Compiling Juvix to Cairo

Lukasz Czajka

Juvix

- Purely functional programming language for intent-centric privacy-preserving applications on Anoma.

Juvix

- Purely functional programming language for intent-centric privacy-preserving applications on Anoma.
- Strong static typing.

Juvix

- Purely functional programming language for intent-centric privacy-preserving applications on Anoma.
- Strong static typing.
- Similar to OCaml, Haskell, Lean, Agda, ...

Juvix

- Purely functional programming language for intent-centric privacy-preserving applications on Anoma.
- Strong static typing.
- Similar to OCaml, Haskell, Lean, Agda, ...
- Seamless integration of different compilation targets.

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.
- Multiple backends:

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.
- Multiple backends:
  - transparent Anoma VM,

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.
- Multiple backends:
  - transparent Anoma VM,
  - native code and WASM via C,

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.
- Multiple backends:
  - transparent Anoma VM,
  - native code and WASM via C,
  - zkVMs: Cairo, RISC0,

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.
- Multiple backends:
    - transparent Anoma VM,
    - native code and WASM via C,
    - zkVMs: Cairo, RISC0,
    - formerly: zkLLVM, VampIR,

Why Juvix?

- For business logic. The rest is handled by the Anoma OS.
- Declarative functional – matches the Anoma architecture (resource logics, transaction functions).
- Static typing, formal guarantees.
- Relatively simple.
- Multiple backends:
  - transparent Anoma VM,
  - native code and WASM via C,
  - zkVMs: Cairo, RISC0,
  - formerly: zkLLVM, VampIR,
  - comparatively easy to add new backends.

Example Juvix program

```
import Stdlib.Data.Fixity open;

type List A :=
  | nil
  | :: A (List A);

open List;

syntax operator :: cons;

map {A B} (f : A -> B) : List A -> List B
  | nil := nil
  | (x :: xs) := f x :: map f xs;
```

Cairo

- A practically-efficient Turing-complete STARK-friendly CPU architecture for generating zero-knowledge proofs of program execution integrity.

Cairo

- A practically-efficient Turing-complete STARK-friendly CPU architecture for generating zero-knowledge proofs of program execution integrity.
- Low-level imperative assembly.

Cairo

- A practically-efficient Turing-complete STARK-friendly CPU architecture for generating zero-knowledge proofs of program execution integrity.
- Low-level imperative assembly.
- RISC architecture.

Cairo

- A practically-efficient Turing-complete STARK-friendly CPU architecture for generating zero-knowledge proofs of program execution integrity.
- Low-level imperative assembly.
- RISC architecture.
- Memory is read-only!

Cairo

- A practically-efficient Turing-complete STARK-friendly CPU architecture for generating zero-knowledge proofs of program execution integrity.
- Low-level imperative assembly.
- RISC architecture.
- Memory is read-only!
- No “assignment” – only equality assertions.

Cairo

- A practically-efficient Turing-complete STARK-friendly CPU architecture for generating zero-knowledge proofs of program execution integrity.
- Low-level imperative assembly.
- RISC architecture.
- Memory is read-only!
- No "assignment" – only equality assertions.
- Memory accesses required to be continuous (at increasing addresses without gaps).

Example Cairo Assembly program

Computing the factorial of 10.

```
start:
  [ap] = 10
  [ap + 1] = 1
  ap += 2
loop:
  [ap] = [ap - 2] - 1
  [ap + 1] = [ap - 1] * [ap - 2]
  ap += 2
  jmp loop if [ap - 2] != 0
```

