yourlabs.io:pyratzlabs/pymich.git
```

Running the tests is then done using `pytest`:

```
cd pymich
pytest
```

CHAPTER

TWO

GETTING STARTED

In this document, we'll be writting a Pymich implementation of the FA12 standard.

If you have not installed Pymich yet, please go ahead and do so before moving on with the rest of this document.

2.1 Contract Structure

A Pymich contract is defined as a class inheriting from `pymich.michelson_types.Contract`, and its attributes statically defined. Not that Pymich uses a subset of PEP 484 style hints to type the contract. Let's investigate how an FA1.2 token enhanced with a mint functionality might be implemented:

```
from pymich.michelson_type import *
from pymich.stdlib import SENDER
from dataclasses import dataclass


@dataclass
class AllowanceKey(Record):
    owner: Address
    spender: Address


class FA12(Contract):
    tokens: BigMap[Address, Nat]
    allowances: BigMap[AllowanceKey, Nat]
    total_supply: Nat
    owner: Address

    def mint(self, _to: Address, value: Nat):
        if SENDER != self.owner:
            raise Exception("Only owner can mint")

        self.total_supply = self.total_supply + value

        self.tokens[_to] = self.tokens.get(_to, Nat(0)) + value
```

2.1.1 Storage

The FA12 class implements our contract. We can see that it defines a storage with four keys, and is roughly equivalent to:

```
@dataclass
class Storage:
    tokens: BigMap[Address, Nat]
    allowances: BigMap[AllowanceKey, Nat]
    total_supply: Nat
    owner: Address
```

Where BigMap, Address and Nat are classes that will compile to the Michelson types big map, address and nat respectively. This storage defines the following keys:

- storage.tokens, the contract ledger, used to associate balances to addresses.
- storage.allowances, the FA1.2 standard allows for transfering tokens in the name of another address, provided that other address has allowed it via the FA12.approve method, which we'll implement shortly.
- storage.total_supply, an storage entry that keeps track of the token total supply.
- storage.owner, to keep track of some administrator. For instance, only the owner will be allowed to mint.

Notice also the use of the Record type which allows, through inheritence, to define one's own records (name tuples). We're using this feature to define an AllowanceKey data structure that'll compile to an annotated Michelson Pair n whene n is the record dimensionality. We're using this user-defined record as the storage.allowances big map key type.

2.1.2 Entrypoints

Entrypoints are defined as *public* methods for classes that inherit from pymich.michelson_types.Contract returning None. Our FA12.mint entrypoint takes two parameters, which will be refactored into a record by the compiler since Michelson entrypoints only support single arguments. Note that the entrypoint functions require type hints for the arguments.

Looking at the FA12.mint entrypoint, we are introduced to some of the Pymich syntax:

- Michelson's FAILWITH instruction can be raised by throwing regular Python excptions.
- Michelson's SENDER is available through the constant variable pymich.stdlib.SENDER.
- Michelson's nat type can be pushed onto the stack using the pymich.michelson_types.Nat constructor.
- We use pymich.michelson_type.BigMap.get(key, default) to retrieve the value at a given key if it exists and return a default otherwise.

Finally, you can see that the pymich.michelson_types.BigMap defines attribute getters and setters the same way a regular python dictionary does. However, as described in the Pymich types section, they behave differently as they are more representative of the underlying Michelson datastructure. Having a look at the pymich.michelson_types.BigMap implementation, we can clearly see that:

```
class BigMap(MichelsonType, Generic[KeyType, ValueType]):
    """Michelson big map abstraction. It differs with a regular Python
    dictionary in that it:
    - is instanciated from literals by deepcopying the literal key/values
    - adding an element will add a copy of that element
    - getting an element will return a copy of that element
```

(continues on next page)

(continued from previous page)

```
    - it is not iterable
    """

    def __init__(self):
        self.__dictionary: Dict[KeyType, ValueType] = {}

    def __getitem__(self, key: KeyType) -> ValueType:
        """Returns a COPY of the value to retrieve"""
        return deepcopy(self.__dictionary[key])

    def __setitem__(self, key: KeyType, value: ValueType):
        """Store a COPY of the value"""
        self.__dictionary[key] = deepcopy(value)
```

With that in mind, implementing the rest of the FA12 standard is fairly trivial. Let's go on by implementing the `FA2.approve` entrypoint:

```
@dataclass
class FA12(Contract):
    ...

    def approve(self, spender: Address, value: Nat):
        allowance_key = AllowanceKey(SENDER, spender)

        previous_value = self.allowances.get(allowance_key, Nat(0))

        if previous_value > Nat(0) and value > Nat(0):
            raise Exception("UnsafeAllowanceChange")

        self.allowances[allowance_key] = value
```

We simply define a new public method taking as arguments a `spender` and a `value` that the spender can spend in the same of `SENDER`.

The transfer function is a little longer to implement since we need to take into account the allowances that might have been setup:

```
@dataclass
class FA12(Contract):
    ...

    def transfer(self, _from: Address, _to: Address, value: Nat):
        if SENDER != _from:
            allowance_key = AllowanceKey(_from, SENDER)

            authorized_value = self.allowances.get(allowance_key, Nat(0))

            if (authorized_value - value) < Int(0):
                raise Exception("NotEnoughAllowance")

            self.allowances[allowance_key] = abs(authorized_value - value)

        from_balance = self.tokens.get(_from, Nat(0))
```

(continues on next page)

(continued from previous page)

```
        if (from_balance - value) < Int(0):
            raise Exception("NotEnoughBalance")

        self.tokens[_from] = abs(from_balance - value)

        to_balance = self.tokens.get(_to, Nat(0))

        self.tokens[_to] = to_balance + value
```

Notice that we are instantiating integers using `pymich.michelson_types.Int` when making the comparison `(authorized_value - value) < Int(0)`. This is because the subtraction of two natural numbers results in an integer, and comparison is only compatible between either natural numbers or integers in Michelson. This behavior is implemented in both `pymich.michelson_types.Nat` and `pymich.michelson_types.Int` such that Python typecheckers such as Mypy and Pyright that can be integrated directly in your editor can signal you a type error.

2.1.3 Views (Pre-Hangzhou)

Pre-Hangzhou, views were defined by the TZIP-4 standard and required a callback pattern. Although post-Hangzhou, this is not used so much, the FA1.2 and FA2 token standards were developped before Hangzhou and hence, require some getter entrypoints to behave the same way as TZIP-4. Pymich provides a simple to implement such views by having a contract class public method that returns some value. We now implement all three view methods that the FA1.2 standard defines:

```
class FA12(Contract):
    ....

    def getAllowance(self, owner: Address, spender: Address) -> Nat:
        return self.allowances.get(AllowanceKey(owner, spender), Nat(0))

    def getBalance(self, owner: Address) -> Nat:
        return self.tokens.get(owner, Nat(0))

    def getTotalSupply(self) -> Nat:
        return self.total_supply
```

2.2 Compiling the contract

Let's now compile the contract using the Pymich CLI. You can either output Michelson or its JSON serialization, Micheline. You can also view the Pymich intermediate repersentation if you so desire:

