Fundamentals of MCS (CS)

Mechanical proof assistance system (1)

2015.12.14 Ken Wakita

Basic Definition: Types

- **Basic types**
  - **nat** (natural numbers)
  - **bool** (Boolean values)
- **Functional Types**
  - **nat → nat** (the type of functions that take a natural number and give a natural number)
    - succ : **nat → nat** (next natural number)
  - **nat → nat → nat** (the type of functions that take two natural numbers and give a natural number)
    - plus : **nat → nat → nat** (_ + _)
    - mult : **nat → nat → nat** (_ × _)
  - &&: **bool → bool → bool** (logical AND)
  - ||: **bool → bool → bool** (logical OR)
    - lessThan: **nat → nat → bool** (<)
- **Generic Types** (or Types with parameters)
  - **list** α (the type of a list, where α stands for the type of its elements. α can be instantiated by other types like nat, bool)
  - **list nat** (the type of lists of natural numbers)
  - **list bool** (the type of lists of Boolean numbers)

Outline

- Example: Arithmetic expression
  - Source Language: Structure and semantics of AE
  - Target Language: Stack-based virtual machine
  - Compiler
  - Correctness: Correctness condition of the whole system

Section	Content
Source Language	Formal specification of the grammar and semantics of arithmetic expressions
Target Language	Formal specification of the (stack) machine
Translation (Compiler)	Formal specification of the compiler, which translates an arithmetic expressions to an instruction sequence.
Translation Correctness	Proof of semantic equivalence between the arithmetic expressions and the instruction sequence.

Example

- A source-level arithmetic expression
  - **e** = Binop Times (Binop Plus (Const 2) (Const 3)) (Const 7)
- Instruction sequence
  - **instr** = [iConst 7, iConst 3, iConst 2, iBinop Plus, iBinop Times]
- Equivalence
  - The arithmetic expression **e** evaluates to **35**.
  - The stack machine executes **instr** and leaves **35** on the stack.

Source Language

Source Language

Grammar	Semantics (binopDenote/expDenote)
Inductive binop@p. 1 *binop* := | Plus | Times	Definition binopDenote@p. 1 $\mathcal{B}$ Plus ⇒ plus $\mathcal{B}$ Times ⇒ mult
Inductive exp@p. 1 *exp* := | Const *n* | Binop *b* e_1 e_2	Definition expDenote@p. 2 $\mathcal{E}$ (Const *n*) ⇒ *n* $\mathcal{E}$ (Binop *b* e_1 e_2) ⇒ $\mathcal{B}$ *b* ($\mathcal{E}$ e_1) ($\mathcal{E}$ e_2)

Example

- Const(42)
- Binop Plus (Const 2) (Const 3)
- Binop Times (Binop Plus (Const 2) (Const 3)) (Const 7)

Target Language

PL Basic – Lists (1/2)

- [1, 2, 3] : list int
- [True, False, True] : list bool
- [“Hello”, “World”] : list string
- Generic lists (a list of arbitrary type α)
  - [x_1, x_2, x_3] : list α, where $x_1 : \alpha$, $x_2 : \alpha$, and $x_3 : \alpha$