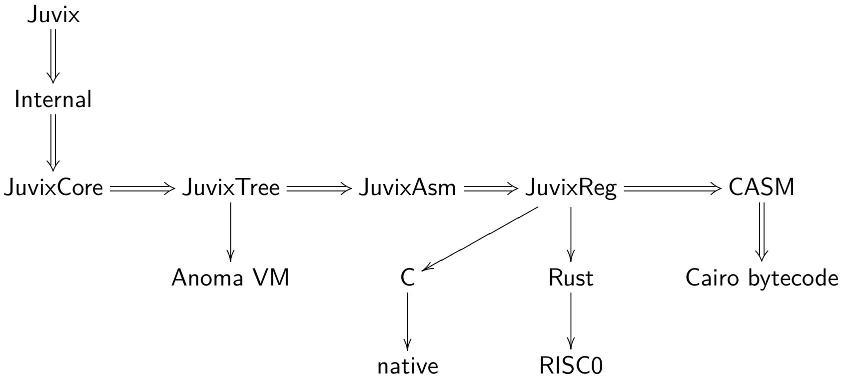
Example Cairo Assembly program

Computing the factorial of 10.

```
start:
  [ap] = 10
  [ap + 1] = 1
  ap += 2
loop:
  [ap] = [ap - 2] - 1
  [ap + 1] = [ap - 1] * [ap - 2]
  ap += 2
  jmp loop if [ap - 2] != 0
```

Result available in `[ap - 1]`.

Juvix compilation pipeline



JuvixCore

Minimalistic functional IR based on the lambda-calculus.

JuvixCore

Minimalistic functional IR based on the lambda-calculus.

JuvixCore expressions

- Application: `t s`.

JuvixCore

Minimalistic functional IR based on the lambda-calculus.

JuvixCore expressions

- Application: `t s`.
- Lambda: `\(x : T) t`.

JuvixCore

Minimalistic functional IR based on the lambda-calculus.

JuvixCore expressions

- Application: `t s`.
- Lambda: `\(x : T) t`.
- Let: `let x : T := t in s`.

JuvixCore

Minimalistic functional IR based on the lambda-calculus.

JuvixCore expressions

- Application: `t s`.
- Lambda: `\(x : T) t`.
- Let: `let x : T := t in s`.
- Match:

```
match t : T with : T' {
  pattern1 := b1;
  pattern2 := b2
}
```

JuvixCore

Minimalistic functional IR based on the lambda-calculus.

JuvixCore expressions

- Application: `t s`.
- Lambda: `\(x : T) t`.
- Let: `let x : T := t in s`.
- Match:

```
match t : T with : T' {
  pattern1 := b1;
  pattern2 := b2
}
```

- Case:

```
case t of {
  c1 x1 x2 := b1;
  c2 x1 x2 x3 := b2
}
```

JuvixCore: Example program

```
map {A B} (f : A -> B) : List A -> List B
  | nil := nil
  | (cons x xs) := cons (f x) (map f xs);
```

JuvixCore: Example program

```
map {A B} (f : A -> B) : List A -> List B
  | nil := nil
  | (cons x xs) := cons (f x) (map f xs);
```

```
def map : Π A : Type, Π B : Type,
          (A -> B) -> List A -> List B :=
  \(A : Type) \(B : Type) \(f : A -> B) \(_X : List A)
    match (_X : List A) with : (List B) {
      (nil (A' : Type)) := nil B;
      (cons (A' : Type) (x : A) (xs : List A)) :=
        cons B (f x) (map A B f xs)
    };
```

JuvixCore transformations

- *Pattern matching compilation* converts complex pattern matches into one-level case-expressions.

JuvixCore transformations

- *Pattern matching compilation* converts complex pattern matches into one-level case-expressions.
- *Lambda-lifting* removes anonymous and local functions.

JuvixCore transformations

- *Pattern matching compilation* converts complex pattern matches into one-level case-expressions.
- *Lambda-lifting* removes anonymous and local functions.
- *Type erasure* removes runtime type information.

JuvixCore: Example program

```
def map : Π A : Type, Π B : Type,
          (A -> B) -> List A -> List B :=
  \(A : Type) \(B : Type) \(f : A -> B) \(l : List A)
    match (l : List A) with : (List B) {
      (nil (A' : Type)) := nil B;
      (cons (A' : Type) (x : A) (xs : List A)) :=
        cons B (f x) (map A B f xs)
    };
```

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

Stripped JuvixCore

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

Stripped JuvixCore

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

- All functions are defined at the top level (no nested or anonymous functions).