```
# Michelson output
pymich compile FA12.py > FA12.tz

# Micheline output
pymich compile FA12.py micheline > FA12.tz

# Pymich IR output
pymich compile FA12.py ir > FA12.pymich-ir
```

2.3 Appendix

The whole contract can be found in the Pymich end-to-end folder.

CHAPTER

THREE

SUPPORTED PYTHON SUBSET

Python and Michelson have very different semantics.

Our constraints are:

- **A Pymich typechecked program should always typecheck in MyPy +++**
  - This allows IDEs to signal most, if not all, typechecking errors.
  - Type errors that are caught by Pymich but not MyPy should throw at runtime when the smart contract is ran in CPython.
- How are closures handled?
- How to distinguish between signed and unsigned integers?
- How to play nice with python LSP autocompletes? (are we really going to compile Python's `List.append`?)

+++ since MyPy cannot be as restrictive than Pymich, *i.e.* valid MyPy typechecks can be invalid in Pymich

All of these problems can be solved by using immutable datastructures.

Note: Python's dynamic features are not supported. Indeed, since Pymich is statically compiled, some of these features are impossible to compile efficiently. Hence, think of Pymich as a static subset of Python's features.

3.1 Immutable data structures

Pymich is as simple Python immutable data structures library

Which makes sense, to each tool its job:

- ORM's have their own abstractions to manipulate DBs
- Numerical lib's have their own abstractions to manipulate array of numbers (ex: `np.array`)
- Pymich has **its own abstractions to manipulate Michelson datastructures**, that are fundamentally immutable.

All datastructures are documented here.

3.2 Closures

Keeping in mind that:

- Since we are using immutable datastructures,
- and reassigning variables are prevented in Python unless `global`, `local` or `nonlocal` keywords are used,
- then dissallowing them solves the problem of compiling closure reassignments.
- these keywords are so rarely used in Python, that this should not matter much

3.3 Closures

3.3.1 Example

```
k = Nat(10)

def f(x: Nat) -> Nat:
    return x + k

def g(x: Nat) -> Nat:
    return k * x

def compose(x: Nat) -> Nat:
    return k * f(g(x))

result = compose(k)
```

3.3.2 Counter-example

Since Pymich closures partially apply variables in closures, the can not be mutated after the closure is defined to preserve. Pymich should throw an exception in strict mode and a warning otherwise:

```
i = Nat(1)
def f(x: Nat) -> Nat:
    return x + i
i = Nat(2)

result = f(10)
# CPython returns Nat(12)
# Pymich retuns Nat(11)
```

3.4 Inter contract calls

The `Contract` class exposes an `ops` attribute of type `Operations`. It is just a Michelson list of operations that only allows adding elements by pushing. Under the hood, the compiler prepends all transactions and inverses the operation list befor returning it.

This allows for constructing operations through functions.

The following code typechecks in MyPy:

```
@dataclass
class MintOrBurn(Record):
    quantity: Int
    target: Address


def mint_or_burn(
        ops: Operations,
        lqt_address: Address,
        target: Address,
        quantity: Int,
    ) -> Operations:
    mint_or_burn_entrypoint = Contract[MintOrBurn](lqt_address, "%transfer")
    return ops.push(
        mint_or_burn_entrypoint,
        Mutez(0),
        MintOrBurn(quantity, target),
    )


class Proxy(Contract):
    def mint_or_burn(self, x: Nat) -> None:
        self.ops = mint_or_burn(self.ops, token_address, owner, Int(10))
```

3.4.1 Mypy typechecking

The `Operations` class type annotated in such a way that Mypy fails in the following example since `MintOrBurn(quantity, target)` is not of type `Nat`:

```
mint_or_burn_entrypoint = Contract[Nat](lqt_address, "%transfer")
return ops.push(
    mint_or_burn_entrypoint,
    Mutez(0),
    MintOrBurn(quantity, target),
)
```

3.5 Exceptions

Exceptions will be compiled as Michelson `PUSH string "my error message" ; FAILWITH` instructions and can be raised as expected:

```
def floor_div(num: Nat, denom: Nat):
    if denom == Nat(0):
        raise Exception("Cannot divide by 0!")
    else:
        return num // denom
```

Note: Note that `Exception` takes a Python string as argument rather than a Michelson string as this was more convenient than writting `Exception(String("My error message"))`.

3.6 Python unsupported by Pymich

- passing `self` as a function parameter
- return anywhere other than as the last expression of a function body

CHAPTER

FOUR

PYMICH TYPES

Pymich defines some special data structures in order to be able to respect the semantic of the underlying generated Michelson code. Indeed, Michelson lists behave vastly differently than Python lists. Although it would be possible to compile Python lists to Michelson, this could not be done without introducing a significant overhead resulting in much higher gas costs than expected. Instead, the route that Pymich takes is to redefine the basic datastructures provided by Michelson and respect its semantics. These reimplemented data-structures ensure that one gets the expected gas consumptions when compiling down to Michelson while maintaining the same results when ran under CPython and a Michelson VM.

Morever, a second argument in favor of redefining the datatypes is that Michelson data types do not necessarily implement all methods that Python types do. An example of that are lists that, in Python, define an `append`, whereas its Michelson counterpart do not. Hence, by redefining the data structures, we can expose only the relevant methods, and Python IDEs will be able to typecheck Pymich code and provide proper auto complete.

These data structures are implemented à la Immutable.js, a Javascript library designed by Facebook in order to disallow mutations. The idea is that any datastructure modification returns a new data structure rather than modifies the original one. The implementation of these datastructures can be found here

Before diving into the specifics of each data-type, let us go through an example of how these immutable data structures behave:

```
from pymich.michelson_types import Nat, Map

michelson_map_a = Map[Nat, Nat]()
michelson_map_a[Nat(1)] = Nat(1)

michelson_map_b = Map[Nat, Map[Nat, Nat]]()
michelson_map_b[Nat(1)] = michelson_map_a

michelson_map_a_copy = michelson_map_b[Nat(1)]
michelson_map_a_copy[Nat(2)] = Nat(2)

michelson_map_a_copy[Nat(2)]  # works
michelson_map_a[Nat(2)]       # throws
```

Where as using the native Python dictionary would lead to the following behavior:

```
python_map_a = {1: 1}
python_map_b = {1: python_map_a}
python_map_a_bis = python_map_b[1]
python_map_a_bis[2] = 2
```

(continues on next page)

(continued from previous page)

```
python_map_a_bis[2]  # works
python_map_a[2]      # works as well
```

Hence, by using the Pymich data structures, we can run our contracts in the CPython interpreter and benefit from all the Python tooling while playing well with the underlying Michelson data structures.

4.1 Base types

Base types are the simplest types possible in Pymich. Notice that all base types are constructed from a constructor in `pymich.michelson_types` and take a python type as argument. Hence in Pymich, we only use Python strings to construct `pymich.michelson_types.String` and `pymich.michelson_types.Address`. Python numbers are used to construct `pymich.michelson_types.Nat`, `pymich.michelson_types.Int`, `pymich.michelson_types.Mutez` and `pymich.michelson_types.Timestamp`.