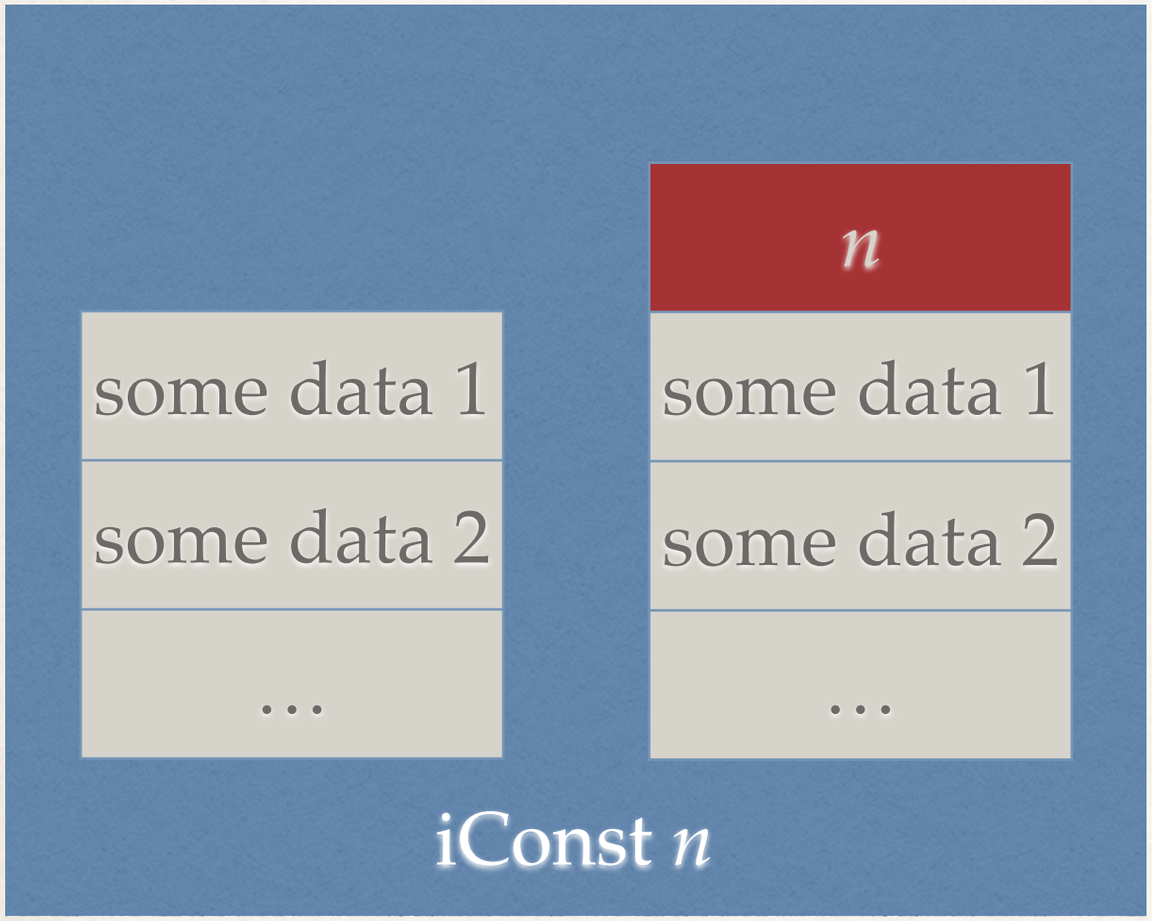
PL Basic – List (2/2) – constructors

- list ::=
  - | **nil**
  - | x :: list
- Examples
  - nil
  - 1 :: nil
  - 1 :: 2 :: 3 :: nil
- True :: False :: nil
- "x :: y :: z" should be read as "x :: (y :: z)"
- **Joining** two lists
  - $list_1$ ++ $list_2$
  - [1, 2]++[3, 4]=[1, 2, 3, 4]

