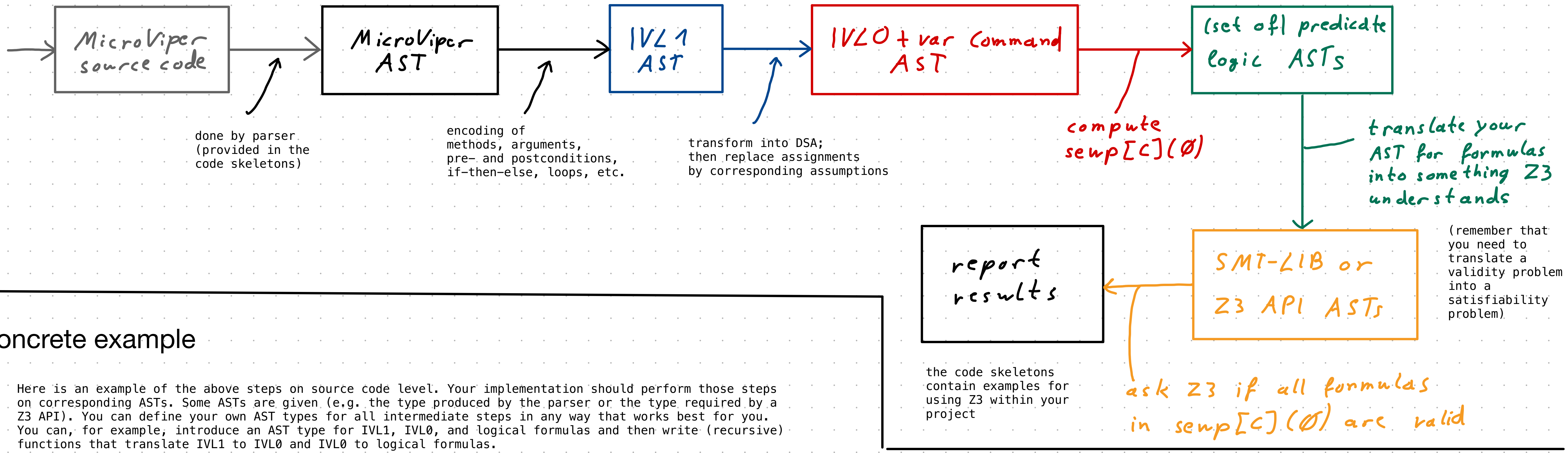
Sketch of one possible approach to get started with project A

Main idea: verification by compilation, that is, translate microViper ASTs (abstract syntax trees) obtained from a microViper Parser into ASTs of verification conditions that you can pass to Z3 via multiple translation steps as discussed in modules 1 – 4.



MicroViper source code

MicroViper AST

IVL1 AST

IVL0 + var Command AST

(set of) predicate logic ASTs

SMT-LIB or Z3 API ASTs

report results

done by parser (provided in the code skeletons)

encoding of methods, arguments, pre- and postconditions, if-then-else, loops, etc.

transform into DSA; then replace assignments by corresponding assumptions

compute sewp[C](∅)

translate your AST for formulas into something Z3 understands

(remember that you need to translate a validity problem into a satisfiability problem)

