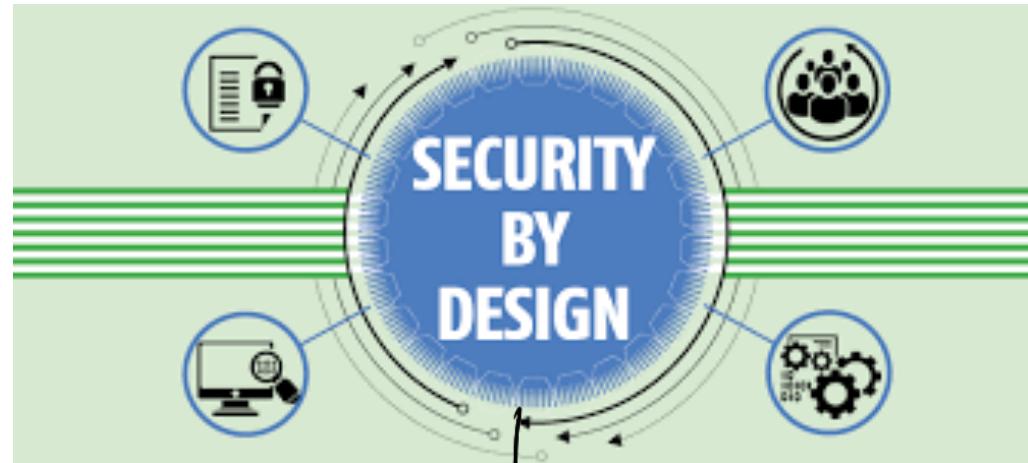


# Formal Techniques in Action: Ensuring Correctness & Security

# A science of security

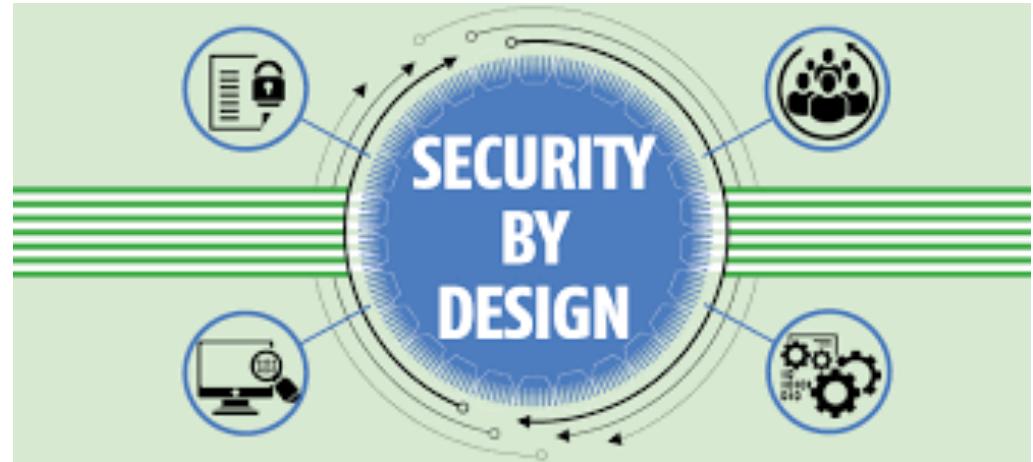
We have the recipes



Idea: Think of security as we design, no putting patches

# A science of security

We have the recipes



But .. what about the  
math-foundations?



We have a program (a code base)  
We have the world of models and  
map between world of syntax and world  
of models as called Semantics.

Formal methods

- The **program** block represents the actual code—its **syntax** (the way it's written, its structure).
- The **mathematical models** block represents how we interpret what the program does in a more abstract, precise way (often using logic, algebra, or automata).
- The **arrow labeled "semantics"** means that we use formal rules to translate from syntax (the program) into its corresponding mathematical meaning (its model).

Now, inside the **mathematical models** block, you have "formal methods" with a circular arrow. This means that within the world of models, we apply rigorous mathematical techniques (like model checking, theorem proving, etc.) to verify properties about the **program**—ensuring security, correctness, and reliability.

# Ensuring security of the programming language ecosystems

*Security is a problem of ecosystem*

---

To ensure the **correctness, security, and reliability** of a programming language ecosystem, multiple aspects must be considered beyond just code execution.

---

These include **Static Analysis, Secure Code Generation, and Instrumentation**, each playing a crucial role in detecting errors, enforcing security, and verifying runtime behavior.

We have mathematical models that allow us to build tools to support security; three main elements are **Static Analysis, Code generation and Code analysis**.

Yes, that statement is correct! Mathematical models provide a foundation for building tools that enhance security in various ways. Here's how each of those tools fits into the picture:

- **Static Analysis** → Uses mathematical models to analyze code without running it, detecting vulnerabilities, bugs, or security risks based on formal methods.
- **Code Generation** → Some tools generate secure code automatically from mathematical specifications, ensuring correctness by construction.
- **Code Instrumentation** → Involves modifying a program (inserting checks, logs, or security mechanisms) to enforce security properties, often guided by formal models.

→ Applied without sending the program into execution

## Static Analysis

- Static analysis involves examining source code **without executing it** to identify potential security vulnerabilities.
- **Type Checking:** Ensuring that type constraints are met, preventing runtime errors.
- **Data Flow Analysis:** Tracking data usage to detect uninitialized variables, potential race conditions,
- **Information Flow Control:** tracking how data moves within a system to prevent unauthorized access, leaks, or manipulation. \*
- **Taint Analysis:** Identifying user input that might lead to security vulnerabilities.

\* Track how data flows to avoid statically written or not data can be used without privileges.

# Secure Code Generation

---

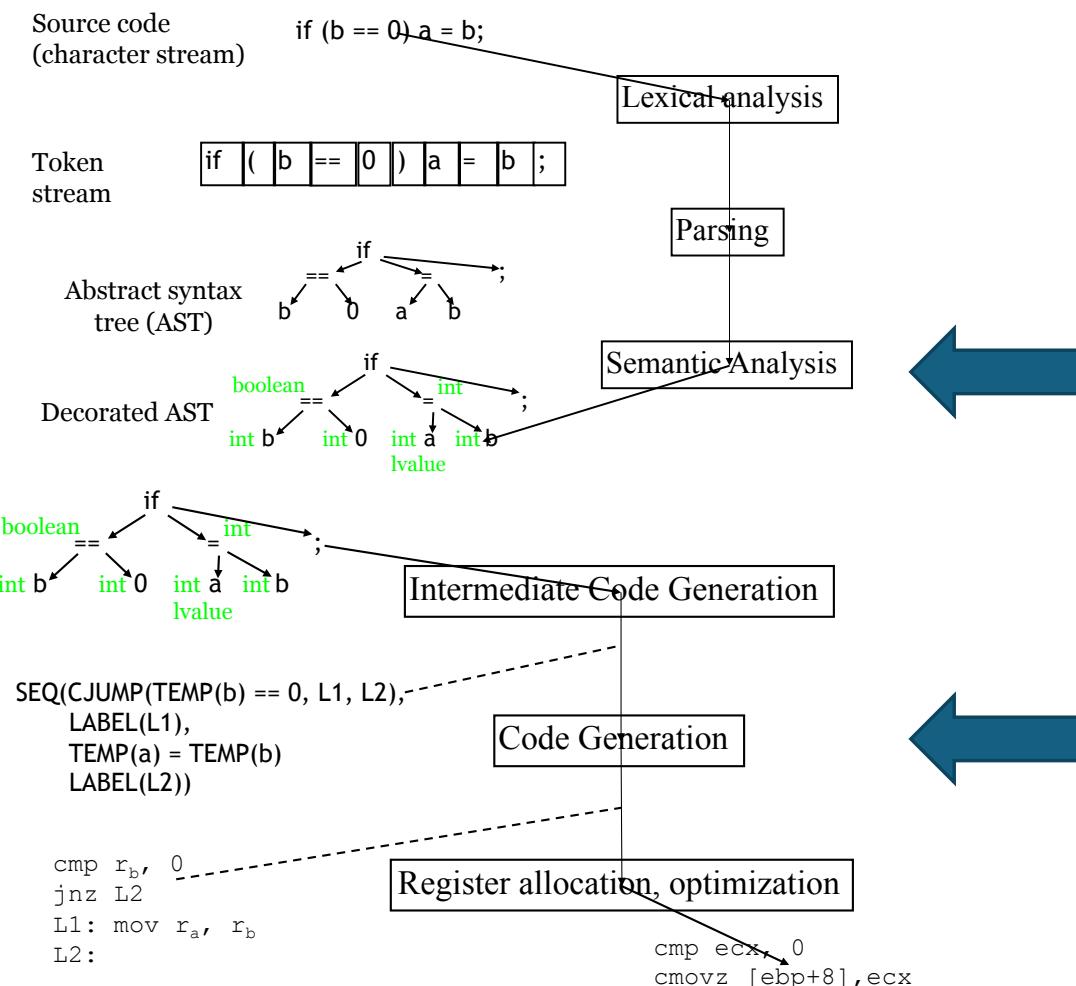
- Secure code generation ensures that the compiler translates high-level source code into **semantically equivalent and secure** machine code. This process must maintain:
  - **Semantic Preservation:** Ensuring that the generated code faithfully executes the original program logic.
  - **Memory Safety Guarantees:** Preventing buffer overflows and memory corruption.
  - **Control-Flow Integrity (CFI):** Ensuring that the execution path follows expected control flow to prevent exploits.