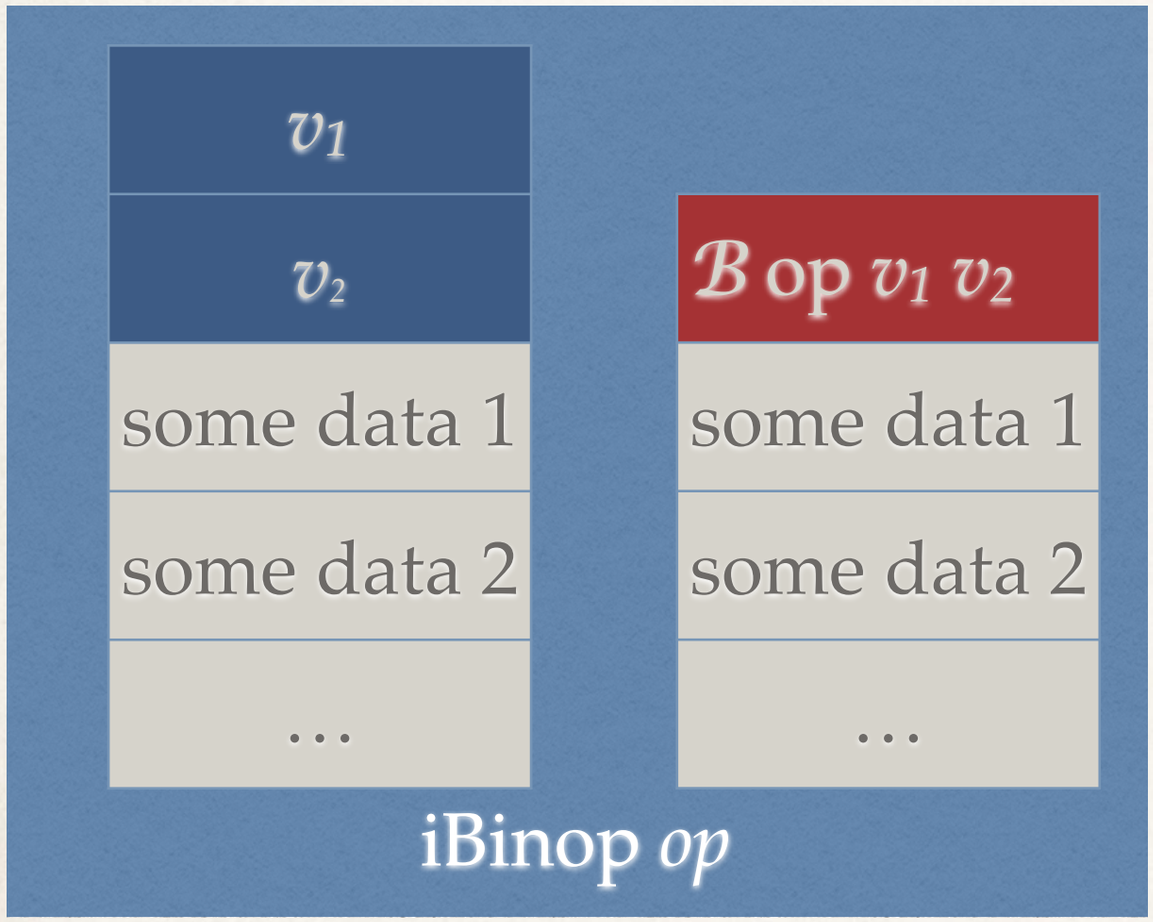
Target Language

Grammar	Semantics (instrDenote / progDenote)
instr@p. 2 *instr* ::= ∣ iConst *n* ∣ iBinop *binop*	instrDenote@p. 2 $\mathcal{I}$ (iConst n) s $\Rightarrow n :: s$ $\mathcal{I}$ (iBinop op) $v_1 :: v_2 :: s \Rightarrow (\mathcal{B}$ op $v_1\ v_2) :: s$ $\mathcal{I}$ (iBinop op) $s \Rightarrow$ ***error***
program@p. 2 *program* ::= ∣ nil ∣ *instr* :: *program*	progDenote@p. 2 $\mathcal{P}$ nil $s \Rightarrow s$ $\mathcal{P}\ (i :: p')\ s \Rightarrow \mathcal{P}\ p'\ (v :: s)$ $\mathcal{P}\ p$ ***error*** $\Rightarrow$ ***error***

Grammar	Semantics (instrDenote / progDenote)
instr ::=	$\mathcal{I}$ (iConst n) s $\Rightarrow$ n :: s
| iConst n	$\mathcal{I}$ (iBinop op) v_1 :: v_2 :: s $\Rightarrow$ ($\mathcal{B}$ op v_1 v_2) :: s
| iBinop *binop*	$\mathcal{I}$ (iBinop op) s $\Rightarrow$ ***error***



Grammar	Semantics (instrDenote / progDenote)
instr ::=	$\mathcal{I}$ (iConst n) s $\Rightarrow$ n :: s
| iConst n	$\mathcal{I}$ (iBinop op) v_1 :: v_2 :: s $\Rightarrow$ ($\mathcal{B}$ op v_1 v_2) :: s
| iBinop *binop*	$\mathcal{I}$ (iBinop op) s $\Rightarrow$ ***error***



If the iBinop instruction fails to find two values on the stack, its computation cannot proceed. We consider this situation an error.

