Expression Programs to X86

Compilers 2024

Aslan Askarov

Expression programs

Expression programs

```
type binop = Add | Sub | Mul | Div    (* Binops as before *)

type varname = string                 (* Variables are just strings *)

type expr = (* Arithmetic expressions with variables *)
  | Int of int                        (* - Integer constant *)
  | BinOp of binop * expr * expr      (* - Binary operation *)
  | Var of varname                    (* - Variable lookup  *)

type estmt =   (* Expression statements *)
  | Val of varname * expr             (* - Value binding  *)
  | Input of varname                  (* - Input          *)

(* Expression program is a list of statements followed by an expression *)
type eprog = estmt list * expr
```

Assignment 2

Compiling Expressions Programs to x86

1. Pretty printer
2. Semantic analysis to detect unwanted programs
3. Evaluator
4. Compiler to X86

Focus of today's lecture

Semantic analysis

Semantic analysis/design decisions

- Should we report only the first error or as many as we can find?
  - We're in a simple language, we can report many errors at once
- We need a data structure for the results of semantic analysis
  - See next slide
- We need a data structure for keeping track of declared variables
  - A simple list of varnames is OK for the initial prototype; ideally, we should use Set from the standard library

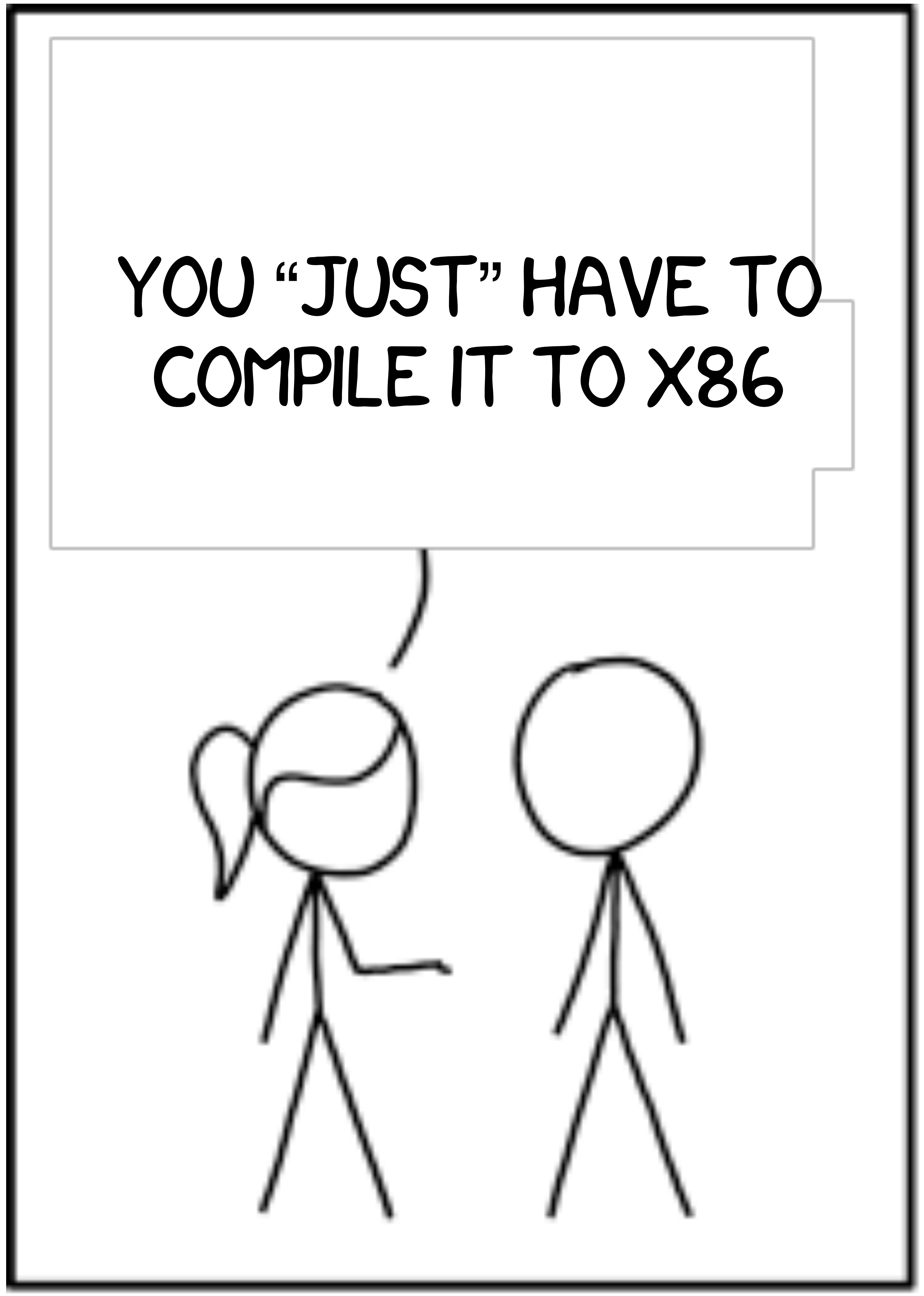
Error reporting

```
type semant_error
  = Undeclared of varname
  | Duplicate of varname

type semant_result
  = Ok
  | Error of semant_error list
```