Stripped JuvixCore

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

- All functions are defined at the top level (no nested or anonymous functions).
- Functions are simply-typed with the number of arguments (top lambda-abstractions) matching the type.

Stripped JuvixCore

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

- All functions are defined at the top level (no nested or anonymous functions).
- Functions are simply-typed with the number of arguments (top lambda-abstractions) matching the type.
- All application expressions have the form $f t_1 \ldots t_n$ where f is a function name, a variable or a constructor.

Stripped JuvixCore

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

- All functions are defined at the top level (no nested or anonymous functions).
- Functions are simply-typed with the number of arguments (top lambda-abstractions) matching the type.
- All application expressions have the form $f t_1 \ldots t_n$ where f is a function name, a variable or a constructor.
- Polymorphic arguments have the $*$ type.

JuvixTree

Applicative functional IR with explicit closure operations and uncurried top-level functions.

JuvixTree

Applicative functional IR with explicit closure operations and uncurried top-level functions.

JuvixTree expressions

- Reference the `nth` function argument: `arg[n]`.

JuvixTree

Applicative functional IR with explicit closure operations and uncurried top-level functions.

JuvixTree expressions

- Reference the `nth` function argument: `arg[n]`.
- Push value onto the temporary stack: `save(t) { expr }`.

JuvixTree

Applicative functional IR with explicit closure operations and uncurried top-level functions.

JuvixTree expressions

- Reference the nth function argument: `arg[n]`.
- Push value onto the temporary stack: `save(t) { expr }`.
- Reference the nth cell from the bottom in the temporary stack: `tmp[n]`.

JuvixTree

Applicative functional IR with explicit closure operations and uncurried top-level functions.

JuvixTree expressions

- Reference the `n`th function argument: `arg[n]`.
- Push value onto the temporary stack: `save(t) { expr }`.
- Reference the `n`th cell from the bottom in the temporary stack: `tmp[n]`.
- Branch on a boolean:

```
br(t) {
  true: expr1
  false: expr2
}
```

JuvixTree

JuvixTree expressions

- Constructor data (tag + arguments) allocation: `alloc[ctr](t1, ..., tn)`.

JuvixTree

JuvixTree expressions

- Constructor data (tag + arguments) allocation:
  `alloc[ctr](t1, ..., tn)`.
- Branch on a constructor tag:

```
case[List](l) {
  nil: expr1
  cons: expr2
}
```

JuvixTree

JuvixTree expressions

- Constructor data (tag + arguments) allocation: `alloc[ctr](t1, ..., tn)`.
- Branch on a constructor tag:

```
case[List](l) {
  nil: expr1
  cons: expr2
}
```

- Reference the kth constructor argument: `r.ctr[k]` where `r` is `arg[n]` or `tmp[n]`, and `ctr` is a constructor.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.

JuvixTree

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
    - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.
  - For `n < argsnum(t)`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.
  - For `n < argsnum(t)`.
- Closure call: `call(t, t1, ..., tn)`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.
  - For `n < argsnum(t)`.
- Closure call: `call(t, t1, ..., tn)`.
  - For `n = argsnum(t)`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.
  - For `n < argsnum(t)`.
- Closure call: `call(t, t1, ..., tn)`.
  - For `n = argsnum(t)`.
- What to do when `n > argsnum(t)`?

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.
  - For `n < argsnum(t)`.
- Closure call: `call(t, t1, ..., tn)`.
  - For `n = argsnum(t)`.
- What to do when `n > argsnum(t)`?
  - Overapplication: `id f a : B` for `id : * -> *`, `f : A -> B` and `a : A`.