Too few values on the stack

Target Language (Revised) Dealing with Errors

Grammar	Semantics (instrDenote / progDenote)

instr ::=
  | iConst *n*
  | iBinop *binop*
	$\mathcal{I}$ (iConst n) (Some s) ⇒ Some (n :: s)
	$\mathcal{I}$ (iBinop op) Some(v_1 :: v_2 :: s) ⇒
	  Some(($\mathcal{B}$ op v_1 v_2) :: s)
	$\mathcal{I}$ (iBinop op) s ⇒ None

program ::=
  | nil
  | *instr* :: *program*
	$\mathcal{P}$ nil Some(s) ⇒ Some(s)
	$\mathcal{P}$ (i :: p') Some(s') ⇒ $\mathcal{P}$ p' Some(v :: s')
	  where instrDenote i s ⇒ Some s'
	$\mathcal{P}$ p ***None*** ⇒ ***None***

Execution of instructions

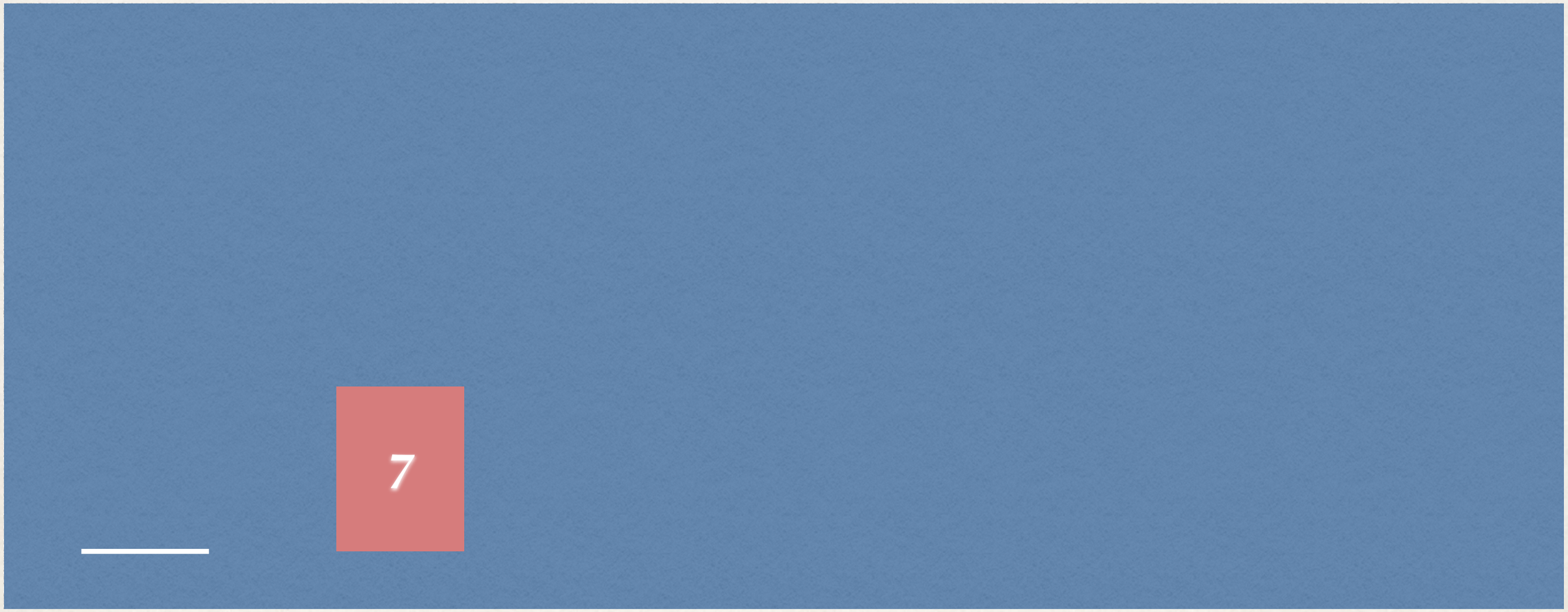
prog = [**iConst 7**, iConst 3, iConst 2, iBinop +, iBinop ×]