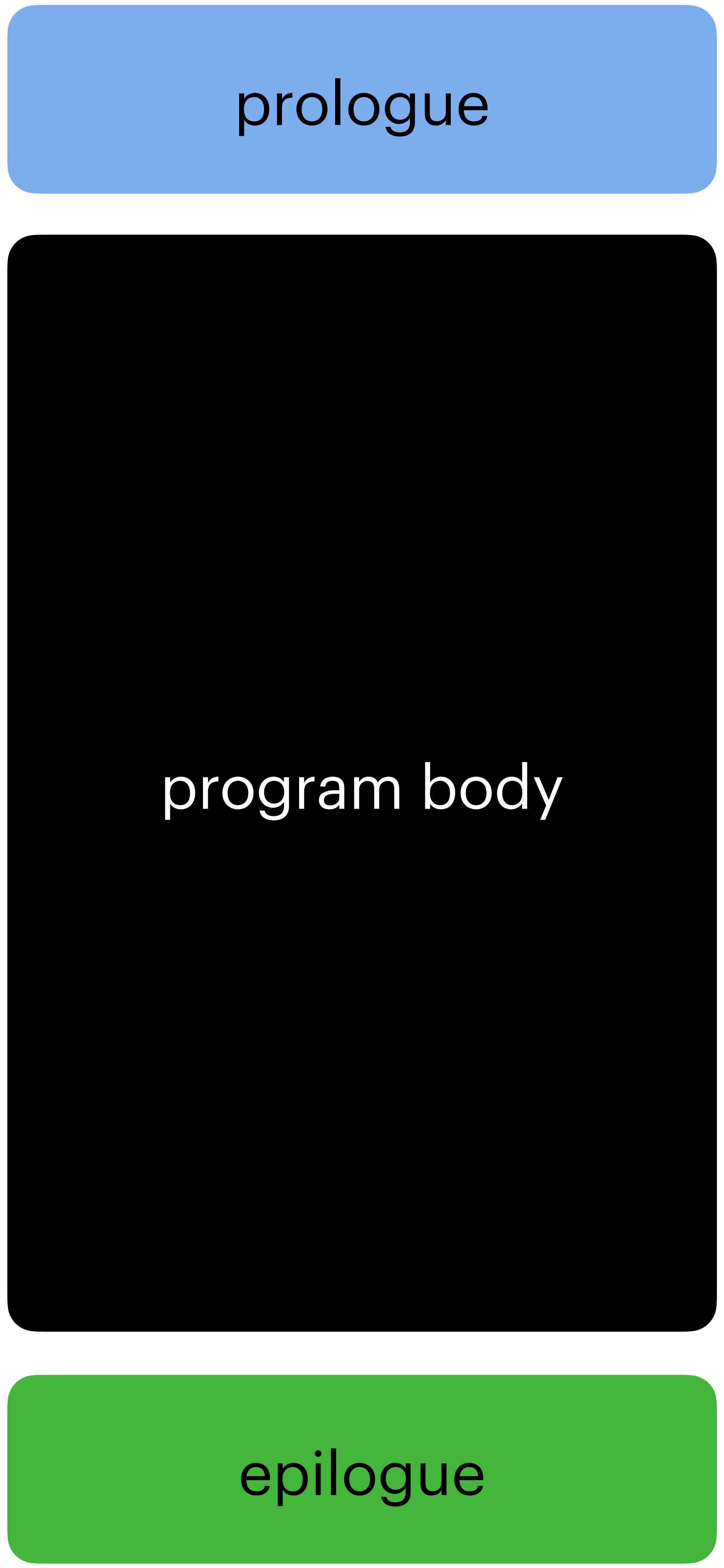
x86 code generation

Challenges in x86 compilation?