# Code Instrumentation

---

- Instrumentation involves **adding extra code** during compilation to monitor, log, and verify program execution. This helps in:
- **Runtime Safety Checks**: Ensuring array bounds checking, null pointer dereferences, and memory leaks are prevented.
- **Security Enforcement**: Detecting anomalies such as unauthorized memory access or privilege escalation.

Back to compilers



**Static Analysis**

**Secure Code &  
Code Instrumentation**

We want a model to discuss program behavior

## Our approach: two issues

**Semantics:** How can we model program execution?

**Analysis & Security:** How can we use these models to guarantee that certain security properties hold?

So when we talk about semantics for our program we are asking ourselves how to model formally the execution of a program, how to define formal rules that explain how the abstract machine should behave given a certain code, correct?

Exactly! When we talk about **semantics** for a program, we're trying to formally describe how the program executes—what each piece of code *means* in terms of behavior.

- We need a **formal model** to define execution in a precise way, avoiding ambiguity.
- This involves setting up **formal rules** that describe how an abstract machine (a theoretical model of computation) should process code.
- The goal is to capture how a program's syntax translates into its actual effects when executed, which is crucial for reasoning about correctness and security.

You're getting it—semantics is all about defining the *meaning* of code in a mathematically rigorous way!

# Formal Techniques Based on Operational Semantics in Compiler Code Generation

# Operational Semantics

- gives you tool to describe how a program behaves by describing a program in terms of states and transitions.  
• Operational semantics provides a well-defined mathematical framework that describes how a program executes step by step, making it a valuable tool for ensuring correctness, security, and optimizations in code generation.
- Many compilers define operational semantics using abstract machines. These machines act as an intermediate execution model, helping bridge high-level semantics with low-level code generation.

① A b. mach. introduces data types and rules of execution. They provide an intermediate level, a model of the execution that you compile to the actual language

## What is Operational Semantics?

Operational semantics is a way of defining the meaning of a program by describing **how it executes step by step** on an abstract machine. Instead of just saying *what* a program does in a high-level way, it gives a precise, rule-based description of *how* each individual operation changes the program's state.

## How does it work?

- We define a **formal machine model** that represents the system executing the program.
- We set up **rules** that describe how each statement or instruction transforms the machine's state.
- The execution is broken down into **discrete steps**, moving from one configuration (state) to the next.

Think of it as a **mathematical simulation** of how a program runs. It's like writing a recipe for an interpreter that follows exact rules to process code.

## Compilers and Operational Semantics

A **compiler** translates high-level code (like C, Python, or Java) into lower-level code (like assembly or machine code). To do this systematically, many compilers use **abstract machines** as an intermediate step.

## What is an Abstract Machine?

An **abstract machine** is a theoretical model that executes a program according to well-defined operational semantics. It sits between:

1. The **high-level language** (where we define semantics formally).
2. The **low-level machine code** (which actually runs on real hardware).

By defining operational semantics using an abstract machine, we get a clear, structured way to model execution before translating it into actual instructions for a real processor.

So the semantics of a program are describing its formal execution on an abstract machine, the compiler then takes this formal execution and translates it into actual machine language?

Yes, that's exactly it!

# Our case study: Web Assembly

- **WebAssembly (WASM)** is a low-level, binary instruction format designed for **fast, secure, and efficient execution** of code on web browsers and other environments.
  - is a universal format that compiles or translates code into, and this translated code can run on any system that supports WASM
- **WASM** is a portable compilation target, allowing applications written in languages like **C, C++, and Rust** to run on the web alongside JavaScript.

# Web Assembly

- WebAssembly is the most popular programming language ever (how many people have a browser, after all?) to have a complete, verified formal specification.
  - <https://webassembly.github.io/spec/core/index.html>
- WebAssembly is a provably type-safe language.\*
  - It has a type-checker (not common for assembly languages)
  - It has no undefined behavior and no stuck states.
- WebAssembly notable feature stands on its security-first design.  
*\* It has a complete proved model for safety, explained in the specifications*

# Our goal

Formalize a subset of WebAssembly using the formal techniques:  
operational semantics.

Remark: The operational semantics presented here differ from those in the formal specification for the sake of clarity.



Operational  
semantics  
smoothly

Simple imperative language

# The While language

$n \in \mathbf{Num}$        $x \in \mathbf{Var}$

$a \in \mathbf{Aexp}$  *Arithmetik*

$a ::= n \mid x \mid a_1 + a_2 \mid \dots$

$b \in \mathbf{Bexp}$  *Boolean*

$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 > a_2$   
 $\mid \text{not } b \mid b_1 \text{ and } b_2 \mid \dots$

$c \in \mathbf{Cmd}$  *(commands)*

$c ::= x := a \mid \text{skip} \mid c_1 ; c_2$   
 $\mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$

integers and variables  
arithmetic expressions

boolean expressions

commands (i.e. programs)

↗ Notation for CFG

This definition is called a **BNF grammar**: (not used for AST)

- ▶ Different syntactic categories
- ▶ Certain basic categories are assumed: numbers, variables.
- ▶ Valid programs are described by **abstract syntax trees** (ASTs);

## What does it do?

BNF provides a precise set of rules that describe how valid statements in a language can be formed. It defines what sequences of symbols (keywords, operators, etc.) are considered **correct syntax**.

## How does it work?

BNF uses **production rules**, where each rule has:

- A **non-terminal** (something to be expanded further).
- A **definition** (what it can be replaced with).
- **Terminals** (actual characters or tokens, like `if`, `+`, `123` ).

# Operational semantics

- The operational semantics of a programming language is defined by the abstract machine of the language which is formalised as a transition system.  
→ automata, but \*states is not finite
- Transition system  $(\Gamma, T, \rightarrow)$  Math. structure: triple ( $\Gamma$ : set of all possible states,  $T$ : final states,  $\rightarrow$ : transition relation)
- $\Gamma$  is the set of configuration (the set of states of the abstract machine)
- $T \subseteq \Gamma$  is the set of final configuration Then the transition relation
- $\rightarrow \subseteq \Gamma \times \Gamma$  is the transition relation
  - $(s, s') \in \rightarrow$  written  $s \rightarrow s'$  describes state evolution: how state  $s$  evolves into state  $s'$

A command is a state transformer

## 1. $\Gamma$ (Set of Configurations):

- **Configurations** represent the state of the system at any given moment during execution.
- A configuration contains all the necessary information to describe the state of the system, such as the current values of variables, the program counter (if applicable), or any other relevant system state.
- The set  $\Gamma$  is the set of all possible configurations that the system can be in.

## 2. $T$ (Set of Final Configurations):

- **Final configurations** are the terminal states of the system.
- These configurations represent the end of the computation, meaning the program has completed its execution or reached a state from which it can no longer transition (e.g., an error or halting state).
- The set  $T$  is a subset of  $\Gamma$ , containing those configurations where no further transitions are possible.

## 3. $\rightarrow$ (Transition Relation):

- The **transition relation**  $\rightarrow$  defines how configurations evolve over time. It specifies the rules for state transitions.
- A transition  $\gamma_1 \rightarrow \gamma_2$  means that the system can move from configuration  $\gamma_1$  to configuration  $\gamma_2$  according to the defined operational semantics.
- Transitions occur based on the actions or instructions defined in the program. For example, an instruction might cause a variable to change value or a loop to be executed, which leads to a new configuration.

CONFIGURATION = STATE

Gotcha. True or false: if we assume that to represent our state we use also a set of the instructions yet to be executed, when we arrive in a state in which this set is empty, that is a final state.

True.

# Transition Relation

- Two main styles of definitions for the transition relation:
  - Small-step semantics also called Structural Operational Semantics (SOS)
    - The Transition Relation → describes all intermediate, individual steps ①
  - Big-step semantics also called Natural semantics (NS)
    - Relation  $\Downarrow$  describes how to obtain the final result of computation ②
- ① All the steps of program evolution within abstract machine
- ② Provide only initial state and final state in a computation.

## Small-Step Semantics:

- In **small-step semantics**, the transition relation describes **individual, intermediate steps** of computation.
- This means that the execution is broken down into **very fine-grained steps** where each step represents a single operation or transition.
- For example, if the program involves an expression like  $3 + 4$ , a small-step transition would describe **each individual step** that leads to the result:
  - First, the system would reduce  $3 + 4$  to  $7$ .
  - So, a small-step transition would describe the change from one state to the next, step by step.
- The relation describes how the program moves forward, one **small step at a time**.

## Big-Step Semantics:

- In **big-step semantics**, the transition relation ( $\Downarrow$ ) describes **how to obtain the final result of the computation in a single, large step**.
- Instead of describing intermediate steps, big-step semantics **focuses on the end result** after the entire computation has finished.
- So, instead of breaking down each individual operation, you **express the computation from start to finish** in one go, showing how the entire program (or a large part of it) computes its result in a single transition.
- It's like **moving directly from the initial state to the final state** without caring about the intermediate steps.

So the relation transition is in that case a function that brings you from an initial state to the final one, that's it. Correct?