Execution of the program starts with an empty stack

Red in prog indicates the next instruction to execute

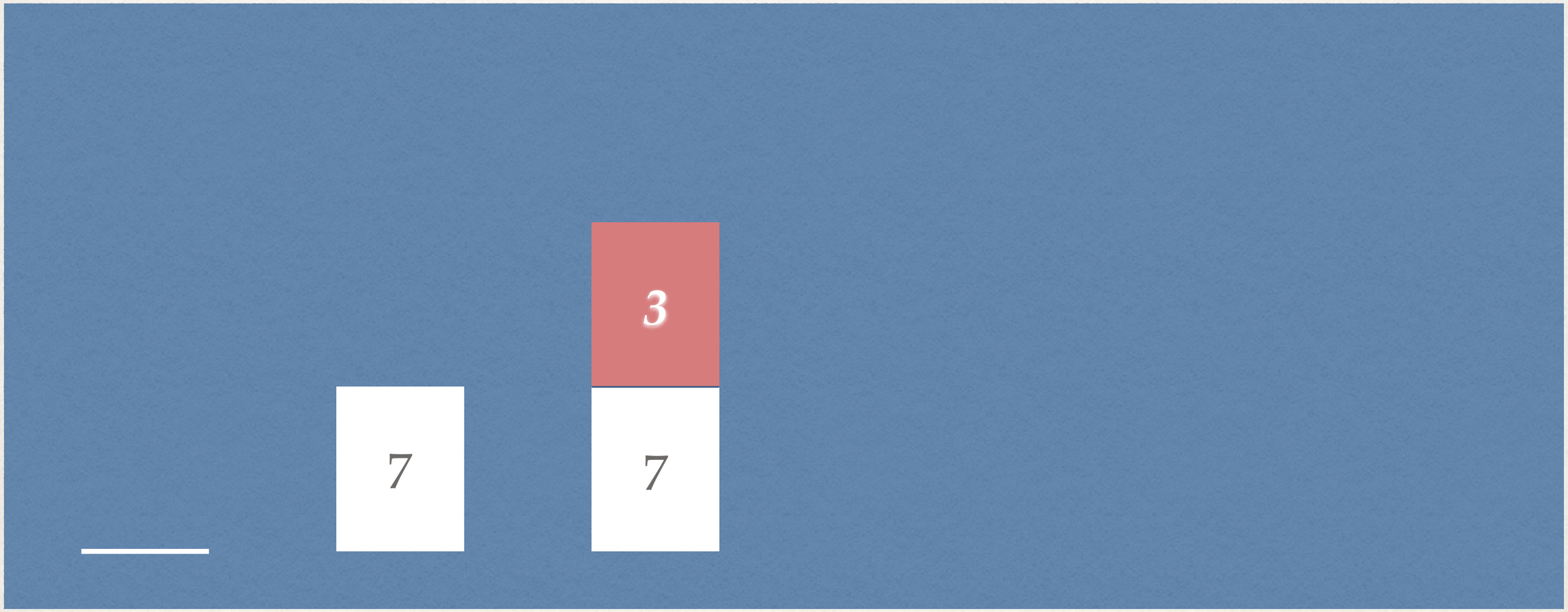
Execution of instructions

prog = [**iConst 7**, **iConst 3**, iConst 2, iBinop +, iBinop ×]