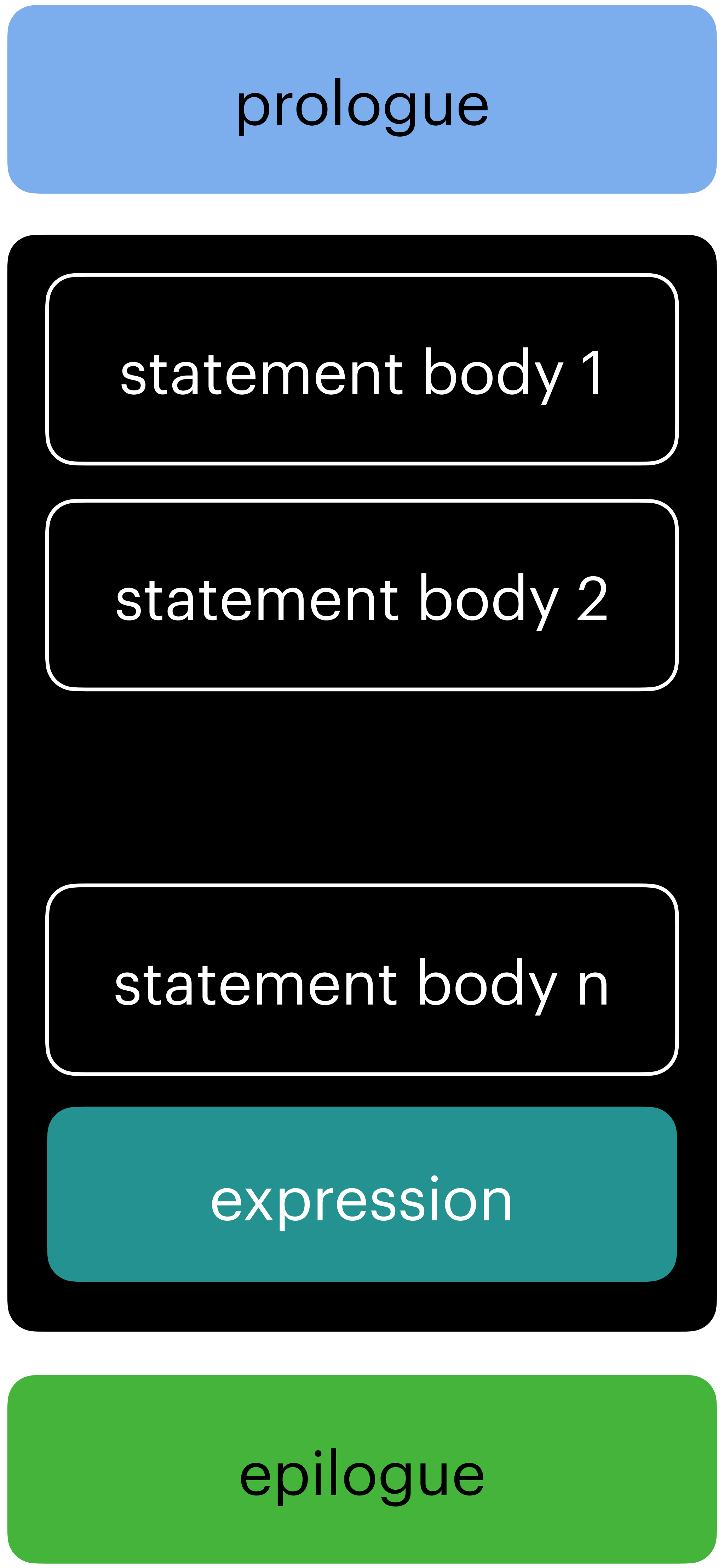


Go to menti.com and use the code 13 26 26 9

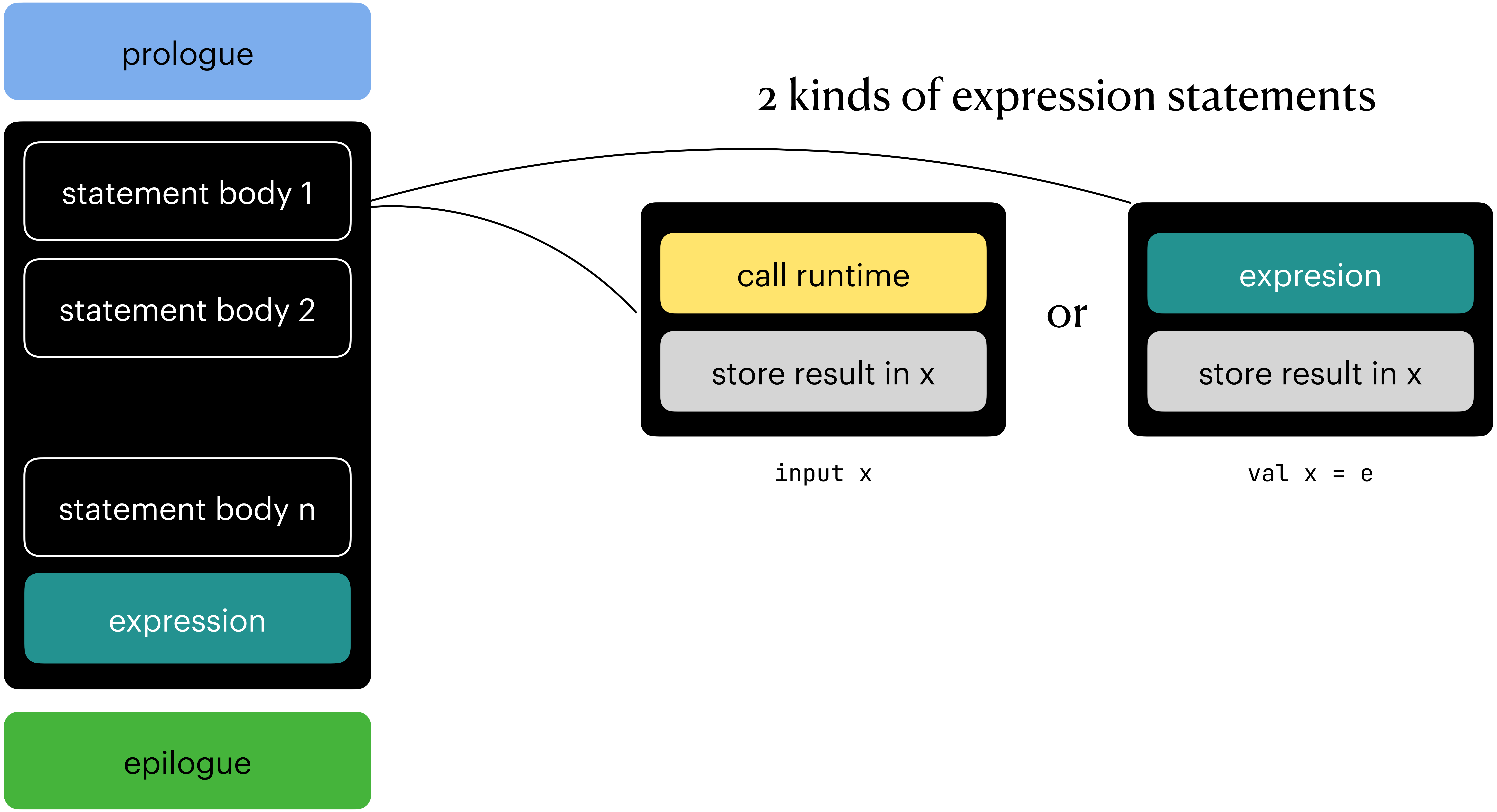