```
from pymich.michelson_types import *

some_address = Address("KT1Ha4yFVeyzw6KRAdkzq6TxDHB97KG4pZe8")
some_string = String("my string")
some_nat = Nat(10)
some_int = Int(-10)
some_mutez = Mutez(42)
some_timestamp = Timestamp(1650053840)
```

This choice allows for powerful integration into the Python dev tooling ecosystem. For example, using Mypy, one can get proper typechecking right into any editor:

```
Nat(10) - Nat(15)  # returns an Int
Int(10) + Nat(10)  # returns an Int
Int(10) < Nat(15)  # throws a type error
Nat(10) / Nat(3)   # div operator not allowed
Nat(10) // Nat(3)  # floor div is allowed
len(String("foo")  # returns a String
Int(-1).is_nat()   # returns None
Int(1).is_nat()    # returns Nat(1)
```

4.2 Generic types

Generic types are data structures that are build from basic types such as lists, maps, big maps, contracts and sets.

4.2.1 Lists

```
my_list = List[Nat]()
my_new_list = my_list.prepend(Nat(42))
my_list_length = len(my_list)

# loops
sum = Nat(0)
for el in my_list:
    sum = sum + el
```

Note: In the near future, list literals will be able to instanciated with `List(el1, ..., eln)`.

4.2.2 Maps

```
my_map = Map[Nat, String]()
my_key = Nat(42)
my_map[my_key] = String("The answer.")
my_string = my_map[my_key]
my_second_map = my_map.add(Nat(10), String("Ten"))
foo = Nat(10) in my_second_map  # foo is typed as a boolean

# loops
sum = Nat(0)
for key, value in my_list:
    sum = key + len(value)
```

4.2.3 Big maps

```
my_big_map = BigMap[Nat, String]()
my_key = Nat(42)
my_big_map[my_key] = String("The answer.")
my_string = my_big_map[my_key]
my_second_big_map = my_big_map.add(Nat(10), String("Ten"))
foo = Nat(10) in my_second_map  # foo is typed as a boolean
```

4.2.4 Sets

```
my_big_map = BigMap[Nat, String]()
my_key = Nat(42)
my_big_map[my_key] = String("The answer.")
my_string = my_big_map[my_key]
my_second_big_map = my_big_map.add(Nat(10), String("Ten"))
foo = Nat(10) in my_second_map  # foo is typed as a boolean
```

4.2.5 Records

```
class Student(Record):
    name: String
    age: Nat
    grades: Map[String, Nat]

alice = Student(
    Name("alice"),
    Nat(42),
    Map[String, Nat](),
)
class_code = String("FR101")
alice.grades[class_code] = Nat(15)
french_grade = alice.grades[class_code]
```

Note: Just like with other data structures, record attribute access returns a **copy** of its content

4.2.6 Contracts

```
@dataclass
class TransferParam:
    _from: Address
    _to: Address
    value: Nat

fa12 = "KT1EwUrkbmGxjiRvmEAa8HLGhjJeRocqVTFi"
transfer_entrypoint = Contract[TransferParam](fa12, "%transfer")
```

4.2.7 Transactions

```
# Transfer to a contract
transaction(
    transfer_entrypoint,
    Mutez(0),
    TransferParam(self.fa12, self.fa12, Nat(10)),
)
```

(continues on next page)

(continued from previous page)

```
# Transfer to an implicit address
transaction(Contract[Unit](SENDER), AMOUNT, Unit)
```

4.3 Grammar

```
python_types     ::=
                   | python_number
                   | python_string

base_types       ::=
                   | Unit
                   | String
                   | Nat
                   | Int
                   | Boolean
                   | Address
                   | Bytes

type             ::=
                   | base_type
                   | Operation
                   | Contract
                   | Map[type, type]
                   | BigMap[type, type]
                   | List[type]
                   | Set[type]
                   | Record[attr1=type, ..., attrn=type]
```

4.4 API

class michelson_types.**Address**(*address: str*)

__eq__(*other: Address*) → bool

Return self==value.

__hash__() → int

Return hash(self).

class michelson_types.**BaseContract**(*ops: 'Operations' = <michelson_types.Operations object at 0x7f8a0acf3650>*)

__eq__(*other*)

Return self==value.

__hash__ = None

class michelson_types.**BigMap**

Michelson big map abstraction. It differs with a regular Python dictionary in that it: - is instanciated from literals by deepcopying the literal key/values - adding an element will add a copy of that element - getting an element will return a copy of that element - it is not iterable

__getitem__(*key: KeyType*) → ValueType

Returns a COPY of the value to retrieve

__setitem__(*key: KeyType*, *value: ValueType*) → None

Store a COPY of the value

__str__() → str

Return str(self).

add(*key: KeyType*, *value: ValueType*) → *BigMap*[KeyType, ValueType]

Returns a COPY of self with a COPY of the value added

get(*key: KeyType*, *default: ValueType*) → ValueType

Returns a copy of the value

remove(*key: KeyType*) → *BigMap*[KeyType, ValueType]

Returns a COPY of the value

class michelson_types.**Bytes**(*value:* MichelsonType)

__eq__(*other:* Bytes) → bool

Return self==value.

__ge__(*other:* Bytes) → NotImplementedError

Return self>=value.

__gt__(*other:* Bytes) → NotImplementedError

Return self>value.

__hash__() → int

Return hash(self).

__le__(*other:* Bytes) → NotImplementedError

Return self<=value.

__lt__(*other:* Bytes) → NotImplementedError

Return self<value.

class michelson_types.**Contract**(*address:* Address, *entrypoint_name: Optional[str] = None*)

class michelson_types.**Int**(*value: int*)

__eq__(*other:* Int) → bool

Return self==value.

__ge__(*other:* Int) → bool

Return self>=value.

__gt__(*other:* Int) → bool

Return self>value.

__hash__() → int

Return hash(self).

__le__(*other: Int*) → bool

Return self<=value.

__lt__(*other: Int*) → bool

Return self<value.

class michelson_types.**List**(**elements: ParameterType*)

__hash__() → int

Return hash(self).

__str__() → str

Return str(self).

class michelson_types.**Map**

Michelson big map abstraction. It differs with a regular Python dictionary in that it: - is instanciated from literals by deepcopying the literal key/values - adding an element will add a copy of that element - getting an element will return a copy of that element

__hash__() → int

Return hash(self).

__setitem__(*key: KeyType*, *value: ValueType*) → None

Store a COPY of the value

__str__() → str

Return str(self).

add(*key: KeyType*, *value: ValueType*) → *Map*[KeyType, ValueType]

Returns a COPY of self with a COPY of the value added

get(*key: KeyType*, *default: ValueType*) → ValueType

Returns a copy of the value

remove(*key: KeyType*) → *Map*[KeyType, ValueType]

Returns a COPY of the value

class michelson_types.**MichelsonType**

Base type to inherit from to define further types.

class michelson_types.**Mutez**(*amount: int*)

__eq__(*other: Mutez*) → bool

Return self==value.

__ge__(*other: Mutez*) → bool

Return self>=value.

__gt__(*other: Mutez*) → bool

Return self>value.

__hash__() → int

Return hash(self).

__le__(*other: Mutez*) → bool

Return self<=value.

__lt__(*other: Mutez*) → bool

Return self<value.

class michelson_types.**Nat**(*value: int*)

__eq__(*other:* Nat) → bool

Return self==value.

__ge__(*other:* Nat) → bool

Return self>=value.

__gt__(*other:* Nat) → bool

Return self>value.

__hash__() → int

Return hash(self).

__le__(*other:* Nat) → bool

Return self<=value.

__lt__(*other:* Nat) → bool

Return self<value.

class michelson_types.**Operation**

class michelson_types.**Option**(*el: Optional[ParameterType] = None*)

__eq__(*other:* Option) → bool

Return self==value.

__hash__ = None

class michelson_types.**Record**

__eq__(*o: object*) → bool

Return self==value.

__hash__() → int

Return hash(self).

__setattr__(*_Record__name: str*, *value:* MichelsonType) → None

Implement setattr(self, name, value).

class michelson_types.**Set**(**args: ParameterType*)

Michelson set abstraction. It differs with a regular Python set in that when instanciating or adding an element to a set, a copy of that element is added rather than the element itself.

__hash__() → int

Return hash(self).

__str__() → str

Return str(self).

add(*element: ParameterType*) → None

Adds a COPY of the value

remove(*element: ParameterType*) → None

Removes an element from the set

class michelson_types.**String**(*value: str*)

Michelson string type.

Parameters
value – value

__add__(*other:* String) → *String*

String concatenation

__eq__(*other:* String) → bool

Return self==value.

__ge__(*other:* String) → bool

Return self>=value.

__gt__(*other:* String) → bool

Return self>value.

__hash__() → int

Return hash(self).

__le__(*other:* String) → bool

Return self<=value.

__lt__(*other:* String) → bool

Return self<value.

__str__() → str

Return str(self).

class michelson_types.**Timestamp**(*timestamp: int*)

__eq__(*other:* Timestamp) → bool

Return self==value.

__ge__(*other:* Timestamp) → bool

Return self>=value.

__gt__(*other:* Timestamp) → bool

Return self>value.

__hash__() → int

Return hash(self).

__le__(*other:* Timestamp) → bool

Return self<=value.

__lt__(*other:* Timestamp) → bool

Return self<value.

__str__() → str

Return str(self).

class michelson_types.**Unit**

Michelson Unit type.

CHAPTER

FIVE

PYMICH CLI

Pymich provides a simple self-documented CLI using cli2. If you have installed Pymich via Pypi, you will have the executable available in your shell provided Python's packages are in your path.