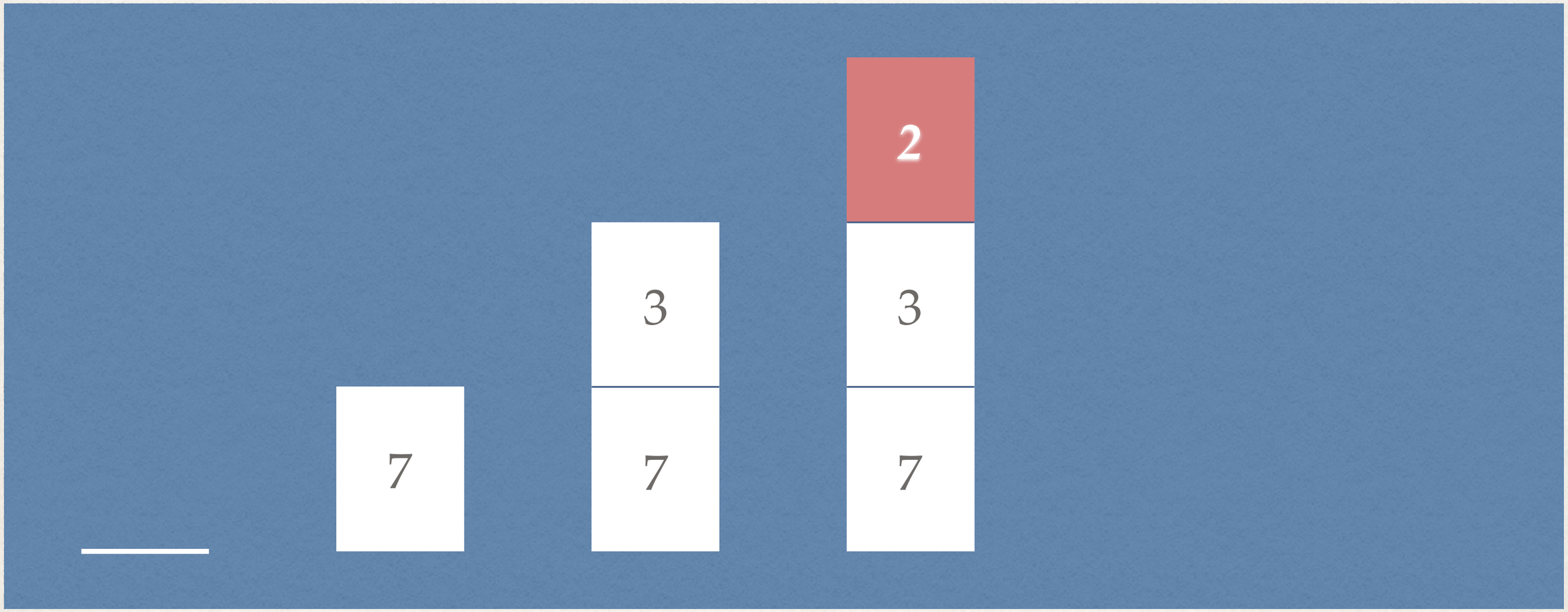
Execution of instructions

prog = [iConst 7, **iConst 3**, **iConst 2**, iBinop +, iBinop ×]



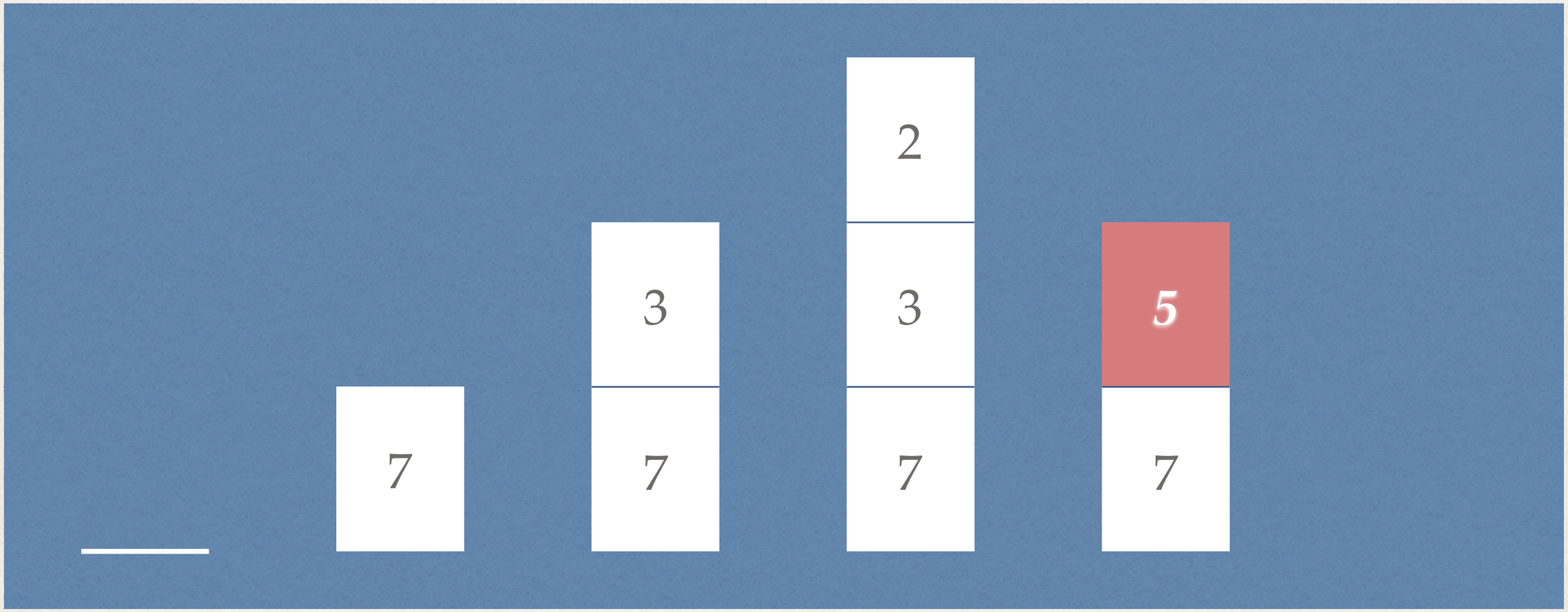
Execution of instructions

prog = [iConst 7, iConst 3, **iConst 2**, **iBinop +**, iBinop ×]