x86 code generation



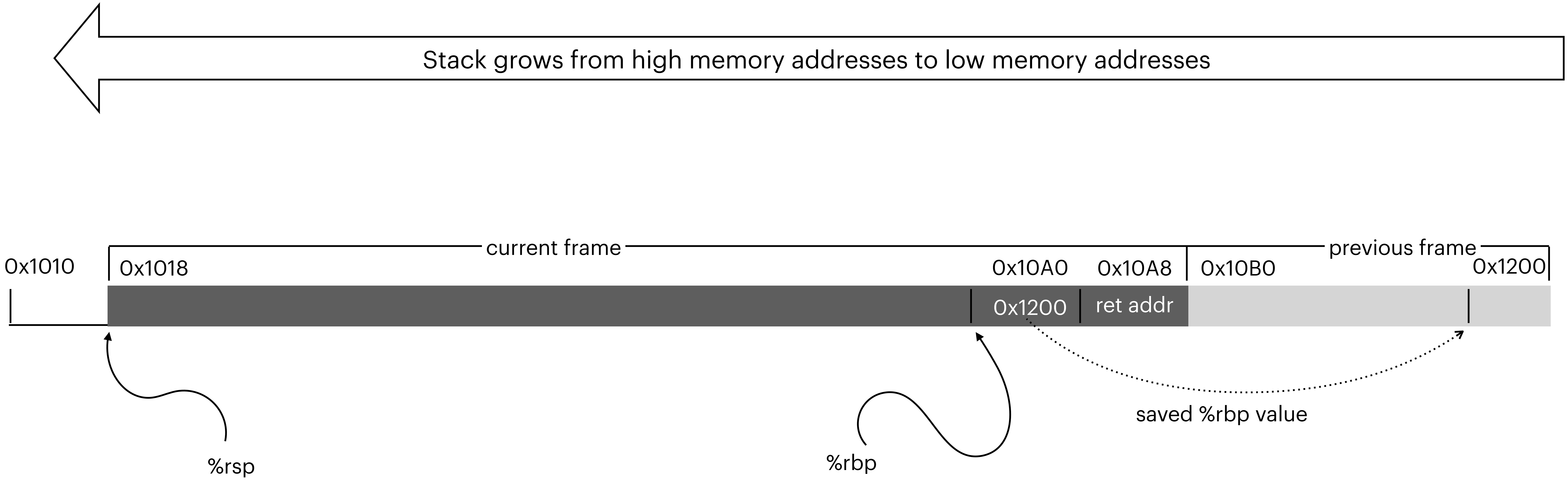
x86 code generation



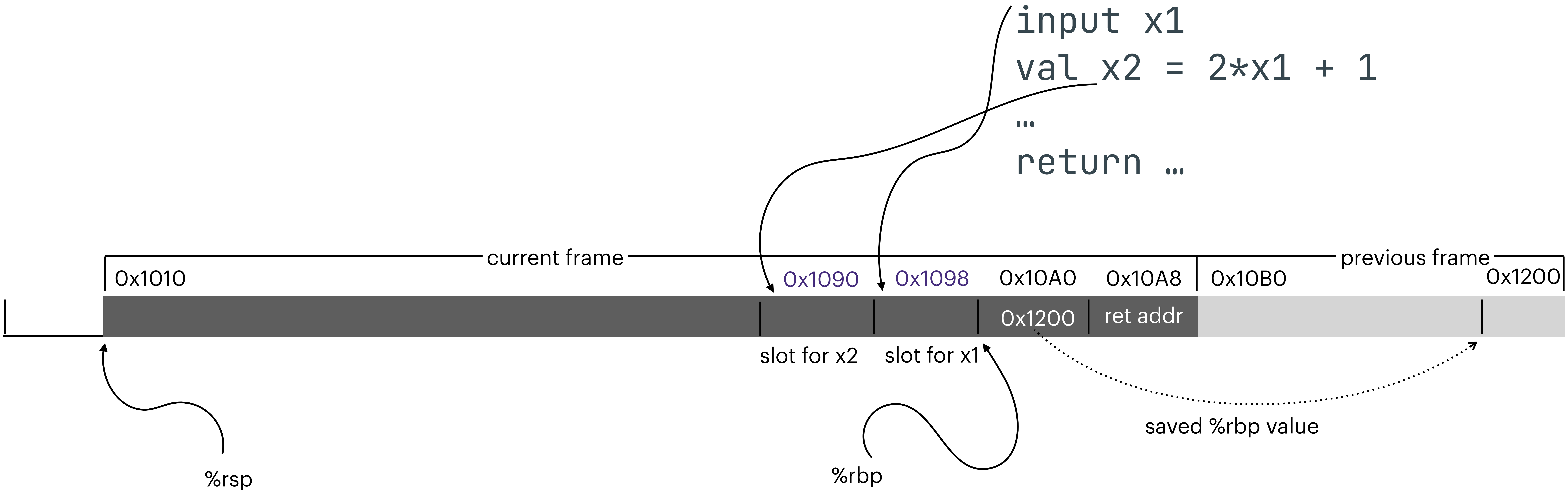