Yes, exactly! In **big-step semantics**, the transition relation ( $\Downarrow$ ) is **essentially a function that maps an initial state (or configuration) directly to a final state (or result)**. The key point is that it describes the **overall outcome of the computation**, skipping over the intermediate steps.

To put it another way:

- The relation  $\Downarrow$  in big-step semantics is a **single transition** from the starting configuration to the final result.
- It focuses purely on the **final result** after the program has completed its execution, without showing the **intermediate steps**.

# Transition Relation: While Language

- To define the operational semantics of the while language we need to manage **statements (commands)** and **state**.
- States are mapping from variables into integers
  - $\sigma \in \Sigma, \sigma: Var \rightarrow Int$  states are mappings of prog. variables into values
- **Configurations**
  - $\Gamma = \{ (c, \sigma) \mid c \in Cmd, \sigma \in \Sigma \} \cup \Sigma \rightarrow \text{final states, that cannot be executed anymore}$
- Execution  $(c, \sigma) \rightarrow (c', \sigma')$  of the command  $c$  has not terminated and  $(c', \sigma')$  is left to execution
- Execution  $(c, \sigma) \rightarrow \sigma'$  of the command  $c$  has terminated and the final state is the state  $\sigma'$

[STATE = memory]

## 1. States ( $\sigma \in \Sigma$ ):

In the context of the **While** language, a **state** is simply a mapping from variables to integers. Essentially, it's a snapshot of the values of the variables in the program at a particular point in time.

- **Formally:** A state  $\sigma$  is a function that takes a **variable** from a set of variables (denoted `Var`) and maps it to an **integer** (denoted `Int`).

bash

Copia Modifica

$\sigma \in \Sigma$ , where  $\sigma: Var \rightarrow Int$

This means that a **state** represents the **current values** of all the variables in the program. For example, if we have two variables `x` and `y`, a state might look like:

$\sigma = \{ x \rightarrow 5, y \rightarrow 10 \}$

Here, the state is telling us that  $x = 5$  and  $y = 10$ .

## 2. Configurations ( $\Gamma$ ):

A **configuration** is a pair that combines the **current state** and the **command** that is about to be executed.

- **Formally:** A configuration is either:

1. A pair  $(c, \sigma)$ , where  $c$  is a **command** (from the set `Cmd`) and  $\sigma$  is a **state** (from the set  $\Sigma$ ), or
2. Just a state  $\sigma$  if the program has terminated (i.e., the execution of the command has finished).

$\Gamma = \{ (c, \sigma) \mid c \in Cmd, \sigma \in \Sigma \} \cup \Sigma$

Copia Modifica

- So, a configuration can represent:

- A **pair of a command and a state** — which means there is still some work left to do (the command hasn't been fully executed yet).
- A **state alone** — which represents the **final state** after the command has been executed and the program has terminated.

so, when we write:  $(c, \sigma) \rightarrow (c', \sigma')$ , when in state  $(c', \sigma')$  it doesn't actually mean we completed the execution of  $c$ , otherwise we would be in a final state. Correct?

Yes, you're absolutely right! Let's clarify that:

When you write  $(c, \sigma) \rightarrow (c', \sigma')$ , this means that you've **executed the command  $c$  partially or in some way, but you have not completed its execution yet**. So:

# Notations

☞ arithmetic, boolean ...

- We use  $ev(a)\sigma$  to denote the value obtained by evaluating the expression  $a$  in the state  $\sigma$
- Example: if  $\sigma = \{a = 3, b = 4\}$  then  $ev(a + b)\sigma = 7$
- The notation  $\sigma[a = 5]$  is used to denote the operation of updating the value of variable  $a$  in the state  $\sigma$
- Example: if  $\sigma = \{a = 3, b = 4\}$  then  $\sigma[a = 5] = \{a = 5, b = 4\}$

↓  
This is part of the  
next

# Small steps transitions

**Cmd**  $\ni c ::= x := a \mid \text{skip} \mid c_1 ; c_2$   
 $\mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$

Easy for **atomic commands**:

Executing **skip** terminates in one step and doesn't modify the state

We have  $(\text{skip}, \sigma) \rightarrow \sigma$

Executing an assignment terminates in one step, and updates the state

We have  $(x := a, \sigma) \rightarrow \sigma[x = ev(a)\sigma]$

↑  
command

# What about compound statements?

Consider the sequential composition

$$(c_1; c_2, \sigma) \rightarrow ???$$

Intuition: we have to first execute statement  $c_1$  in small-step fashion and then  $c_2$ .

The transition relation needs to be defined inductively!

## Inductive Definition:

- Inductive means that the definition of the transition relation will be built step-by-step, typically based on the structure of the commands. It's often used in cases where you need to define how complex structures (like commands or programs) can be evaluated by breaking them down into simpler cases.
- For example, in your case with sequential composition ( $c_1; c_2$ ), you need to define how the transition happens for this composite command based on what happens with each individual command. This would be defined by handling the transitions for:
  1. Executing  $c_1$ : You first define the transition relation for how  $c_1$  executes (perhaps in small steps).
  2. Executing  $c_2$ : Once  $c_1$  completes, the next transition defines how  $c_2$  executes.

In this context, induction refers to mathematical induction or structural induction, which is a technique used to define or prove things based on smaller, simpler cases. When applied to defining the transition relation for commands, induction means that you define the behavior of more complex commands (like  $c_1; c_2$ ) in terms of their simpler components (like  $c_1$  and  $c_2$ ).

# Rule system

The standard mathematical machinery to handle inductive definition is provided by **rule systems**

A rule has the form

$$\frac{\text{premise}_1, \dots, \text{premise}_n}{\text{conclusion}} \quad \text{if (side - conditions)}$$

↑ Premises have to be derived by the same rule

it reads: "**If premises hold, and if side conditions are satisfied, then conclusion holds**". Premises must be, in turn, justified by rules.

The **conclusion holds whenever there is a finite derivation tree whose leaves are axioms** (rules without premisses).

↓  
Conclusion holds without any other conditions?

## Key terms:

1. **Derivation Tree:** A **derivation tree** is a tree structure where each node represents an application of a rule, and the leaves (the terminal nodes) are **axioms** or **basic rules** that don't have any premises to be satisfied.
  - The **root** of the tree represents the **conclusion** (the final state or result of applying rules).
  - The **edges** represent the application of rules to get from one step to the next.
  - The **leaves** are the axioms, which are the base cases or foundational rules.
2. **Axioms:** These are **rules without premises**—they are the fundamental truths or **base cases** of the system. In other words, you **don't need to justify axioms with other rules**. They're the starting points of your derivation.
  - For example, in **operational semantics**, a rule for a simple command like `x := 5` could be an **axiom** because it has no further substeps to decompose.
3. **Finite Derivation:** The **finite derivation tree** refers to a tree that eventually reaches an **end** (finite, not infinite), starting from the conclusion and breaking down into the axioms or base rules.

$$\frac{\frac{[y : A \wedge B]}{A} \text{(A)} \quad \frac{[x : A \Rightarrow B \Rightarrow C]}{B \Rightarrow C} \text{(A)}}{B \Rightarrow C} \text{(\wedge E)} \quad \frac{\frac{[y : A \wedge B]}{B} \text{(A)} \quad \frac{C}{A \wedge B \Rightarrow C} \text{(\Rightarrow I, y)}}{B} \text{(\wedge E)} \text{(\Rightarrow E)} \\
 \frac{C}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)} \text{(\Rightarrow I, x)}$$

---

Example: propositional logic

# Small step transition ... inductively

The rules of the transition relation are of the form

$$\frac{\gamma_1 \rightarrow \gamma'_1, \dots, \gamma_n \rightarrow \gamma'_n}{\gamma \rightarrow \gamma'}$$

In order to derive transition from  $\gamma$  to  $\gamma'$ , we need to observe the premises

Program is characterized by a syntax. So we know the basic statements that compose compound statements (basic: skip and assignment). You pick up an inductive structure from these two entities, which we are able to deal.

## Structured Operational Semantics (1)

$$\frac{-}{(x := a, \sigma) \rightarrow \sigma[x = ev(a)\sigma]}$$

ASG

$$\frac{\text{Assignment or skip are defined by themselves using transformation}^*}{(skip, \sigma) \rightarrow \sigma}$$

Skip state  
transformation\*

$$\frac{(c_1, \sigma) \rightarrow \sigma'}{(c_1; c_2, \sigma) \rightarrow (c_2, \sigma')}$$

SEQ1

$$\frac{(c_1, \sigma) \rightarrow (c'_1, \sigma')}{(c_1; c_2, \sigma) \rightarrow (c'_1; c_2, \sigma')}$$

SEQ2

\* Those are the axioms for transition relation system.

These two rules explain control flow.

To understand  $c_1; c_2$ , we execute  $c_1$  first (math. explanation of PC). If the step in  $c_1$  allow us to terminate, good. What remains is to execute  $c_2$  with the state in which we ended up after  $c_1$ .  $c_2$  can start only after  $c_1$  terminates.