```
$ pymich

ERROR: No sub-command provided

SYNOPSYS
pymich.py SUB-COMMAND <...>
pymich.py help SUB-COMMAND

DESCRIPTION
Represents a group of named commands.

SUB-COMMANDS
help     Get help for a command or group
compile  Compiles a Python file and outputs the result to stdout
```

5.1 Compiling

To Michelson:

```
pymich compile <file-path> michelson
```

To Micheline:

```
pymich compile <file-path> micheline
```

To Pymich-IR:

```
pymich compile <file-path> ir
```

CHAPTER

SIX

TESTING

CHAPTER

SEVEN

TOOLING

CHAPTER

EIGHT

EXAMPLES

8.1 FA1.2

```
from dataclasses import dataclass
from pymich.michelson_types import *


@dataclass(eq=True, frozen=True)
class AllowanceKey(Record):
    owner: Address
    spender: Address


@dataclass(kw_only=True)
class FA12(BaseContract):
    tokens: BigMap[Address, Nat]
    allowances: BigMap[AllowanceKey, Nat]
    total_supply: Nat
    owner: Address

    def mint(self, _to: Address, value: Nat) -> None:
        if Tezos.sender != self.owner:
            raise Exception("Only owner can mint")

        self.total_supply = self.total_supply + value

        self.tokens[_to] = self.tokens.get(_to, Nat(0)) + value

    def approve(self, spender: Address, value: Nat) -> None:
        allowance_key = AllowanceKey(Tezos.sender, spender)

        previous_value = self.allowances.get(allowance_key, Nat(0))

        if previous_value > Nat(0) and value > Nat(0):
            raise Exception("UnsafeAllowanceChange")

        self.allowances[allowance_key] = value

    def transfer(self, _from: Address, _to: Address, value: Nat) -> None:
        if Tezos.sender != _from:
```

(continues on next page)

(continued from previous page)

```
            allowance_key = AllowanceKey(_from, Tezos.sender)

            authorized_value = self.allowances.get(allowance_key, Nat(0))

            if (authorized_value - value) < Int(0):
                raise Exception("NotEnoughAllowance")

            self.allowances[allowance_key] = abs(authorized_value - value)

        from_balance = self.tokens.get(_from, Nat(0))

        if (from_balance - value) < Int(0):
            raise Exception("NotEnoughBalance")

        self.tokens[_from] = abs(from_balance - value)

        to_balance = self.tokens.get(_to, Nat(0))

        self.tokens[_to] = to_balance + value

    def getAllowance(self, owner: Address, spender: Address) -> Nat:
        return self.allowances.get(AllowanceKey(owner, spender), Nat(0))

    def getBalance(self, owner: Address) -> Nat:
        return self.tokens.get(owner, Nat(0))

    def getTotalSupply(self) -> Nat:
        return self.total_supply
```

8.2 FA2 multi-asset

```
from dataclasses import dataclass
from pymich.michelson_types import *


@dataclass#(eq=True, frozen=True)
class OperatorKey(Record):
    owner: Address
    operator: Address
    token_id: Nat

@dataclass#(eq=True, frozen=True)
class LedgerKey(Record):
    owner: Address
    token_id: Nat

@dataclass#(eq=True, frozen=True)
class TokenMetadata(Record):
    token_id: Nat
    token_info: Map[String, Bytes]
```

(continues on next page)

(continued from previous page)

```
@dataclass
class TransactionInfo(Record):
    to_: Address
    token_id: Nat
    amount: Nat

@dataclass
class TransferArgs(Record):
    from_: Address
    txs: List[TransactionInfo]


def require_owner(owner: Address) -> Nat:
    if Tezos.sender != owner:
        raise Exception("FA2_NOT_CONTRACT_ADMINISTRATOR")

    return Nat(0)