x86 code generation



Stack organization

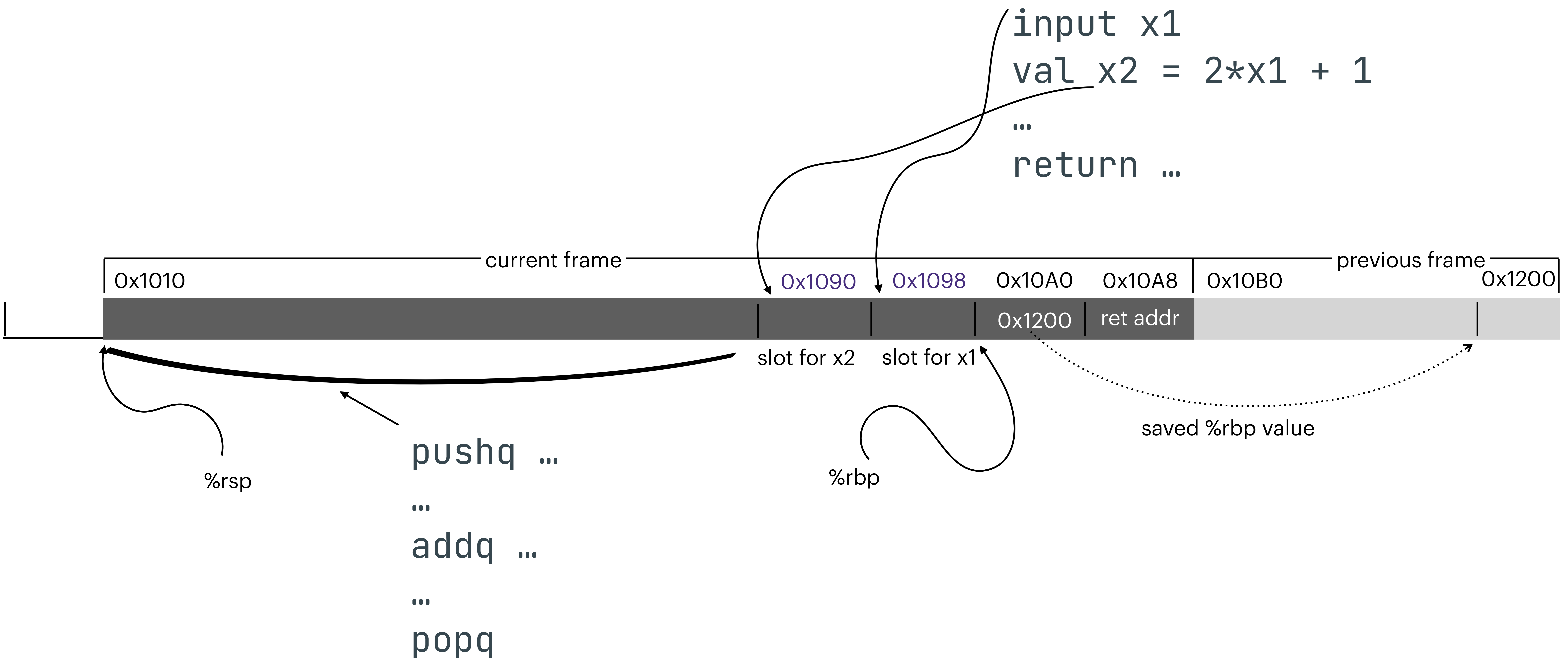


Stack slots for declared variables