Rule 2:

We first perform first step

$$\sigma = \{x=1, y=2, C=3\}$$

$$x = x + 1; \underset{c_1}{y} = y + x; \underset{c_2}{C} = y + z; \underset{c_3}{z} = 7;$$

$c_1; c_2; c_3 \rightarrow$  your execution

Imagine you have:

$((c_1; c_2); c_3, \sigma)$ . If we applied SEQ1:

$$\frac{(c_1; c_2, \sigma) \rightarrow \sigma'}{((c_1; c_2); c_3, \sigma) \rightarrow (c_3, \sigma')}$$

But this is wrong. We have to apply SEQ2:

$$\frac{(c_1; c_2, \sigma) \rightarrow (\bar{c}_1, \sigma')}{((c_1; c_2), c_3, \sigma) \rightarrow (\bar{c}_1; c_3, \sigma')}$$

## 2. SOS and Inductive Definitions:

SOS often employs an **inductive definition** to define the operational semantics of a programming language.

In this context, **inductive definition** means defining the semantics of complex constructs by starting with the most basic constructs and recursively applying rules to describe more complex constructs.

- **Base Case:** The simplest constructs (like basic assignments or expressions) are defined directly by transition rules.
- **Inductive Step:** More complex constructs (like compound statements or loops) are defined in terms of simpler constructs using **recursive rules**.

# Structured Operational Semantics (2)

$$\frac{ev(b)\sigma = \text{true}}{(if\ b\ then\ c_1\ else\ c_2, \sigma) \rightarrow (c_1, \sigma)}$$

IF-TT   Evaluation of expression does not change states

$$\frac{ev(b)\sigma = \text{false}}{(if\ b\ then\ c_1\ else\ c_2, \sigma) \rightarrow (c_2, \sigma)}$$

IF-FF



# Structured Operational Semantics (3)

---

$$\frac{}{(while\ b\ do\ c, \sigma) \rightarrow (if\ b\ then\ \{c; while\ b\ do\ c\}\ else\ skip, \sigma)}$$

WHI



↙ re-execute while

PROPERTY of a WHILE: The meaning of a while is defined with while itself. It's like a rectangle.

→ execution in an abstract machine

A small-step **execution** of a *While* command is a sequence of configurations

$\gamma_0, \dots, \gamma_p, \dots$  such that, for each  $i$ ,  $\gamma_i \rightarrow \gamma_{i+1}$  ↗  
↳ you impose this

We write :

- \* Reflexive and transitive closure of → : finite number of transitions
- + Transitive closure of → : finite, non-zero number of transitions
- <sup>i</sup> Exactly  $i$  transitions

Execution of  $(c, \sigma)$  is said

- ▶ to **terminate** iff there exists  $\sigma'$  such that  $(c, \sigma) \rightarrow^* \sigma'$
- ▶ to **loop** iff there exists an infinite transition sequence starting from  $(c, \sigma)$

## 2. $\rightarrow^*$ (Reflexive and transitive closure of $\rightarrow$ ):

- This means zero or more transitions.
- Reflexive: If no transition happens ( $c, \sigma$  is already in its final state), it's still valid.
- Transitive: If multiple transitions happen, it's still valid.
- Example: If a program takes 3 steps to reach a final state, we write  $(c, \sigma) \rightarrow^* \sigma'$ .

## 3. $\rightarrow^+$ (Transitive closure of $\rightarrow$ ):

- This means one or more transitions.
- Unlike  $\rightarrow^*$ , this does not allow for zero transitions—at least one step must occur.
- Example:  $(c, \sigma) \rightarrow (c', \sigma') \rightarrow (c'', \sigma'') \rightarrow^+ \sigma''$ .

## 4. $\rightarrow^i$ (Exactly $i$ transitions):

- This means that the execution takes precisely  $i$  steps.
- Example: If we know a computation takes exactly 5 steps, we write  $(c, \sigma) \rightarrow^5 \sigma'$ .

# Forgetting intermediate states

The semantics we defined allows to view commands as state transformers. Formally, command  $c$  turns state  $\sigma$  into state  $\sigma'$  when

$$(c, \sigma) \xrightarrow{*} \sigma' \quad \text{There is a finite number of transitions}$$

We write

$$(c, \sigma) \Downarrow \sigma'$$

# Forgetting intermediate states

Can we define the relation

$$(c, \sigma) \Downarrow \sigma'$$

Inductively?

Behaviour of a sequential program is a memory transfor-

# Big Step Semantics

$\nearrow$  we forget intermediate states

$$\frac{-}{(x := a, \sigma) \Downarrow \sigma[x = ev(a)\sigma]}$$

ASG

$$\frac{(c_1, \sigma) \Downarrow \sigma'_1, (c_2, \sigma'_1) \Downarrow \sigma'}{(c_1 ; c_2, \sigma) \Downarrow \sigma'}$$

$$\frac{-}{(skip, \sigma) \Downarrow \sigma}$$

Skip  
 $\downarrow$  a skip command has no more problems

We compute the final state of executing  $c_1$  alone, then

SEQ  $c_2$  to the state of  $c_1$ .

The result we get  $\boxed{\sigma'}$

$$\frac{(c_1, \sigma) \Downarrow \sigma'_1, (c_2, \sigma'_1) \Downarrow \sigma'}{(c_1 ; c_2, \sigma) \rightarrow \sigma'}$$

# Biog Step Semantics (2)

$$\frac{ev(b)\sigma = \text{true } (c_1, \sigma) \Downarrow \sigma'}{(if\ b\ then\ c_1\ else\ c_2, \sigma) \Downarrow \sigma'} \quad \text{IF-TT}$$

$$\frac{ev(b)\sigma = \text{false } (c_2, \sigma) \Downarrow \sigma'}{(if\ b\ then\ c_1\ else\ c_2, \sigma) \Downarrow \sigma'} \quad \text{IF-FF}$$

## Big Step Semantics (3)

$$\frac{\begin{array}{c} \text{if guard is true, we now we have to evaluate the body} \\ \uparrow \\ ev(b)\sigma = \text{true } (c, \sigma) \Downarrow \sigma_1 (\text{while } b \text{ do } c, \sigma_1) \Downarrow \sigma_2 \end{array}}{(\text{while } b \text{ do } c, \sigma) \Downarrow \sigma_2} \quad \text{WHI-TT}$$

$$\frac{\begin{array}{c} \text{if guard is false, no while, provoke back state} \\ \uparrow \\ ev(b)\sigma = \text{false} \end{array}}{(\text{while } b \text{ do } c, \sigma) \Downarrow \sigma} \quad \text{WHI-FF}$$

# FROM OPERATIONAL RULES TO INTERPRETERS



# OCAML: A Language for writing interpreters

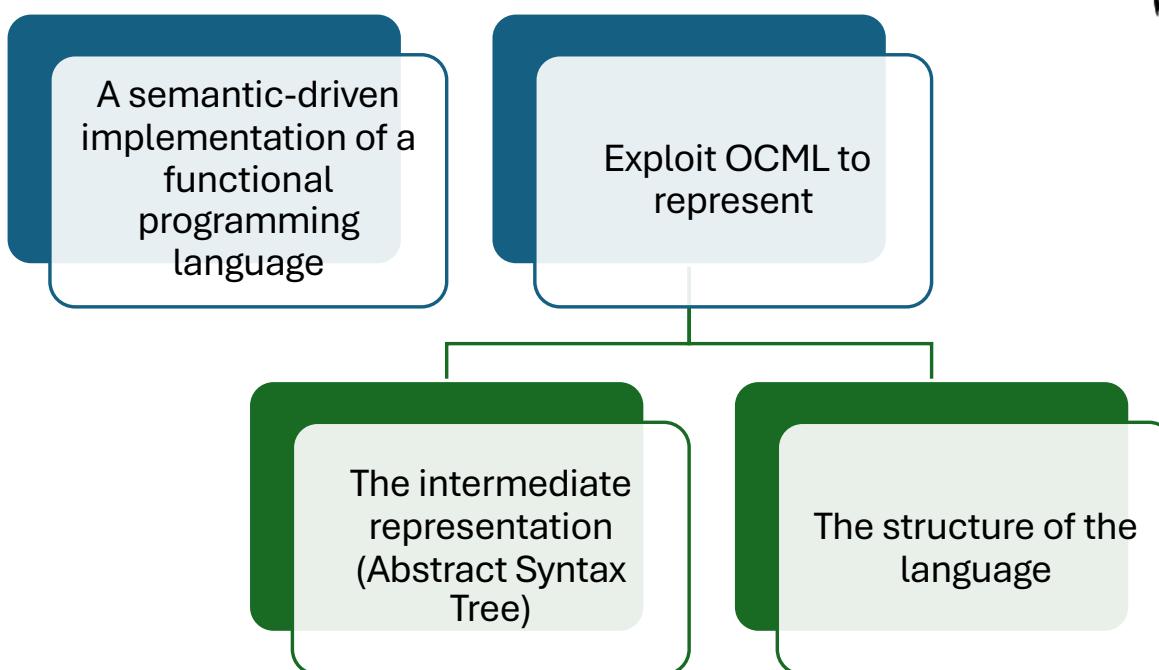
Functional programming languages have sometimes been called “domain-specific languages for compiler writers”

Datatypes are amazing for representing complicated tree-like structures and that is exactly what a program is.

Use a different constructor for every different sort of expression

- one constructor for variables
- one constructor for let expressions
- one constructor for numbers
- one constructor for binary operators, like add
- ...