@dataclass(kw_only=True)
class FA2(BaseContract):
    ledger: BigMap[LedgerKey, Nat]
    operators: BigMap[OperatorKey, Nat]
    token_total_supply: BigMap[Nat, Nat]
    token_metadata: BigMap[Nat, TokenMetadata]
    owner: Address

    def create_token(self, metadata: TokenMetadata) -> None:
        require_owner(self.owner)

        new_token_id = metadata.token_id

        if new_token_id in self.token_metadata:
            raise Exception("FA2_DUP_TOKEN_ID")
        else:
            self.token_metadata[new_token_id] = metadata
            self.token_total_supply[new_token_id] = Nat(0)

    def mint_tokens(self, owner: Address, token_id: Nat, amount: Nat) -> None:
        require_owner(self.owner)

        if not token_id in self.token_metadata:
            raise Exception("FA2_TOKEN_DOES_NOT_EXIST")

        ledger_key = LedgerKey(owner, token_id)
        self.ledger[ledger_key] = self.ledger.get(ledger_key, Nat(0)) + amount
        self.token_total_supply[token_id] = self.token_total_supply[token_id] + amount

    def transfer(self, transactions: List[TransferArgs]) -> None:
        for transaction in transactions:
            for tx in transaction.txs:
```

(continues on next page)

(continued from previous page)

```
                if not tx.token_id in self.token_metadata:
                    raise Exception("FA2_TOKEN_UNDEFINED")
                else:
                    if not (transaction.from_ == Tezos.sender or OperatorKey(transaction.
↪from_, Tezos.sender, tx.token_id) in self.operators):
                        raise Exception("FA2_NOT_OPERATOR")

                    from_key = LedgerKey(transaction.from_, tx.token_id)
                    from_balance = self.ledger.get(from_key, Nat(0))

                    if tx.amount > from_balance:
                        raise Exception("FA2_INSUFFICIENT_BALANCE")

                    self.ledger[from_key] = abs(from_balance - tx.amount)

                    to_key = LedgerKey(tx.to_, tx.token_id)
                    self.ledger[to_key] = self.ledger.get(to_key, Nat(0)) + tx.amount

    def updateOperator(self, owner: Address, operator: Address, token_id: Nat, add_
↪operator: Boolean) -> None:
        if Tezos.sender != owner:
            raise Exception("FA2_NOT_OWNER")

        operator_key = OperatorKey(owner, operator, token_id)
        if add_operator:
            self.operators[operator_key] = Nat(0)
        #else:
        #    del self.operators[operator_key]

    def balanceOf(self, owner: Address, token_id: Nat) -> Nat:
        if not token_id in self.token_metadata:
            raise Exception("FA2_TOKEN_UNDEFINED")

        return self.ledger.get(LedgerKey(owner, token_id), Nat(0))
```

8.3 Auction

```
from pymich.michelson_types import *


class Auction(BaseContract):
    owner: Address
    top_bidder: Address
    bids: BigMap[Address, Mutez]

    def bid(self) -> None:
        if Tezos.sender in self.bids:
            raise Exception("You have already made a bid")

        self.bids[Tezos.sender] = Tezos.amount
```

(continues on next page)

(continued from previous page)

```
        if Tezos.amount > self.bids[self.top_bidder]:
            self.top_bidder = Tezos.sender

    def collectTopBid(self) -> None:
        if Tezos.sender != self.owner:
            raise Exception("Only the owner can collect the top bid")

        self.ops = self.ops.push(
            Contract[Unit](Tezos.sender),
            self.bids[self.top_bidder],
            Unit(),
        )

    def claim(self) -> None:
        if not (Tezos.sender in self.bids):
            raise Exception("You have not made any bids!")

        if Tezos.sender == self.top_bidder:
            raise Exception("You won!")

        self.ops = self.ops.push(
            Contract[Unit](Tezos.sender),
            self.bids[Tezos.sender],
            Unit(),
        )
```

8.4 Decentralized Exchange

```
# Pymich implementation of https://gitlab.com/dexter2tz/dexter2tz/-/blob/master/dexter.
↪mligo
# Also uses flat curve from https://github.com/tezos-checker/flat-cfmm/blob/master/flat_
↪cfmm.mligo

from dataclasses import dataclass
from pymich.michelson_types import *


PRICE_NUM = Nat(1)
PRICE_DENOM = Nat(1)

AMM = Nat(0)
CFMM = Nat(1)


def mutez_to_natural(a: Mutez) -> Nat:
    return a // Mutez(1)


def natural_to_mutez(a: Nat) -> Mutez:
    return a * Mutez(1)
```

(continues on next page)

(continued from previous page)

```
def ceildiv(numerator: Nat, denominator: Nat) -> Nat:
    res = Nat(0)
    if denominator == Nat(0):
        raise Exception("DIV by 0")
    else:
        q = numerator // denominator
        r = numerator % denominator
        if r == Nat(0):
            res = q
        else:
            res = q + Nat(1)
    return res


@dataclass
class TransferParam(Record):
    _from: Address
    _to: Address
    value: Nat


@dataclass
class MintOrBurn(Record):
    quantity: Int
    target: Address


def token_transfer(
        ops: Operations,
        token_address: Address,
        from_: Address,
        to: Address,
        token_amount: Nat,
    ) -> Operations:
    transfer_entrypoint = Contract[TransferParam](token_address, "%transfer")
    return ops.push(
        transfer_entrypoint,
        Mutez(0),
        TransferParam(from_, to, token_amount),
    )


def mint_or_burn(
        ops: Operations,
        lqt_address: Address,
        target: Address,
        quantity: Int,
    ) -> Operations:
    mint_or_burn_entrypoint = Contract[MintOrBurn](lqt_address, "%transfer")
    return ops.push(
```

(continues on next page)

(continued from previous page)

```
        mint_or_burn_entrypoint,
        Mutez(0),
        MintOrBurn(quantity, target),
    )


def xtz_transfer(
        ops: Operations,
        to: Address,
        amount: Mutez,
    ) -> Operations:
    return ops.push(
        Contract[Unit](to),
        amount,
        Unit(),
    )


def amm_tokens_bought(pool_in: Nat, pool_out: Nat, tokens_sold: Nat) -> Nat:
    return tokens_sold * Nat(997) * pool_out // (pool_in * Nat(1000) + (tokens_sold *␣
↪Nat(997)))

@dataclass
class Point:
    x: Nat
    y: Nat


@dataclass
class SlopeInfo:
    """
    Gives information relative to the slope of a 2D curve

    :param [x]: x coodinate at which the slope is calculated
    :param [dx_dy]: derivative of x with respect to y
    """
    x: Nat
    dx_dy: Nat


def util(p: Point) -> SlopeInfo:
    plus = p.x + p.y
    minus = p.x - p.y
    plus_2 = plus * plus
    plus_4 = plus_2 * plus_2
    plus_8 = plus_4 * plus_4
    plus_7 = plus_8 // plus
    minus_2 = minus * minus
    minus_4 = minus_2 * minus_2
    minus_8 = minus_4 * minus_4
    minus_7 = Int(0)
    if minus != Int(0):
```

(continues on next page)

(continued from previous page)

```
        minus_7 = minus_8 // minus

    return SlopeInfo(
        abs(plus_8 - minus_8),
        Nat(8) * abs(minus_7 + plus_7),
    )