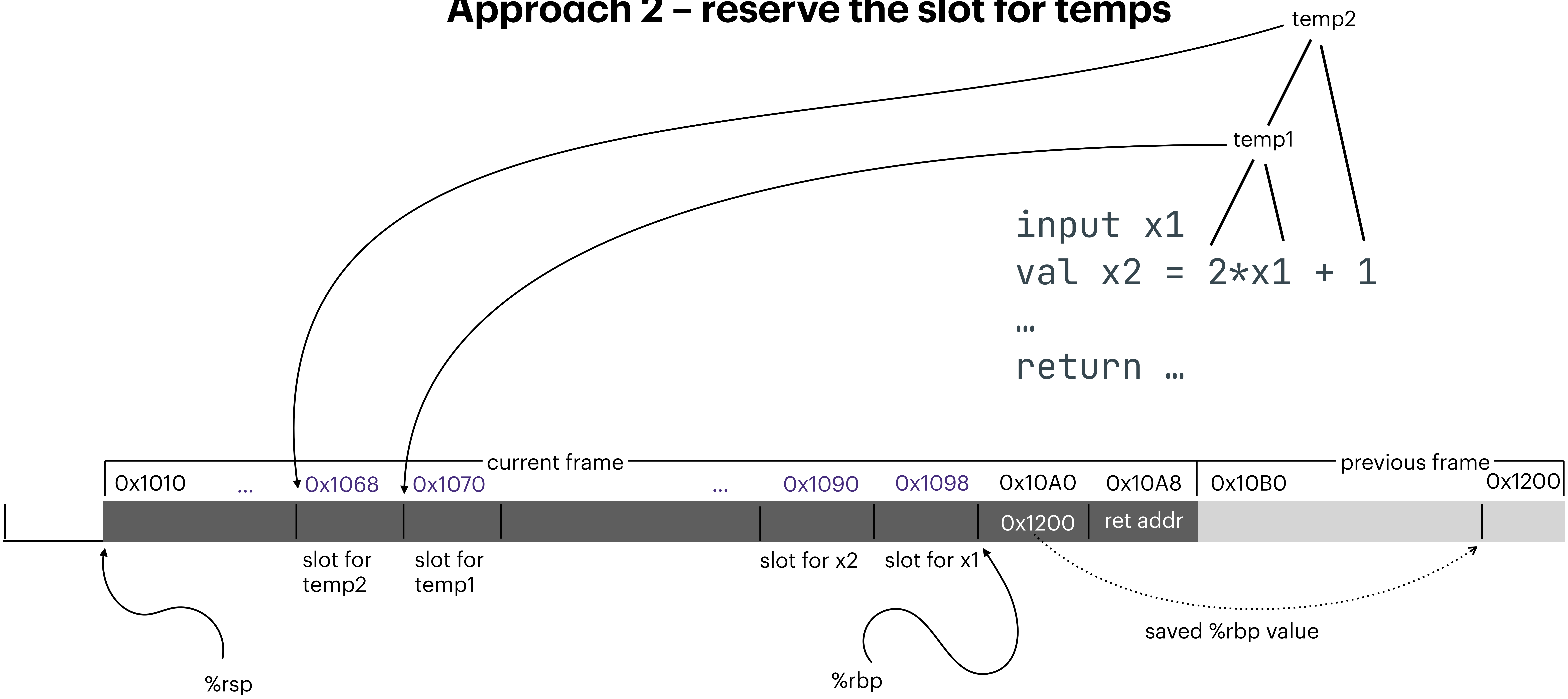


2 approaches to x86 for expressions

Approach 1 – mimic a stack machine via push/pop/mov/...