# THE EXAMPLE



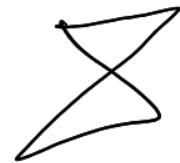
Walk with big step

We can represent an abstract syntax easily on OCaml

## The program

**Let x = 5 in  
(x+2)\*(x-18)**

CONCRETE SYNTAX



Will be represented by the expression

ABSTRACT SYNTAX

```
Let("x",
  Eint 5,
  Times(Sum(Eint 2, Den "x"), Minus(Den "x", Eint 18))
)
```

# ALGEBRAIC TYPES AND ABSTRACT SYNTAX TREES



This is how to derive type expressions,  
identifiers

```
# type ide = string
type exp = Eint of int
| Den of ide
| Sum of exp*exp
| Times of exp * exp
| Minus of exp * exp
| Let of ide * exp * exp;;
```

```
type ide = string
type exp =
  Eint of int
| Den of ide
| Sum of exp * exp
| Times of exp * exp
| Minus of exp * exp
| Let of ide * exp * exp
```

```
# Let("x",
  Eint 5,
  Times(Sum(Eint 2, Den "x"), Minus(Den "x" 18))
)    ;;
```

```
- : exp =
Let ("x", Eint 5, Times (Sum (Eint 2, Den "x"), Minus (Den "x", Eint 18)))
#
```

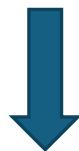
# Evaluation: Steps of the Interpreter

AST

Let("x",  
    Eint 5,  
    Times(Sum(Eint 2, Den "x"),  
          Minus(Den "x", Eint 18, ))  
)



Times(Sum(Eint 2, Den "x"),  
        Minus(Den "x", Eint 18, ))



We need to evaluate a  
LET: let declares that x  
is bound to 5 and you  
use this binding in the  
body of the LET.

x	5
---	---

RUN TIME  
DATA STRUCTURE  
STACK

But to manage this  
declaration we need  
a data structure  
that manages this  
binding of variables!

(Close to SECD machines)

# Binding & scope (you well know...)



With the term **binding** we mean an association between a name and a language entity (function, data structure, object, etc.).

The **scope** of a binding defines that part of the program in which the binding is active.

And to bind identifiers to values, you use  
the environment

---

The **environment** is defined as  
the set of name-entity binding  
existing at run time at a specific  
point in the program and at a  
specific time of execution

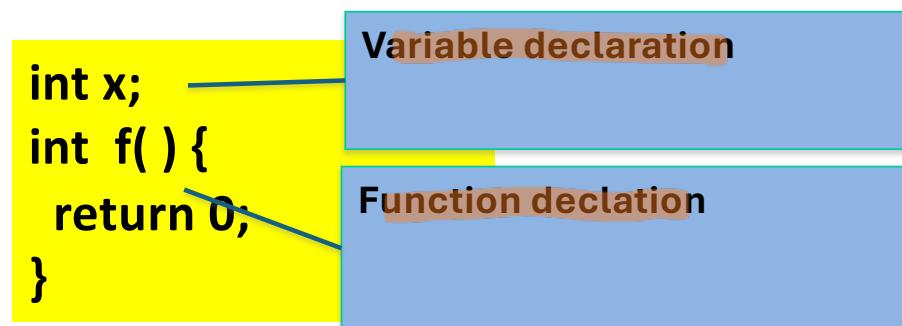
---

In the abstract language  
machine, for each name and for  
each section of the program,  
the environment determines  
the correct association

# Environment

# Environment & Declaration

- Which are the linguistic construct that allows associations to be introduced into the environment?



You can declare a type, a class. Depends on the expressive power of the language

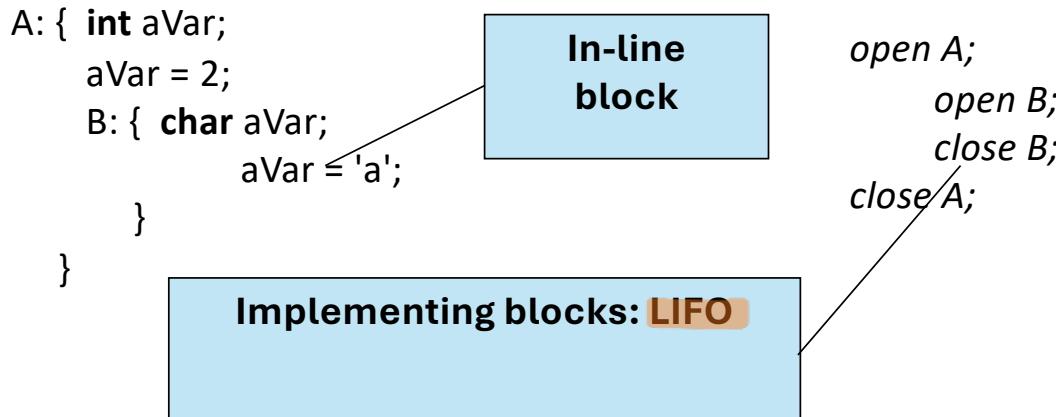
**type BoolExp =**

- | True
- | False
- | Not of BoolExp
- | And of BoolExp\*BoolExp

The environment in the abstract machine as soon as you have a declaration (so when using a new name, associated to a value).

Semantically this means that any variable you declare is associated to an initial value, cause you want to avoid possibility of having declarations of names without any value associated.

# Blocks



***Changes in the environment occur upon entering  
and leaving blocks (including nested blocks)***

block is like function without name and parameter passing -  
You manage it exactly in the same way -

# Environments

- *Local environment: the set of locally declared associations, including the associations related to parameters*
- *Non-local environment: associations for names that are visible within the block but not declared in the block itself*
- *Global environment: associations for names that can be used by all components of the program*



# OCAML SIMULATION OF THE ENVIRONMENT

**Environment** : The environment is a function that maps variables to values

- The environment **env** is a collection of **bindings**
  - **env = {x = 25, y = 6}**
- **env** contains two “bindings”
  - Binding between **x** and the value **25**
  - Binding between **y** and the value **6**
  - **z** is not bound in **env**
- The **env type** function that goes from domain of identifiers of the language into the domain of value enriched by the Unbound value  
**Ide → Value + Unbound**
- The constant **Unbound** makes the env function a total function

## 1. "The env type: `Id` -> Value + Unbound"

This describes the **type signature** of the environment (`env`), which is a function that maps **identifiers** (variables) to **values**. The expression `Id -> Value + Unbound` can be explained as follows:

- `Id` represents an identifier (or variable). This is typically a **string** or some other unique name that represents a variable.
- `Value` represents the value associated with that identifier. This could be an integer, string, or any other type of value, depending on the context.
- `Unbound` is a special marker that indicates that a variable has not been assigned a value yet (i.e., the variable is uninitialized or undefined in the environment). It's a way to handle cases where a variable is used before it is given a value.

## 2. "The constant Unbound makes the env function a total function"

A **total function** is a function that is **defined for every possible input**. In this case:

- The environment function `env` is defined for every possible **identifier** (`Id`), whether that identifier is bound to a value or not.

# Environment

- **env: Ide  $\rightarrow$  Value + Unbound**
- **env(x)** denotes either the value v bound to x in env or the special value **Unbound**
- **env[x=v]** denotes the environment
  - **env[x=v](y) = v if y = x**
  - **env[x=v](y) = env(y) if y != x**
- Assume env = {x = 25, y = 7} then  
 $\text{env}[x=5] = \{x = 5, y = 7\}$

8

- In dumb list are pairs inside brackets
- OP1 is lookup of a name.
  - :: is the cons operation

# Implementation (simple)

```

let emptyenv = [] empty environment
(* the empty environment *)

let rec lookup env x =
  match env with
    | []           -> failwith ("not found")
    | (y, v)::r -> if x = y then v else lookup r x
      ↪ list       ↪ otherwise of is a list; name y and value v, add r, the rest of
      ↪ list       ↪ the list.
let bind env x val = (x val)::env
  ↪ add a new association.
  ↪ we put (x val) at the beginning of env.

```

If  $x$  is  $y$  we finished, we move to recursively looking to  $r$ .

add a new association.

\* We put  $(x \text{ val})$  at the beginning of  $\text{env}$ .

*examples*

```
let emptyenv = [];;
val emptyenv : 'a list = []
```

(\* the empty environment \*)

```
let rec lookup env x =
  match env with
  | []    -> failwith ("not found")
  | (y, v)::r -> if x = y then v else lookup r x;;
val lookup : ('a * 'b) list -> 'a -> 'b = <fun>
```

```
let bind env (x:string) (v:int) = (x,v)::env;;
val bind : (string * int) list -> string -> int -> (string * int) list =
<fun>
```

Syntax of the language we are considering

**type ide = string**

**type exp =**

- | **CstInt of int**
- | **CstTrue**
- | **CstFalse**
- | **Times of exp \* exp**
- | **Sum of exp \* exp**
- | **Sub of exp \* exp**
- | **Eq of exp \* exp**
- | **Iszero of exp**
- | **Or of exp \* exp**
- | **And of exp \* exp**
- | **Not of exp**

- | **Den of ide**
- | **Ifthenelse of exp \* exp \* exp**
- | **Let of ide \* exp \* exp**
- | **Fun of ide list \* exp**
- | **Apply of exp \* exp list**

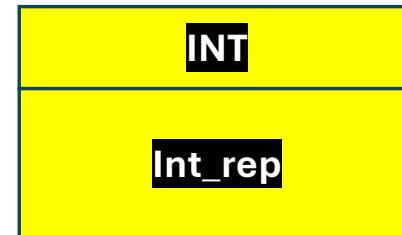
How to apply op. semantics

When you have to implement a programming language, you also have to implement which are the data available at run-time. These are what is called the values expressible inside of a language, the values you write in your data structures at runtime execution. In our implementation of interpreters we have integers and boolean. INT because we have a Type descriptor in the abstract machine. ①

## Expressible values: the values of the run-time

- Expressible values (result of the evaluation of expressions)

```
type evT = Int of int  
| Bool of bool  
| Unbound
```



environment: ide  $\rightarrow$  evT env

Type descriptor  
Sequence of bits, so you need to know how to interpret them.