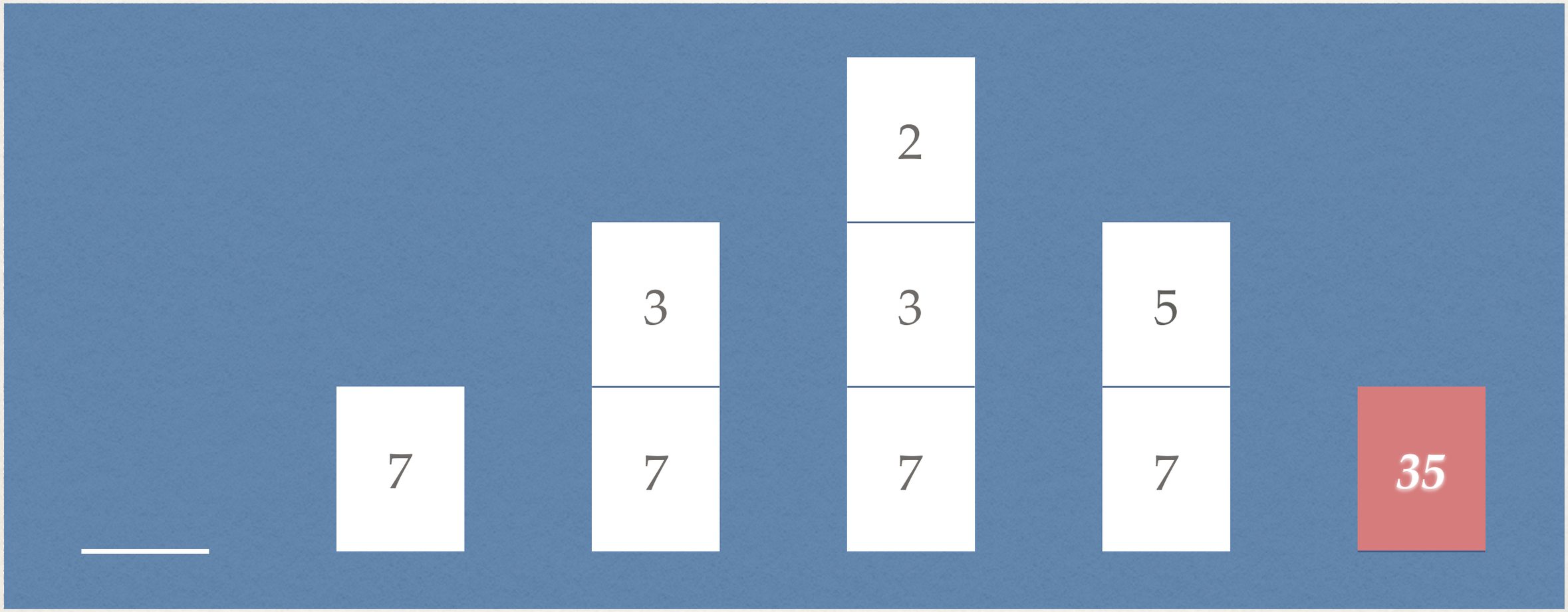
Execution of instructions

prog = [iConst 7, iConst 3, iConst 2, **iBinop +**, **iBinop ×**]



Execution of instructions

prog = [iConst 7, iConst 3, iConst 2, **iBinop +**, **iBinop ×**]



Execution of instructions

- Initial state
  - prog = [iConst 3, iConst 2, iBinop +]
  - stack = nil
- [iConst 3, iConst 2, iBinop +]
  nil
- [iConst 2, iBinop +]
  3 :: nil
- [iBinop +]
  2 :: 3 :: nil
- []
  5 :: nil

Compiler

Compiler

- compile : exp → prog
  where prog = list instr

- compile (Const n) ⇒ iConst n :: nil

- compile (Binop b e1 e2) ⇒
  (compile e2) ++ (compile e1) ++ (iBinop b) :: nil

Examples

- compile (Const 2) ⇒ iConst 2

- compile (Binop + (Const 2) (Const 3)) ⇒
  (iConst 3) :: (iConst 2) :: (iBinop +) :: nil