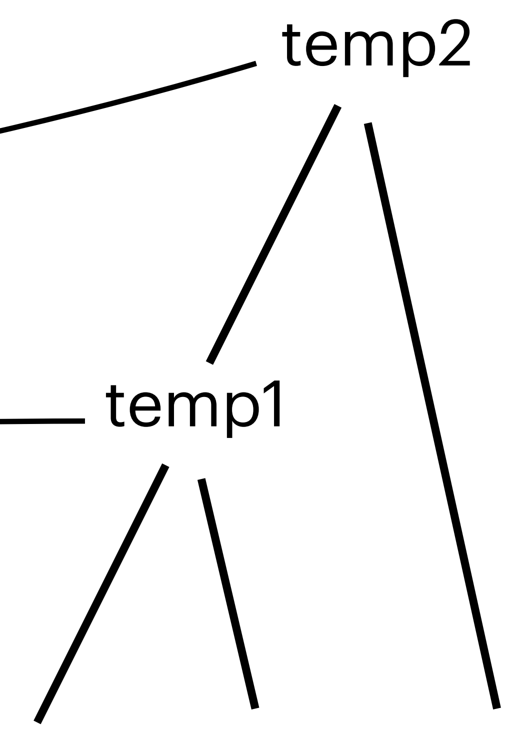
Approach 2 – reserve the slot for temps



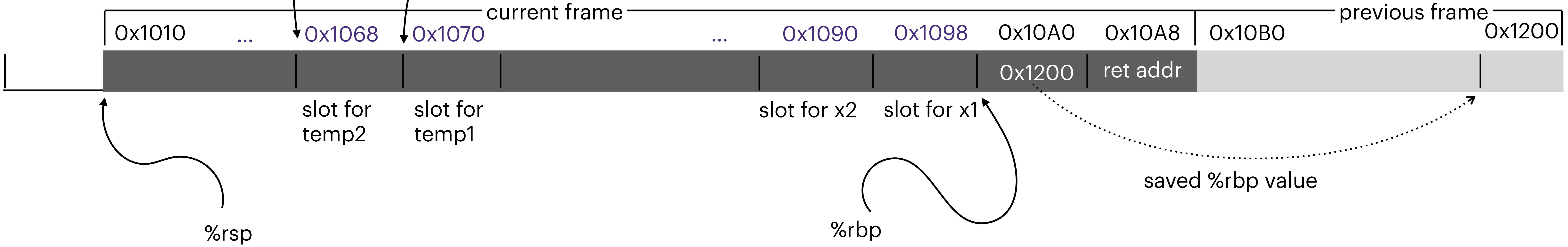
Slot organization

(One of several possibilities)

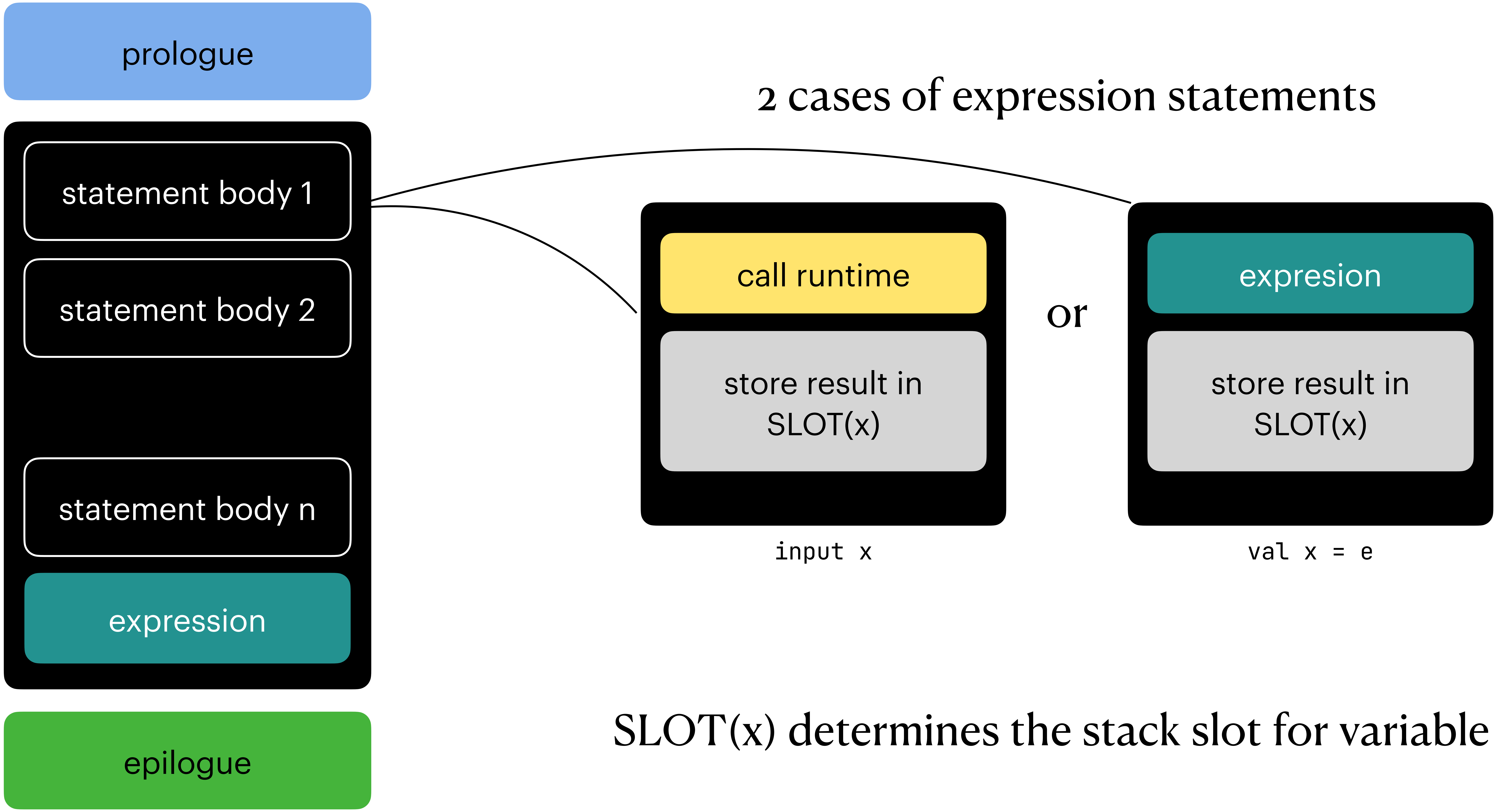
- We need an environment mapping each variable to an integer – their order in the frame layout, e.g., [(“x1”, 0), (“x2”, 1)]
- Regular variable **i** has slot **%rbp - 8*(i + 1)**
- Temporary **j** has slot **%rbp - 8*(N + j + 1),** where N is the number of (regular) local variables




```
input x1
val x2 = 2*x1 + 1
…
return …
```