If you do not have static analysis you need this.

① In the abstract machine I have a stack of bits and a descriptor telling me the type. We don't do static analysis in interpreters so in the run time we need meta information to represent types.

## 1. Expressible Values (evT)

The "expressible values" are the possible results that your interpreter can produce when evaluating an expression. These are the runtime values, meaning they are what the program will produce when it runs.

```
ocaml
```

```
type evT =  
| Int of int          (* for integer values *)  
| Bool of bool        (* for boolean values *)  
| Unbound             (* for unbound values (i.e., variables that have not been assigned a value) *)
```

Copia Modifica

- `evT` is a **type definition** in OCaml. It's defining a type that represents the possible **expressible values** in the language you are interpreting.

- `Int of int`: An **integer value** (e.g., `Int(5)` represents the integer 5).
- `Bool of bool`: A **boolean value** (e.g., `Bool(true)` or `Bool(false)`).
- `Unbound`: A **special value representing an unbound variable** (i.e., a variable that hasn't been assigned any value).



This is what I meant when I said `env` is a **function type**:

```
ocaml
```

```
type env = ide -> evT
```

Copia Modifica

Here's what this means:

- `ide`: This is the **input type**, which represents an **identifier** (a variable name). For example, it might be a string like `"x"`.
- `evT`: This is the **output type**, which represents the value associated with that identifier (e.g., `Int` for an integer, `Bool` for a boolean, or `Unbound` if the identifier isn't in the environment).

So the type `env` represents a function that takes an **identifier** (`ide`), and **returns an expressible value** (`evT`).

# The Operational rules drive iInterpreter impl.

$\text{CstInt}(n) \Downarrow \text{Int}(n)$

$\text{CstTrue} \Downarrow \text{Bool(true)}$

Interpreter is called eval

On the

↑ Those rules describe how constant integers/booleans  
expressions are evaluated

**let rec eval (e: exp) = match e with**

| **CstInt(n) -> Int(n)**

| **CstTrue -> Bool(true)**

| **CstFalse -> Bool(false)**

| **Iszero(e1) -> ????**

| **Den(i) -> ???** Represents a variable lookup

Picking up the top level operator

↑ In reality takes AST of expression

NOTE BY ME!

- If the interpreter finds an integer constant, it evaluates it as an value of type integer

PRIMITIVE TYPES  
INT & BOOL

- Result of evaluating a constant integer is the evaluation of the integer  
 $\text{Int}(m)$  is the representation of the integer  $m$  inside the machine  
(we need a representation). + method for how to interpret that string of bits

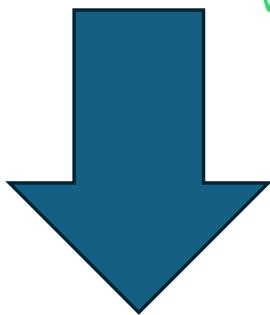
$\text{Dom}(s)$ : Takes value associated to identifier  $s$ . Do we need some data structure?

Yes! Env.

For binding a variable I have to define a rule. On the top we have a natural rule for the binding.

## Binding of variables

$$\frac{\underbrace{env(ide) = val}_{\text{I need to look up what is the value associated to ide and provide it back as a result}} \quad \text{Promise: environment has a mapping for the Operational Rule} \quad \text{Variable ide to value val}}{(ide, env) \Downarrow env(ide)}$$



Code: rephrase in prog. language which are the words.  
**Interpreter Step**

`eval(Den(ide), env)) -----> lookup ide env`

## (ide, env) as the Input:

When you see (ide, env) in the operational rule, it means you are evaluating the expression ide in the context of the environment env.

- ide : This is a variable or identifier (e.g., x, y, etc.) whose value you are trying to find or evaluate.
- env : This is the environment, which is a collection of variable bindings (e.g., x = 5, y = true), essentially a mapping of variable names to their corresponding values.

$$\underline{\text{env}(\text{ide}) = \text{val}}$$

$$(\text{ide}, \text{env}) \Downarrow \text{env}(\text{ide})$$

The result is the val we get from looking up the environment

Notation:

value

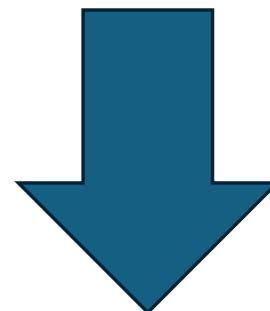
# Evaluation of primitive operations

$$\frac{(e_1, \text{env}) \Downarrow v_1 \cdot (e_2, \text{env}) \Downarrow v_2}{(e_1 + e_2, \text{env}) \Downarrow v_1 + v_2}$$

Annotations:

- A handwritten note above the first  $\Downarrow$  says "Then we have to evaluate".
- An arrow labeled "expression" points to the  $e_1$  term.
- An arrow labeled "result" points to the  $v_1 + v_2$  result.
- A handwritten note above the  $v_1$  term says "produce the result".

**Operational Rule**



**Interpreter Step**

`eval(Sum(e1, e2), env)` ----- >

`int_plus*((eval e1 env), (eval e2 env))`

Direct translation of rule in the code

## Explanation:

This rule is defining how to evaluate the expression  $(e_1 + e_2)$  in the language. Here's the step-by-step breakdown:

- Premises:

- $(e_1, \text{env}) \Downarrow v_1$  means that when you evaluate the expression  $e_1$  in the environment  $\text{env}$ , you get the result  $v_1$ .
- $(e_2, \text{env}) \Downarrow v_2$  means that when you evaluate the expression  $e_2$  in the environment  $\text{env}$ , you get the result  $v_2$ .

- Conclusion:

- The overall expression  $(e_1 + e_2)$  will evaluate to  $v_1 + v_2$  as long as  $v_1$  and  $v_2$  are both numbers that can be added.
- So, the result of evaluating  $(e_1 + e_2)$  in the environment is the sum of the results of  $v_1$  and  $v_2$ .

If we are trying to execute sum of two expressions, + is the basic primitive performing sum for the machine for integers.  
\* We need to provide implementation for integers

# Typechecking (dynamic)

```
let typecheck (type, typeDescriptor) =  
  match type with  
    | "int" ->  
      (match typeDescriptor with  
        | Int(u) -> true  
        | _ -> false)  
    | "bool" ->  
      (match typeDescriptor with  
        | Bool(u) -> true  
        | _ -> false)  
    | _ -> failwith ("not a valid type");;  
  
val typecheck : string * evT -> bool = <fun>
```

In order to apply sum to values  
we need to typecheck the expected  
type with the one associated to a  
value

Now there is a problem. Imagine  $e_1$  evaluates to a boolean. Because we do not perform static type checking, we need in the implementation of Typechecken

# Abstract Machine: Primitive Ops

```
let is_zero x = match (typecheck("int",x), x) with
| (true, Int(y)) -> Bool(y=0)
| (_, _) -> failwith("run-time error");;
```

```
let int_eq(x,y) =
  match (typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Bool(v = w)
  | (_,_,_,_) -> failwith("run-time error ");;
```

```
let int_plus(x, y) =
  match(typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_,_,_,_) -> failwith("run-time error ");;
```

# Abstract Machine: Primitive Ops

```
let is_zero x = match (typecheck("int",x), x) with
| (true, Int(y)) -> Bool(y=0)
| (_, _) -> failwith "is_zero"
```

```
let int_eq(x,y) =
  match (typecheck("int",x), typecheck("int",y)) with
  | (true, true) -> Bool(x=y)
  | (_, _) -> failwith "int_eq"
```

```
let int_plus(x, y) =
  match(typecheck("int",x), typecheck("int",y)) with
  | (true, true) -> Int(x+y)
  | (_, _) -> failwith "int_plus"
```

The basic operations are implemented through an eager evaluation rule:  
before applying the operator, all subtrees (subexpressions) are evaluated

work in a call by value style  
(but not for the booleans because of short circuit eval)

In this language we follow an eager evaluation rule

This means that in the context of the interpreter you're working with, when performing operations (like addition, multiplication, etc.), all subexpressions are first fully evaluated before the actual operation is applied.

## Eager evaluation:

In an **eager evaluation** strategy (also called **strict evaluation**), the operands or subexpressions of an operation are evaluated as soon as the operation is encountered. This is in contrast to **lazy evaluation**, where the evaluation of expressions is deferred until their results are actually needed.

## What it means in your context:

When you apply an operator (such as `+`, `*`, etc.) to an expression, before you actually compute the result, the interpreter will first evaluate the subexpressions that make up that operation.