ask Z3 if all formulas in sewp[C](∅) are valid

the code skeletons contain examples for using Z3 within your project

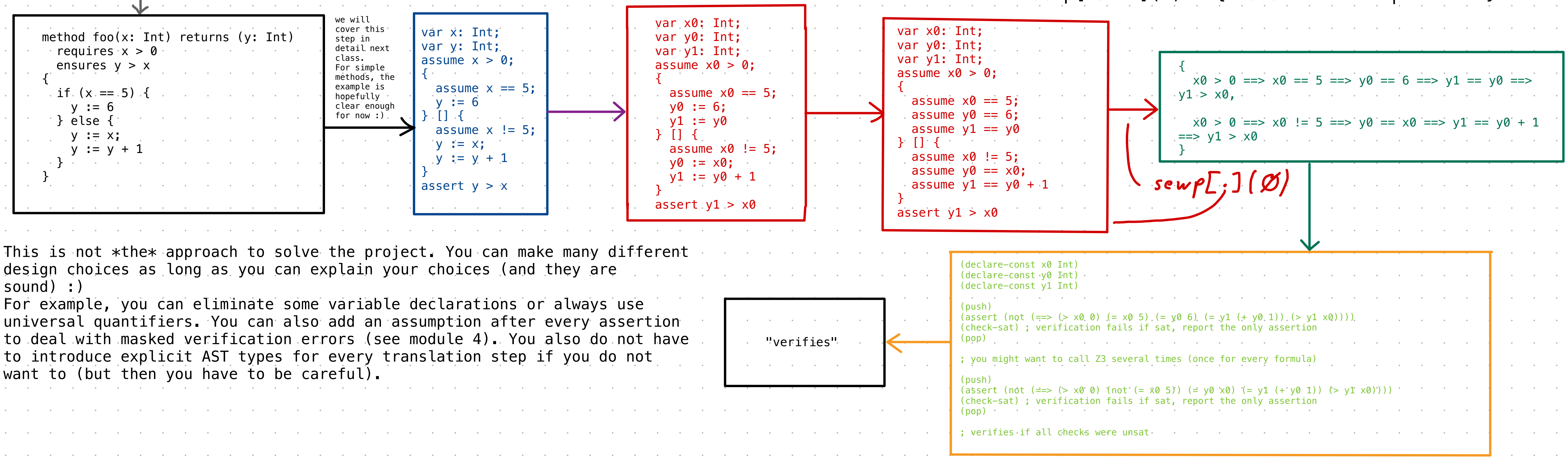
Concrete example

Here is an example of the above steps on source code level. Your implementation should perform those steps on corresponding ASTs. Some ASTs are given (e.g. the type produced by the parser or the type required by a Z3 API). You can define your own AST types for all intermediate steps in any way that works best for you. You can, for example, introduce an AST type for IVL1, IVL0, and logical formulas and then write (recursive) functions that translate IVL1 to IVL0 and IVL0 to logical formulas.

The example is for the first core goal. If you can implement a working verifier this way, then all other features can be constructed on top by encoding additional features into IVL1.

You can find more examples in the code examples that accompany the slides.

you can eliminate variable declarations at the top or introduce quantifiers using
sewp[var x](M) = { forall x :: F | F in M }



```
method foo(x: Int) returns (y: Int)
  requires x > 0
  ensures y > x
{
  if (x == 5) {
    y := 6
  } else {
    y := x;
    y := y + 1
  }
}
```

we will cover this step in detail next class. For simple methods, the example is hopefully clear enough for now :)

```
var x: Int;
var y: Int;
assume x > 0;
{
  assume x == 5;
  y := 6
} [] {
  assume x != 5;
  y := x;
  y := y + 1
}
assert y > x
```

```
var x0: Int;
var y0: Int;
var y1: Int;
assume x0 > 0;
{
  assume x0 == 5;
  y0 := 6;
  y1 := y0
} [] {
  assume x0 != 5;
  y0 := x0;
  y1 := y0 + 1
}
assert y1 > x0
```

```
var x0: Int;
var y0: Int;
var y1: Int;
assume x0 > 0;
{
  assume x0 == 5;
  assume y0 == 6;
  assume y1 == y0
} [] {
  assume x0 != 5;
  assume y0 == x0;
  assume y1 == y0 + 1
}
assert y1 > x0
```

sewp[;](∅)

```
{
  x0 > 0 ==> x0 == 5 ==> y0 == 6 ==> y1 == y0 ==>
y1 > x0,

  x0 > 0 ==> x0 != 5 ==> y0 == x0 ==> y1 == y0 + 1
==> y1 > x0
}
```

```
(declare-const x0 Int)
(declare-const y0 Int)
(declare-const y1 Int)

(push)
(assert (not (==> (> x0 0) (= x0 5) (= y0 6) (= y1 (+ y0 1)) (> y1 x0))))
(check-sat) ; verification fails if sat, report the only assertion
(pop)

; you might want to call Z3 several times (once for every formula)

(push)
(assert (not (==> (> x0 0) (not (= x0 5)) (= y0 x0) (= y1 (+ y0 1)) (> y1 x0))))
(check-sat) ; verification fails if sat, report the only assertion
(pop)

; verifies if all checks were unsat
```

"verifies"

This is not *the* approach to solve the project. You can make many different design choices as long as you can explain your choices (and they are sound) :)
For example, you can eliminate some variable declarations or always use universal quantifiers. You can also add an assumption after every assertion to deal with masked verification errors (see module 4). You also do not have to introduce explicit AST types for every translation step if you do not want to (but then you have to be careful).