@dataclass
class NewtonParam:
    x: Nat
    y: Nat
    dx: Nat
    dy: Nat
    u: Nat


def newton(param: NewtonParam) -> Nat:
    iterations = List[Nat](
        Nat(1), Nat(2), Nat(3), Nat(4),
    )
    for i in iterations:
        slope = util(Point(param.x + param.dx, abs(param.y - param.dy)))
        new_u = slope.x
        new_du_dy = slope.dx_dy
        param.dy = param.dy + abs((new_u - param.u) // new_du_dy)
    return param.dy


@dataclass
class FlatSwapParam:
    pool_in: Nat
    pool_out: Nat
    tokens_sold: Nat


def cfmm_tokens_bought(param: FlatSwapParam) -> Nat:
    x = param.pool_in * PRICE_NUM
    y = param.pool_out * PRICE_DENOM
    slope = util(Point(param.pool_in, param.pool_out))
    u = slope.x
    newton_param = NewtonParam(
        x,
        y,
        param.tokens_sold * PRICE_NUM,
        Nat(0),
        u,
    )
    return newton(newton_param)

def cfmm_xtz_bought(param: FlatSwapParam) -> Nat:
    x = param.pool_in * PRICE_DENOM
```

(continues on next page)

(continued from previous page)

```
    y = param.pool_out * PRICE_NUM
    slope = util(Point(param.pool_in, param.pool_out))
    u = slope.x
    newton_param = NewtonParam(
        x,
        y,
        param.tokens_sold * PRICE_DENOM,
        Nat(0),
        u,
    )
    return newton(newton_param)


def get_tokens_bought(curve_id: Nat, xtz_pool: Nat, token_pool: Nat, nat_amount: Nat) ->␣
↪Nat:
    if curve_id == AMM:
        return amm_tokens_bought(xtz_pool, token_pool, nat_amount)
    else:
        return cfmm_tokens_bought(FlatSwapParam(xtz_pool, token_pool, nat_amount))

def get_xtz_bought(curve_id: Nat, token_pool: Nat, xtz_pool: Nat, nat_amount: Nat) ->␣
↪Nat:
    if curve_id == AMM:
        return amm_tokens_bought(token_pool, xtz_pool, nat_amount)
    else:
        return cfmm_tokens_bought(FlatSwapParam(token_pool, xtz_pool, nat_amount))


@dataclass(kw_only=True)
class Dexter(BaseContract):
    token_pool: Nat
    xtz_pool: Mutez
    lqt_total: Nat
    token_address: Address
    lqt_address: Address
    curve_id: Nat

    def add_liquidity(
            self,
            owner: Address,
            min_lqt_minted: Nat,
            max_tokens_deposited: Nat,
            deadline: Timestamp,
    ) -> None:
        if Timestamp.now() >= deadline:
            raise Exception("The current time must be less than the deadline.")
        else:
            xtz_pool = mutez_to_natural(self.xtz_pool)
            nat_amount = mutez_to_natural(Tezos.amount)
            lqt_minted = nat_amount * self.lqt_total // xtz_pool
            tokens_deposited = ceildiv(nat_amount * self.token_pool, xtz_pool)
```

(continues on next page)

(continued from previous page)

```
            if tokens_deposited > max_tokens_deposited:
                raise Exception("Max tokens deposited must be greater than or equal to␣
↪tokens deposited")
            elif lqt_minted < min_lqt_minted:
                raise Exception("Lqt minted must be greater than min lqt minted")
            else:
                self.lqt_total = self.lqt_total + lqt_minted
                self.token_pool = self.token_pool + tokens_deposited
                self.xtz_pool = self.xtz_pool + Tezos.amount

                self.ops = token_transfer(self.ops, self.token_address, Tezos.sender,␣
↪Tezos.self_address, tokens_deposited)
                self.ops = mint_or_burn(self.ops, self.lqt_address, owner, lqt_minted.to_
↪int())

    def remove_liquidity(
        self,
        to: Address,
        lqt_burned: Nat,
        min_xtz_withdrawn: Mutez,
        min_tokens_withdrawn: Nat,
        deadline: Timestamp,
    ) -> None:
        if Timestamp.now() >= deadline:
            raise Exception("The current time must be less than the deadline")
        elif Tezos.amount > Mutez(0):
            raise Exception("Amount must be zero")
        else:
            xtz_withdrawn = natural_to_mutez(lqt_burned * mutez_to_natural(self.xtz_
↪pool) // self.lqt_total)
            tokens_withdrawn = lqt_burned * self.token_pool // self.lqt_total

            if xtz_withdrawn < min_xtz_withdrawn:
                raise Exception("The amount of xtz withdrawn must be greater than or␣
↪equal to min xtz withdrawn")
            elif tokens_withdrawn < min_tokens_withdrawn:
                raise Exception("The amount of tokens withdrawn must be greater than or␣
↪equal to min tokens withdrawn")
            else:
                self.lqt_total = (self.lqt_total - lqt_burned).is_nat().get("Cannot burn␣
↪more than the total amount of lqt")
                self.token_pool = (self.token_pool - tokens_withdrawn).is_nat().get(
↪"Token pool minus tokens withdrawn is negative")

                self.ops = mint_or_burn(self.ops, self.lqt_address, Tezos.sender,␣
↪(Nat(0) - lqt_burned))
                self.ops = token_transfer(self.ops, self.token_address, Tezos.self_
↪address, to, tokens_withdrawn)
                self.ops = xtz_transfer(self.ops, to, xtz_withdrawn)

    def xtz_to_token(self, to: Address, min_tokens_bought: Nat, deadline: Timestamp) ->␣
↪None:
```

(continues on next page)

(continued from previous page)

```
        if Timestamp.now() >= deadline:
            raise Exception("The current time must be less than the deadline")
        else:
            # we don't check that xtz_pool > 0, because that is impossible
            # unless all liquidity has been removed
            xtz_pool = mutez_to_natural(self.xtz_pool)
            nat_amount = mutez_to_natural(Tezos.amount)
            tokens_bought = get_tokens_bought(self.curve_id, xtz_pool, self.token_pool,␣
↪nat_amount)
            if tokens_bought < min_tokens_bought:
                raise Exception("Tokens bought must be greater than or equal to min␣
↪tokens bought")
            self.token_pool = (self.token_pool - tokens_bought).is_nat().get("Token pool␣
↪minus tokens bought is negative")
            self.xtz_pool = self.xtz_pool + Tezos.amount
            self.ops = token_transfer(self.ops, self.token_address, Tezos.self_address,␣
↪to, tokens_bought)

    def token_to_xtz(self, to: Address, tokens_sold: Nat, min_xtz_bought: Mutez,␣
↪deadline: Timestamp) -> None:
        if Timestamp.now() >= deadline:
            raise Exception("The current time must be less than the deadline")
        elif Tezos.amount > Mutez(0):
            raise Exception("Amount must be zero")
        else:
            # we don't check that tokenPool > 0, because that is impossible
            # unless all liquidity has been removed
            xtz_bought = natural_to_mutez(get_xtz_bought(self.curve_id, self.token_pool,␣
↪mutez_to_natural(self.xtz_pool), tokens_sold))
            if xtz_bought < min_xtz_bought:
                raise Exception("Xtz bought must be greater than or equal to min xtz␣
↪bought")
            self.ops = token_transfer(self.ops, self.token_address, Tezos.sender, Tezos.
↪self_address, tokens_sold)
            self.ops = xtz_transfer(self.ops, to, xtz_bought)
            self.token_pool = self.token_pool + tokens_sold
            self.xtz_pool = (self.xtz_pool - xtz_bought).get("negative mutez")
```

8.5 Election