For example, let's say you have an expression like this:

# Conditional

$$\frac{\begin{array}{c} \text{Stack} \\ \uparrow \\ ev(b)env = \text{true } (e_1, env) \Downarrow env' \\ (if b \text{ then } e_1 \text{ else } e_2, env) \Downarrow env' \end{array}}{\text{if } b \text{ then } e_1 \text{ else } e_2, env \Downarrow env'}$$

• if results:  
↑  
Then

$$\frac{\begin{array}{c} ev(b)env = \text{false } (e_2, env) \Downarrow env' \\ (if b \text{ then } e_1 \text{ else } e_2, env) \Downarrow env' \end{array}}{(if b \text{ then } e_1 \text{ else } e_2, env) \Downarrow env'}$$

\* You don't have that in the rule of operational semantics because a rule says: if the premises hold, the conclusion can be derived. So here we cannot apply the rules.

Ifthenelse(cond,e1,e2) ->

```
let g = eval cond env in  
match (typecheck("bool", g), g) with  
| (true, Bool(true)) -> eval e1 env  
| (true, Bool(false)) -> eval e2 env  
| (_, _) -> failwith ("nonboolean guard")
```

Correct type check

cannot apply rule but  
you have to expect  
the possibility.

You cannot apply  
rules so that's a  
failure.

the conditional does not follow  
an eager strategy: the evaluation  
of the subtree is based on the evaluation  
of the guard

Important difference: what is the AST of an IFE? Here evaluation of condition does not follow an eager strategy. First we evaluate the guard, then the Then or else blocks.

START WEEK 3

# Let: operational semantics

Let introduces a binding. New name in the environment. Let at top level of AST takes name x declared, to express to be evaluated for the binding and then the body of the block.

$$\frac{(e_1, \text{env}) \Downarrow v_1 \quad (e_2, \text{env}[x = v_1]) \Downarrow v_2}{(\text{Let}(x, e_1, e_2), \text{env}) \Downarrow v_2}$$

NEW BINDING

WE PROVIDE THE OVERALL RESULTS

To evaluate let  $x = e_1$  in  $e_2$ :

1. Evaluate  $e_1$  in the current environment to a value  $v_1$ .
2. Evaluate  $e_2$  in the environment containing the binding between  $x$  and  $v_1$  to a value  $v_2$ .
3. The result of evaluating the let expression in the current environment is  $v_2$ .

Blocks are functions with/o names and parameters. So you work with AR

## Let operational semantics (intuition)

env =  
run-time  
stack env=stack

push RA su env

Push activation record, which is binding  
 $x = v_1$

$$\frac{(e_1, \text{env}) \Downarrow v_1 \quad (e_2, \text{env}[x = v_1]) \Downarrow v_2}{(\text{Let}(x, e_1, e_2), \text{env}) \Downarrow v_2}$$

pop env

at the end we pop the AR

# The interpreter step

Recursive. Function can call itself if needed

```
let rec eval((e: exp), (env: evT env)) =  
  match e with  
    :  
  | Let(i, e, eboby) ->  
    eval eboby (bind env i (eval e env))
```

Type + type constructor (probably from note gives val)

expression → Parameters → environment, mapping from variable names to values of type evT

↳ This is how we define functions !!!

right, because the interpreter doesn't work on code like  
"x = a  
fun = function definition"  
but it translates that into its representation right? Like Let function  
be this and we will pass those parameters and execute. Correct?

Exactly! The interpreter doesn't work on raw OCaml-like syntax; instead, it operates on an abstract representation of the program, which breaks everything down into structured elements like Let, Fun, Apply, and so on.

## Is `env` (in `evT env`) a variable or a function?

No, it's neither a variable nor a function. The `env` here is a type constructor.

### What is a type constructor?

A type constructor is like a template that takes a type and produces another type.

For example, with:

ocaml

Copia Modifica

```
type 'v env = (string * 'v) list
```

- `'v env` means "a list of (string, 'v) pairs".
- If you replace `'v` with `evT`, you get `evT env`, which means "a list of (string, evT) pairs".

So `env` is a type constructor, not a value or function.



Ex.

$C_1 = \text{if } x > 10 \text{ then } X := X + Y \text{ else } X := X - Y;$

$C_2 = \text{while } x > 5 \text{ do } X := X - z;$

$$\sigma = \{x=15, y=5, z=16\}$$



$$(P_1, \sigma) \rightarrow ?$$

||

$$(C_1; C_2, \sigma) \rightarrow ?$$

We consider AST of program and walk through it to see how it is changing execution ①

①

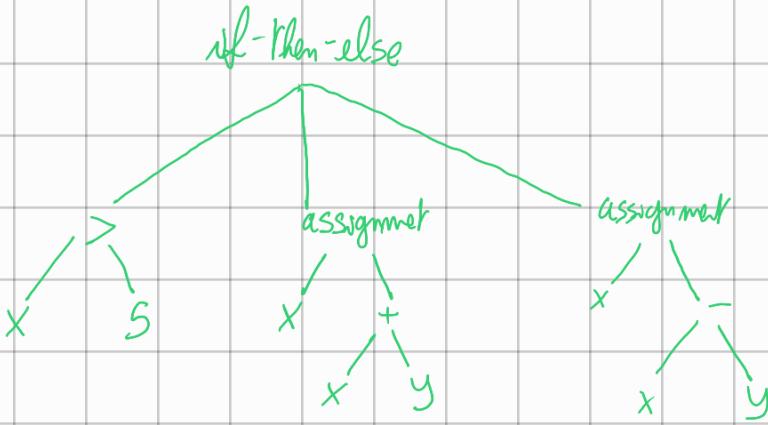


So, what rule do we have to execute for  $C_1$ ?

$$\frac{(C_1, \sigma) \rightarrow ?}{(C_1; C_2, \sigma)}$$

So, what is the AST of  $C_1$ ? For a condition you need to supply a guard ( $x > 10$ ), and syntactically you need to supply a Then and Else branch.





- In  $C_1$ , the top level operator is if-Then-else.

$$\begin{array}{c}
 \text{assignment} \\
 \swarrow \quad \searrow \\
 x \quad + \quad y \\
 \text{---} \\
 (x := x + y, \sigma) \rightarrow \sigma[x = ev(x+y)\alpha] \Rightarrow \sigma[x = 20] = \{x = 20, y = 5, z = 16\} \\
 \hline
 (C_1, \sigma) \rightarrow \sigma' \\
 \hline
 (C_1; C_2, \sigma') \rightarrow (C_2, \sigma')
 \end{array}$$

↑ while doesn't have premises

$$(C_2, \sigma') \rightarrow (\text{if } (x > 5) \text{ then } \{x := x - z\}; C_2 \text{ else skip}, \sigma')$$

You have to think of AST and in terms of top level operation of your tree.

$(GUARD, \sigma) = \text{true}$

$(THEN, \sigma') \rightarrow \sigma'$

$(IFE, \sigma') \rightarrow \sigma'$

$(C_1, \sigma) \rightarrow \sigma'$

$(C_1, C_2, \sigma) \rightarrow (C_2, \sigma')$

We need to understand what this will be

IT remains to define and apply functions



## Functions

### Functional abstraction

- Fun of ide \* exp<sup>↳ body</sup>  
*↳ function takes one parameter*

### Application

- Apply of exp \* exp  
*↳ Dem("f") ↳ in functional prog. languages, Apply takes an role in our case  
↳ In cell of we use apply*

# Functional abstraction

**Anonymous Functions**: to associate name you use *Let*

- **Fun("x", body)**
- **"x" formal parameter,**
- **fbody body of the function**

## Ocaml expressions

**let f x = x+7 in f 2**

**Let("f",** *for the declaration*

**Fun("x", Sum(Den("x"), CstInt(7))),**

**Apply(Den("f"), CsInt(2))**

)

*Abstract syntax tree associated*

## 2. Why is a function called a "functional abstraction"?

A "functional abstraction" refers to the idea that we are abstracting away the details of the function's implementation. Here, the word "abstraction" means that we are **concealing the implementation details** of a function behind its interface (its parameter list and what it returns).

In functional programming, a **function** is considered an **abstraction** because:

- It **encapsulates a piece of behavior** (the code inside the function).
- It **hides the details of how the behavior is implemented**, and you **only interact with the function through its signature** (i.e., the parameters and return type).

In this sense, the "functional abstraction" is the way functions are treated as **first-class values** (you can pass them around, return them from other functions, etc.) and are **abstracted** from the specifics of their implementation.

# 1. Understanding the Slide's Goal

This slide shows:

- How **anonymous functions** are represented.
- How **OCaml expressions** can be rewritten in a more structured format.
- How functions are **defined, stored, and applied** in an interpreter.

## 2. Breaking Down the Given OCaml Code

The OCaml expression given is:

```
ocaml ✖ Copia ⚙ Modifica
let f x = x + 7 in f 2
```

This means:

1. Define a function **f**:

```
ocaml ✖ Copia ⚙ Modifica
f x = x + 7
```

- **f** takes an argument **x** and returns **x + 7**.

2. Apply **f** to **2**:

```
ocaml ✖ Copia ⚙ Modifica
f 2 ↓
```

### 3. How This is Represented in a Formal Notation

Now, let's break down the notation:

less

Copia Modifica

```
Let("f",
  Fun("x", Sum(Den("x"), CstInt(7))),
  Apply(Den("f"), CstInt(2)))
```

This is how the interpreter structures the expression internally.