JuvixTree expressions

- Function call (known `f`): `call[f](t1, ..., tn)`.
- Closure (function code + environment) allocation: `calloc[f](t1, ..., tn)`.
  - Partial application: `f a : B -> C` for `f : A -> B -> C` and `a : A`.
- Closure arguments number: `argsnum(t)`.
  - Impossible to statically determine the number of arguments an unknown function expects: both `map f l` and `map (f a) l'` are okay for `map : (* -> *) -> List -> List`.
- Closure extension: `cextend(t, t1, ..., tn)`.
  - For `n < argsnum(t)`.
- Closure call: `call(t, t1, ..., tn)`.
  - For `n = argsnum(t)`.
- What to do when `n > argsnum(t)`?
  - Overapplication: `id f a : B` for `id : * -> *`, `f : A -> B` and `a : A`.
  - Call closure with first `argsnum(t)` arguments and repeat.

Translation from Stripped JuvixCore to JuvixTree

- *Application translation* selects the right JuvixTree operation for each JuvixCore application (direct call, static or dynamic closure call, closure allocation or extension, constructor data allocation).

Translation from Stripped JuvixCore to JuvixTree

- *Application translation* selects the right JuvixTree operation for each JuvixCore application (direct call, static or dynamic closure call, closure allocation or extension, constructor data allocation).
- *Dynamic closure call compilation* generates efficient code for call sites with possible partial application or overapplication.

JuvixTree: Example program

```
def map : (* -> *) -> List -> List :=
  \(f : * -> *) \(l : List)
    case l of {
      nil := nil;
      cons x xs := cons (f x) (map f xs)
    };
```

```
function map(* -> *, List) : List {
  save(arg[1]) {
    case[List](tmp[0]) {
      nil: alloc[nil]()
      cons:
        alloc[cons](
          call[juvix_apply_1](arg[0], tmp[0].cons[0]),
          call[map](arg[0], tmp[0].cons[1])
        )
    }
  }
```

JuvixTree: Example program

```
function juvix_apply_1(*, *) : * {
  br(eq(1, argsnum(arg[0]))) {
    true: call(arg[0], arg[1])
    false: cextend(arg[0], arg[1])
  }
}

function map(* -> *, List) : List {
  save(arg[1]) {
    case[List](tmp[0]) {
      nil: alloc[nil]()
      cons:
        alloc[cons](
          call[juvix_apply_1](arg[0], tmp[0].cons[0]),
          call[map](arg[0], tmp[0].cons[1])
        )
    }
  }
```

Application functions

```
function juvix_apply_2(*, *, *) : * {
  save(argsnum(arg[0])) {
    br(eq(2, tmp[0])) {
      true: call(arg[0], arg[1], arg[2])
      false: br(eq(1, tmp[0])) {
        true: call[juvix_apply_1](
                call(arg[0], arg[1]),
                arg[2]
              )
        false: cextend(arg[0], arg[1], arg[2])
      }
    }
  }
}
```

JuvixAsm

- Stack-based imperative assembly language suitable as an IR for eager purely functional languages.

JuvixAsm

- Stack-based imperative assembly language suitable as an IR for eager purely functional languages.
- The translation to JuvixAsm linearizes JuvixTree expressions into sequences of stack-based instructions.

JuvixAsm

- Stack-based imperative assembly language suitable as an IR for eager purely functional languages.
- The translation to JuvixAsm linearizes JuvixTree expressions into sequences of stack-based instructions.
- *Value stack* stores intermediate computation results. JuvixAsm instructions typically pop their arguments from the value stack and push the result on the top.

JuvixAsm: Example program

```
function map(* -> *, List) : List {
  push arg[1];
  case List {
    nil: { pop; alloc nil; ret; };
    cons: {
      tsave {
        push tmp[0].cons[1];
        push arg[0];
        call map;
        push tmp[0].cons[0];
        push arg[0];
        call juvix_apply_1;
        alloc cons;
        ret;
      };
    };
  };
};
```

JuvixReg

Three-address code representation of JuvixAsm using local variables instead of the value stack.