```
from dataclasses import dataclass
from pymich.michelson_types import *


def require(condition: Boolean, message: String) -> Nat:
    if not condition:
        raise Exception(message)

    return Nat(0)
```

(continues on next page)

(continued from previous page)

```
@dataclass(kw_only=True)
class Election(BaseContract):
    admin: Address
    manifest_url: String
    manifest_hash: String
    _open: String
    _close: String
    artifacts_url: String
    artifacts_hash: String

    def open(self, _open: String, manifest_url: String, manifest_hash: String) -> None:
        require(Tezos.sender == self.admin, String("Only admin can call this entrypoint
↪"))
        self._open = _open
        self.manifest_url = manifest_url
        self.manifest_hash = manifest_hash

    def close(self, _close: String) -> None:
        require(Tezos.sender == self.admin, String("Only admin can call this entrypoint
↪"))
        self._close = _close

    def artifacts(self, artifacts_url: String, artifacts_hash: String) -> None:
        require(Tezos.sender == self.admin, String("Only admin can call this entrypoint
↪"))
        self.artifacts_url = artifacts_url
        self.artifacts_hash = artifacts_hash
```

8.6 Escrow

```
from dataclasses import dataclass
from pymich.michelson_types import *


@dataclass(kw_only=True)
class Escrow(BaseContract):
    seller: Address
    buyer: Address
    price: Mutez
    paid: Boolean
    confirmed: Boolean

    def pay(self) -> None:
        if Tezos.sender != self.buyer:
            raise Exception("You are not the seller")

        if Tezos.amount != self.price:
            raise Exception("Not the right price")
```

(continues on next page)

(continued from previous page)

```
        if self.paid:
            raise Exception("You have already paid!")

        self.paid = True

    def confirm(self) -> None:
        if Tezos.sender != self.buyer:
            raise Exception("You are not the buyer")

        if not self.paid:
            raise Exception("You have not paid")

        if self.confirmed:
            raise Exception("Already confirmed")

        self.confirmed = True

    def claim(self) -> None:
        if Tezos.sender != self.seller:
            raise Exception("You are not the seller")

        if not self.confirmed:
            raise Exception("Not confirmed")

        self.ops = self.ops.push(Contract[Unit](Tezos.sender), Tezos.balance, Unit())
```

8.7 Lottery

```
from dataclasses import dataclass

from pymich.michelson_types import *
from pymich.stdlib import *


@dataclass
class BidInfo(Record):
    value_hash: Bytes
    num_bid: Nat


@dataclass(kw_only=True)
class Lottery(BaseContract):
    bid_amount: Mutez
    deadline_bet: Timestamp
    deadline_reveal: Timestamp
    bids: BigMap[Address, BidInfo]
    nb_bids: Nat
    nb_revealed: Nat
    sum_values: Nat
```

(continues on next page)

(continued from previous page)

```
    def bet(self, value_hash: Bytes) -> None:
        if Tezos.sender in self.bids:
            raise Exception("You have already made a bid")

        if Tezos.amount != self.bid_amount:
            raise Exception("You have not bidded the right amount")

        if Timestamp.now() > self.deadline_bet:
            raise Exception("Too late to make a bid")

        self.bids[Tezos.sender] = BidInfo(value_hash, self.nb_bids)
        self.nb_bids = self.nb_bids + Nat(1)

    def reveal(self, value: Nat) -> None:
        if not (Tezos.sender in self.bids):
            raise Exception("You have not made a bid")

        if Timestamp.now() > self.deadline_bet or Timestamp.now() > self.deadline_reveal:
            raise Exception("Too late to make a reveal")

        if Bytes(value).blake2b() != self.bids[Tezos.sender].value_hash:
            raise Exception("Wrong hash")

        self.sum_values = self.sum_values + value
        self.nb_revealed = self.nb_revealed + Nat(1)

    def claim(self) -> None:
        if Timestamp.now() < self.deadline_reveal:
            raise Exception("The lottery is not over")

        if not (Tezos.sender in self.bids):
            raise Exception("You have not made a bid")

        num_winner = self.sum_values % self.nb_revealed

        if self.bids[Tezos.sender].num_bid != num_winner:
            raise Exception("You have not won")

        transaction(Contract[Unit](Tezos.sender), Tezos.balance, Unit())
```

8.8 Notarization

```
from dataclasses import dataclass
from pymich.michelson_types import *


@dataclass(eq=False)
class DocumentId(Record):
    owner: Address
    uuid: String
```

(continues on next page)

(continued from previous page)

```
@dataclass(kw_only=True)
class Notarization(BaseContract):
    admin: Address
    document_hashes: BigMap[DocumentId, Bytes]

    def add_document(self, document_uuid: String, document_hash: Bytes) -> None:
        self.document_hashes[DocumentId(Tezos.sender, document_uuid)] = document_hash

    def get_hash(self, document_uuid: String, owner: Address) -> Bytes:
        return self.document_hashes[DocumentId(owner, document_uuid)]
```

8.9 Upgradable contract

```
from dataclasses import dataclass

from pymich.michelson_types import *
from typing import Callable


@dataclass(kw_only=True)
class Upgradable(BaseContract):
    counter: Nat
    f: Callable[[Nat], Nat]

    def update_f(self, f: Callable[[Nat], Nat]) -> None:
        self.f = f

    def update_counter(self, x: Nat) -> None:
        self.counter = self.f(x)
```

8.10 Visitor

```
from dataclasses import dataclass
from pymich.michelson_types import *


@dataclass
class VisitorInfo(Record):
    visits: Nat
    name: String
    last_visit: Timestamp


@dataclass(kw_only=True)
class Visitor(BaseContract):
```

(continues on next page)

(continued from previous page)

```
    visitors: BigMap[Address, VisitorInfo]
    total_visits: Nat

    def register(self, name: String) -> None:
        self.visitors[Tezos.sender] = VisitorInfo(Nat(0), name, Timestamp.now())

    def visit(self) -> None:
        if not (Tezos.sender in self.visitors):
            raise Exception("You are not registered yet!")

        n_visits = self.visitors[Tezos.sender].visits

        if Timestamp.now() - self.visitors[Tezos.sender].last_visit < Int(10) * Int(24)␣
↪* Int(3600):
            raise Exception("You need to wait 10 days between visits")

        if n_visits == Nat(0) and Tezos.amount != Mutez(5):
            raise Exception("You need to pay 5 mutez on your first visit!")

        if n_visits != Nat(0) and Tezos.amount != Mutez(3):
            raise Exception("You need to pay 3 mutez to visit!")