  Or, [iConst 3, iConst 2, iBinop +]

- compile (Binop × (Binop + (Const 2) (Const 3)) (Const 7)) ⇒
  (iConst 7) :: (iConst 3) :: (iConst 2) :: (iBinop +) :: (iBinop ×)

  Or, [iConst 7, iConst 3, iConst 2, iBinop +, iBinop ×]

Correctness

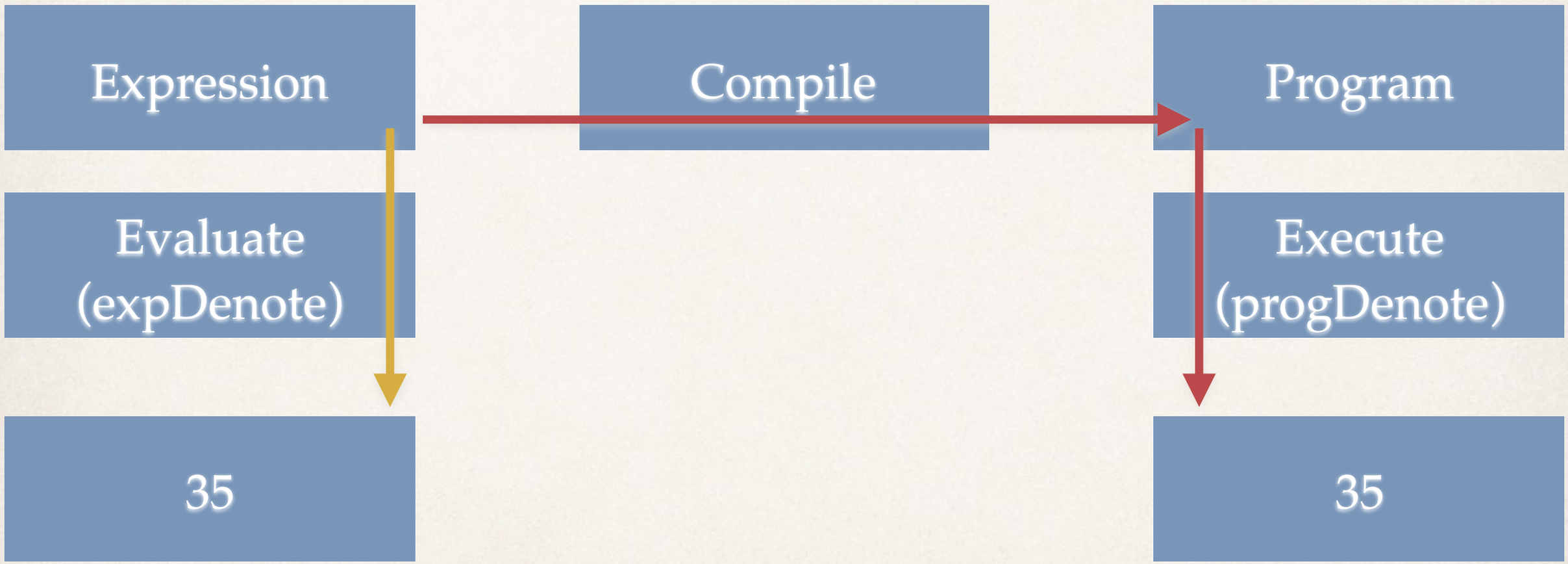
Testing the compiler for some arithmetic expressions

- e1 = Const 2 ⇒ 2
  progDenote (compile e1) ⇒ 2 :: nil

- e2 = Plus (Const 2) (Const 3) ⇒ 7
  progDenote (compile e1) ⇒ 7 :: nil

- e3 = Times (Plus (Const 2) (Const 3)) (Const 7) ⇒ 35
  progDenote (compile e1) ⇒ 35 :: nil

Interpretation vs Compile & Go



Correctness condition for the compiler

- Every arithmetic expression, should be translated by the compiler to equivalent stack machine program.

- for all e. (compile e) equiv. e

Correctness condition for the compiler

- Every arithmetic expression, should be translated by the compiler to equivalent stack machine program.

- for all e. (compile e) equiv. e

- for all e.

  denoteProg (compile e) = (expDenote e) :: nil