JuvixReg: Example program

```
function map(* -> *, List) : List {
  case[List] arg[1] {
    nil: {
      tmp[1] = alloc nil ();
      ret tmp[1];
    };
    cons: {
      tmp[1] = arg[1].cons[1];
      tmp[1] = call map (arg[0], tmp[1]);
      tmp[2] = arg[1].cons[0];
      tmp[2] = call juvix_apply_1 (arg[0], tmp[2]);
      tmp[1] = alloc cons (tmp[2], tmp[1]);
      ret tmp[1];
    };
  };
}
```

JuvixReg transformations

- *Static Single-Assignment form* (SSA) transformation ensures that each local variable is assigned only once.

JuvixReg transformations

- *Static Single-Assignment form* (SSA) transformation ensures that each local variable is assigned only once.
  - Necessary because Cairo memory is read-only.

JuvixReg transformations

- *Static Single-Assignment form* (SSA) transformation ensures that each local variable is assigned only once.
  - Necessary because Cairo memory is read-only.
- *Basic block computation* with live variable analysis is a prerequisite for handling the continuity requirement of Cairo memory access.

CASM

Cairo Assembly is a textual representation of Cairo bytecode.

CASM: Example program

```
map:
  jmp rel [[fp - 4]]
  jmp label_16
label_17: -- cons case
  [ap] = [[fp - 4] + 2]; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp - 3]; ap++
  call map
  ... -- omitted code
  ret
label_16: -- nil case
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 1; ap++
  ... -- omitted code
  ret
```

CASM: Example program (full listing)

```
map:
  jmp rel [[fp - 4]]
  jmp label_16
  [ap] = [[fp - 4] + 2]; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp - 3]; ap++
  call map
  [ap] = [fp - 4]; ap++
  [ap] = [fp - 3]; ap++
  call rel 3
  ret
  [ap] = [[fp - 4] + 1]; ap++
  [ap] = [fp - 6]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp - 3]; ap++
  call juvix_apply_1
  [ap] = [fp - 5]; ap++
  call rel 3
  ret
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 3; ap++
  [ap] = [fp - 4]; ap++
  [ap] = [fp - 3]; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp + 4]; ap++
  ret
label_16:
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 1; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp + 4]; ap++
  ret
```

Summary

- Juvix as a universal zkVM front-end language.

Summary

- Juvix as a universal zkVM front-end language.
- Compilation pipeline details in the ART report: https://doi.org/10.5281/zenodo.13739344.

Summary

- Juvix as a universal zkVM front-end language.
- Compilation pipeline details in the ART report: https://doi.org/10.5281/zenodo.13739344.
- You can play with the IRs yourself.

Summary

- Juvix as a universal zkVM front-end language.
- Compilation pipeline details in the ART report: https://doi.org/10.5281/zenodo.13739344.
- You can play with the IRs yourself.
  - Install Juvix (https://juvix.org)

Summary

- Juvix as a universal zkVM front-end language.
- Compilation pipeline details in the ART report: https://doi.org/10.5281/zenodo.13739344.
- You can play with the IRs yourself.
  - Install Juvix (https://juvix.org)
  - `juvix dev compile TARGET file.juvix`

Summary

- Juvix as a universal zkVM front-end language.
- Compilation pipeline details in the ART report: https://doi.org/10.5281/zenodo.13739344.
- You can play with the IRs yourself.
  - Install Juvix (https://juvix.org)
  - `juvix dev compile TARGET file.juvix`
  - `juvix dev TARGET --help`

Summary

- Juvix as a universal zkVM front-end language.
- Compilation pipeline details in the ART report: https://doi.org/10.5281/zenodo.13739344.
- You can play with the IRs yourself.
  - Install Juvix (https://juvix.org)
  - `juvix dev compile TARGET file.juvix`
  - `juvix dev TARGET --help`
  - `TARGET` is one of `core`, `tree`, `asm`, `reg`, `casm`