        self.visitors[Tezos.sender].visits = n_visits + Nat(1)
        self.total_visits = self.total_visits + Nat(1)
```

CHAPTER

NINE

COMPILER ARCHITECTURE

This document goes over the Pymich compiler internals. We first go over its high-level architecture, before diving it deeper to provide the reader with the information necessary start hacking on the codebase.

9.1 High level architecture

The Pymich compiler is articulated over three components:

- a **frontend**, that compiles from Pymich code to its *intermediate-representation* (Pymich-IR).
- a **middlend**, that transforms Pymich-IR into a more optimized Pymich-IR.
- a **backend**, that compiles from the Pymich-IR to Michelson instructions ready

The compiler source is organized as follows:

9.2 Pymich Intermediate-Representation

Let us start by going over the Pymich-IR, heavily inspired from Albert since it is the only *language* involved in all three components of the compiler. However, it is not quite as low level and leaves variable dealocations to the backend compiler.

The main goal of this IR is to provide the middle-end with a simpler representation to optimize over, and allow the backend to generate Michelson code in a simpler way. The IR provides the following grammar:

```
python_types     ::=
                   | python_number
                   | python_string

base_types       ::=
                   | Unit
                   | String
                   | Nat
                   | Int
                   | Boolean
                   | Address

type             ::=
                   | base_type
                   | Operation
                   | Contract
```

(continues on next page)

(continued from previous page)

```
                    | Map[type, type]
                    | BigMap[type, type]
                    | List[type]
                    | Set[type]
                    | Record[attr1=type, ..., attrn=type]

base_object       ::=
                    | base_type(python_type)

atom              ::=
                    | var
                    | base_object

lhs               ::=
                    | var
                    | record.attribute
                    | map[atom]
                    | big_map[atom]

rhs               ::=
                    | atom
                    | f(atom)
                    | record.attribute
                    | Map[type, type]()
                    | BigMap[type, type]()
                    | List[type]()
                    | Set[type]()
                    | var.unpack[type]()

instruction       ::=
                    | rhs
                    | lhs = rhs
block             ::=
                    | instruction
                    | instruction
                      instruction

loops             ::=
                    | for var in rhs:
                          block

 condition        ::=
                    | if var:
                          block
                      else:
                          block

 exception        ::=
                    | raise Exception(python_string)
```

Entrypoints are factored in a class inheriting from `pymich.michelson_types.Contract` class and all of this class's methods are compiled to entrypoints. Entrypoints take exactly **one** argument and return the new storage. Although the operations are currently returned implicitely by the compiler at the end of every entrypoints, this will most likely be

updated in the near future and the Pymich-IR will require functions returning a `Pair(List[Operation], Storage)`. Finally, in Pymich-IR, all contracts must implement a :python`get_storage_type()` method that returns the storage type.

All functions take exactly **one** argument.

Here is an example of the FA1.2 mint function that was implemented in the getting-started section:

```
class AllowanceKey:
    owner: Address
    spender: Address

class Storage:
    tokens: BigMap[Address, Nat]
    allowances: BigMap[AllowanceKey, Nat]
    total_supply: Nat
    owner: Address

class mintParam:
    _to: Address
    value: Nat

class FA12(Contract):

    def get_storage_type():
        return Storage

    def mint(mint__param: mintParam) -> Storage:
        _to = mint__param._to
        value = mint__param.value
        tmp_var_6 = __STORAGE__.owner
        tmp_var_7 = SENDER != tmp_var_6
        if tmp_var_7:
            raise Exception('Only owner can mint')
        tmp_var_8 = __STORAGE__.total_supply
        __STORAGE__.total_supply = tmp_var_8 + value
        tmp_var_0 = __STORAGE__.tokens
        tmp_var_9 = __STORAGE__.tokens
        tmp_var_10 = tmp_var_9.get
        tmp_var_11 = tmp_var_10(_to, Nat(0))
        tmp_var_0[_to] = tmp_var_11 + value
        __STORAGE__.tokens = tmp_var_0
        return __STORAGE__
```

9.3 Frontend

The goal of the frontend compiler is to go from the subset of python that Pymich supports to the Pymich-IR such that the middle end can then optimize the IR before the backend emits the Michelson.

The passes go as follow (see the frontend compiler code:

1. rewrite pre-hangzhou view functions;
2. factor out the storage into its own dataclass and introduce the `get_storage_type` method;
3. remove `self` from contract method arguments;

4. tuplify function arguments;
5. handle entrypoint and functions with no arguments (convert then to single argument functions of type `Unit`;
6. three address code the python code according to the Pymich-IR grammar.

Supporting a new Python syntax such as the currently unsupported += operator is then as easy as adding a frontend compiler pass that transforms `a += b` into `a = a + b` and letting the middlend and backend do the rest.

Note that the IR is not yet in static-single assignment form due to the difficulty to compile the **phi** function in the backend compiler. Work is currently undergoing to introduce it. We are currently discussing with Nomadics Labs how to best introduce it (or the equivalent *continuous passing style* representation).

Finally, the Python typechecking occurs at the Pymich-IR after all passes have been executed. This allows to easily reuse the typechecking routine after each middlend passes.

9.4 Middlend

The goal for the middlend is to convert, through successive passes, Pymich-IR to a more optimized Pymich-IR.

Although the Middlend is currently not implemented, the main goal in the next few months is to introduce one, since we have simple enough Pymich-IR that optimizations are possible. The optimizations that will be first implemented are the following:

- constant folding
- constant propagation
- dead code elimination
- inlining function that are only use a single time
- common subexpression elimination
- loop invariant code motion

In order to ensure that the program still typechecks after the optimizations, the Pymich-IR typechecker is ran after all optimization passes have successfully executed.

9.5 Backend

Finally, the compiler backend emits Michelson code from Pymich-IR to Michelson. Besides the obvious conversions from Pymich-IR data structure to the Michelson ones, the compiler backend is responsible for allocating and deallocating variables when necessary.

Here are some links to the implementations:

- IR to Michelson compiler
- IR typechecker
- Michelson typechecker
- Compiler integration

Note that although the typechecker is currently used which compiling from the IR to Michelson (link from the first bullet above), it is planned to refactor it into a separate pass that can be called after the frontend compiler, between each optimization pass, and before the backend compiler.

9.6 Error Handling

Exceptions raised by the compiler stages are available here. They each take a line number that is preserved from Pymich code to the IR via `ast.fix_missing_locations(transformed_ast)`.

In the following example, we use an undefined variable `g`:

```
from pymich.michelson_types import *
from typing import Callable


class Contract:
    counter: Nat
    f: Callable[[Nat], Nat]

    def update_f(self, f: Callable[[Nat], Nat]):
        self.f = g  # Error, `g` is undefined

    def update_counter(self, x: Nat):
        self.counter = self.f(x)
```

Which returns as expected the undefined variable error with the correct line number:

```
Function 'g' does not exist at line 10
```

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

G

I

L

M

N

O

R

S

T

U