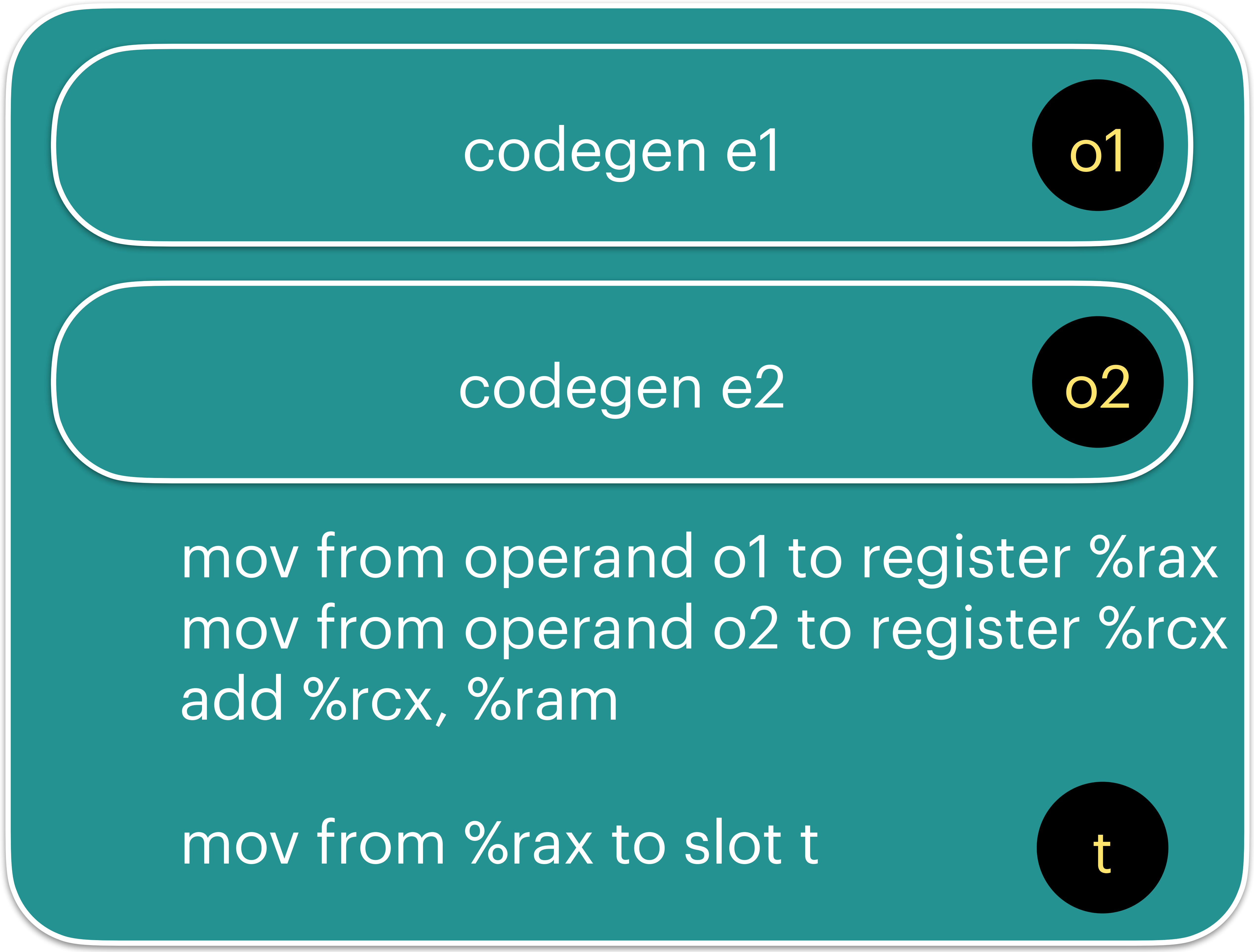


x86 code generation



SLOT(x) determines the stack slot for variable x

Code generation for expressions (almost)



`e1 + e2`

Operands can be immediate or other slots

Putting it all together (w/ approach 2)

-We need an environment mapping each variable to an integer – their order in the frame layout, e.g., [(“x1”, 0), (“x2”, 1)]

- Regular variable **i** has slot **%rbp - 8*(i + 1)**
- Temporary **j** has slot **%rbp - 8*(N + j + 1),** where N is the number of (regular) local variables

- Compute the layout for all the regulars, so you know N
- Codegen all the statements (and expressions)
- In the code generation of expressions, keep a counter for temps K
  - Observation: you can actually reuse the temp counters between expressions. Why?
- Use K + N as the size of the frame in the prologue and epilogue
- Put the whole thing together

Approach 1 vs Approach 2?

- Approach 1 is conceptually simpler
- Approach 2 is used when lowering to a even simple form preceding X-86 (e.g., in LLVM)
  - amenable to simple optimizations
  - the register allocation (later in the course!) heavily affects the final stack layout