#### Step 1: Define `f`

ocaml

Copia Modifica

```
Let("f", Fun("x", Sum(Den("x"), CstInt(7))), ...)
```

- `Let("f", ...)` → We define `f` inside a scope.
- `Fun("x", Sum(Den("x"), CstInt(7)))` →
  - This represents the function `f x = x + 7`.
  - "`x`" is the **formal parameter** (the variable that `f` takes).
  - `Sum(Den("x"), CstInt(7))` means  $x + 7$ .
  - `Den("x")` represents a **variable reference** (meaning we retrieve `x`'s value when needed).
  - `CstInt(7)` represents the **constant integer 7**.

#### Step 2: Apply `f` to `2`

ocaml

Copia Modifica

```
Apply(Den("f"), CstInt(2))
```

- `Den("f")` → This represents retrieving the function `f` (since `f` was defined earlier).
- `CstInt(2)` → This is the argument we are passing to `f`.
- `Apply(...)` → This means we are calling `f` with `2`, so `x` gets replaced with `2`.

## 4. What Happens During Execution?

### Step 1: The let Expression

ocaml

Copia Modifica

```
Let("f", Fun("x", Sum(Den("x"), CstInt(7))), Apply(Den("f"), CstInt(2)))
```

- This stores **f** as a function in the environment.
- Now **f** is associated with the function **Fun("x", Sum(Den("x"), CstInt(7)))**.

### Step 2: The Apply Expression

ocaml

Copia Modifica

```
Apply(Den("f"), CstInt(2))
```

- We fetch **f** from the environment.
- We apply **f** to **2**, meaning we substitute **x** with **2**.

# A first step

- For simplicity we assume that the functional application is of the first order
  - The first argument of the functional application must be the name of the function to be invoked
  - **Apply(e,arg)** must be of the form **Apply(Den("f"), arg)**
- No recursion.

# 1. What is the Slide Saying?

This slide is describing some **simplifying assumptions** about function application in the interpreter:

## 1. "First-order functional application"

- Functions **only take one argument at a time**.
- We are **not dealing with higher-order functions** (functions that take other functions as arguments or return functions).

## 2. "The first argument must be the name of the function"

- We **cannot pass expressions** or function definitions directly.
- The function must be **already defined and stored in the environment**.

## 3. "Apply(e, arg) must be of the form Apply(Den("f"), arg)"

- This means that function application must look like:

```
ocaml
```

```
Apply(Den("f"), some_argument)
```

[Copia](#) [Modifica](#)

- Den("f") means that f must be a **named function retrieved from the environment**.
- You **cannot directly apply an anonymous function** like this:

```
ocaml
```

[Copia](#) [Modifica](#)

```
Apply(Fun("x", Sum(Den("x"), CstInt(7))), CstInt(2))
```

- Instead, you must first **assign it a name** using Let.

## 4. "No recursion"

- This means a function **cannot call itself**.
- If you write something like:

```
ocaml
```

[Copia](#) [Modifica](#)

```
let rec f x = if x = 0 then 1 else x * f (x - 1)
```

It **won't work** under these restrictions, because recursion requires fetching f from the environment while it's still being defined.

# The functional flavour

We assume static scoping (as it happens in Ocaml!!!)

What is the value of a **function**?

**type evT = | Int of int | Bool of bool | Unbound**

| Closure of ide \* exp \* evT env  
↳ speal value ↳ parameth ↳ body ↳ environment  
↳ static scope discipline: binding between name and variables

Values that are represented on the Absk-machine.

Functions are values in functional prog. lang.

the expressible value of a functional abstraction is a closure, which includes  
• name of the formal parameter (ide)  
• code of the declared function (exp)  
• environment at the time of declaration (evT env)

Static scoping: nonlocal references of the abstraction are solved in the function declaration environment

that are non local,  
are explicitly defined by the  
syntax of PL. So we need  
a mechanism at place to define  
when function has been declared!

## 1. What Does "Static Scoping" Mean?

Static scoping refers to how a programming language determines the environment in which variables are bound when a function is called. It is also known as lexical scoping.

- In static scoping, when you call a function, the interpreter looks at where the function is defined to find the values for variables used within the function. It doesn't rely on where the function is called.
- This is opposed to dynamic scoping, where the interpreter would look at the call stack to find variable bindings.

## 2. What is a Closure?

A closure is a data structure that represents a function along with the environment in which it was defined. A closure "closes over" its environment, meaning it captures the local bindings of variables at the time the function was created.

In this case, a closure consists of:

- `ide`: The name of the formal parameter of the function (the parameter the function takes, like `x` in `fun x -> x + 1`).
- `exp`: The body of the function (the expression that the function evaluates, like `x + 1`).
- `evT env`: The environment in which the function was defined. This is a mapping of variable names (like `x` or `y`) to their values at the time the function was declared.

So, when you define a function in a language with static scoping, it doesn't just have the code and parameters—it also remembers where it was created, i.e., the environment where all the variable bindings were in place.

## 3. Let's Look at the Type Declaration:

ocaml

Copia Modifica

```
type evT = | Int of int | Bool of bool | Unbound | Closure of ide * exp * evT env
```

Here, `evT` is a type that can represent different kinds of values in the environment.

- `Int of int`: An integer value.
- `Bool of bool`: A boolean value.
- `Unbound`: A value representing an unbound or undefined variable (to catch errors).
- `Closure of ide * exp * evT env`: This is a closure, representing a function. It includes:
  - `ide`: The name of the parameter.
  - `exp`: The body of the function (expression to evaluate).
  - `evT env`: The environment where the function was defined (the bindings of variables at that point).

## 4. The Value of a Function (Closure)

- In static scoping, when you define a function, you don't just get the function's code. You get a **closure**, which includes:
  - The **name of the formal parameter**.
  - The **body of the function** (expression).
  - The **environment at the time of the function's declaration**.

The **closure** is crucial because it allows the function to "remember" its environment, even when it's called later with different arguments.

## 5. What is Meant by "Static Scoping: Nonlocal References"

In static scoping, nonlocal references refer to variables that are **not local to the function** but are still used inside the function. These variables are looked up in the **environment where the function was defined**, not where it is called.

**Example:**

The screenshot shows an OCaml development environment. On the left, the OCaml code is written:

```
ocaml
let x = 5
let f = fun y -> x + y
let result = f 3
```

On the right, the translated code is shown in a corrected version:

```
ocaml
Let("y", CstInt(5),
  Let("f", Fun("x", Sum(Den("x"), CstInt(3))),
    Let("result", Apply(Den("f"), Prod(Den("y"), CstInt(2))),
      Den("result")
    )
  )
)
```

The interface includes buttons for 'Copia' (Copy) and 'Modifica' (Modify).

Here's how static scoping works:

- When `f` is defined, it **captures the environment** at that time, which includes the binding of `x = 5`.
- When `f 3` is called, it uses `x` from the environment where `f` was **defined** (static scoping), not the environment where it was **called** (which has no `x` binding).

So, the function `f` can access variables that were in scope at the time of its creation (like `x` in this case) even if they are **outside** the function's own parameters.

**To clarify:**

- **Static scoping** means that `f` will remember the **environment at the time it was defined**.
- If you try to reference a variable (`z` in this case) that wasn't in scope at the time of the function's **definition**, it will cause an **error** because the function doesn't have access to variables that weren't bound when it was defined.

# Operations Semantics: functions are values

$(\text{Fun}(x, e), \text{env} \Downarrow \text{Closure}("x", e, \text{env}))$

↑ Value associated to declaration of a function  
as RHS

An anonymous function is already a value.  
No computation to be performed.  
The essence of functional programming.

Function values are values just like `const(x)`

# All together now

$$\frac{Den("f", env) \Downarrow Closure("x", body, fDecEnv) \\ (arg, env) \Downarrow va \quad (fDecEnv[x = va], body) \Downarrow v}{(Apply(Den("f"), arg), env) \Downarrow v}$$

Value scope  
discipline

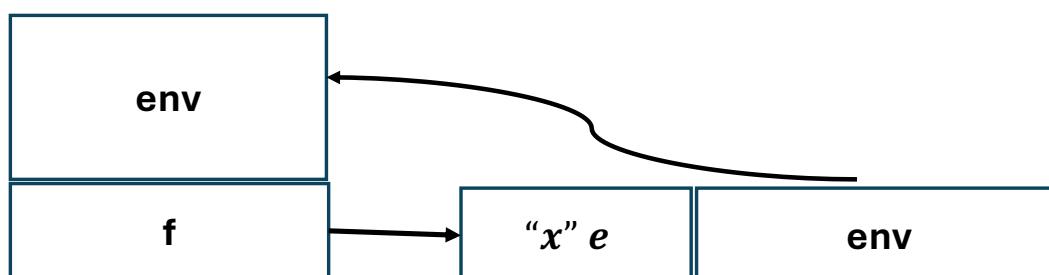
evaluate body in this environment

To evaluate f arg:

1. Lookup the environment to find the value (closure) bounded to f.
2. Evaluate the actual parameter arg in the current environment to get the actual value va (call-by-value)
3. Perform parameter passing to construct the actual execution environment by taking the declaration environment.
4. Evaluate the body of the function in the actual execution environment

# Pictorially

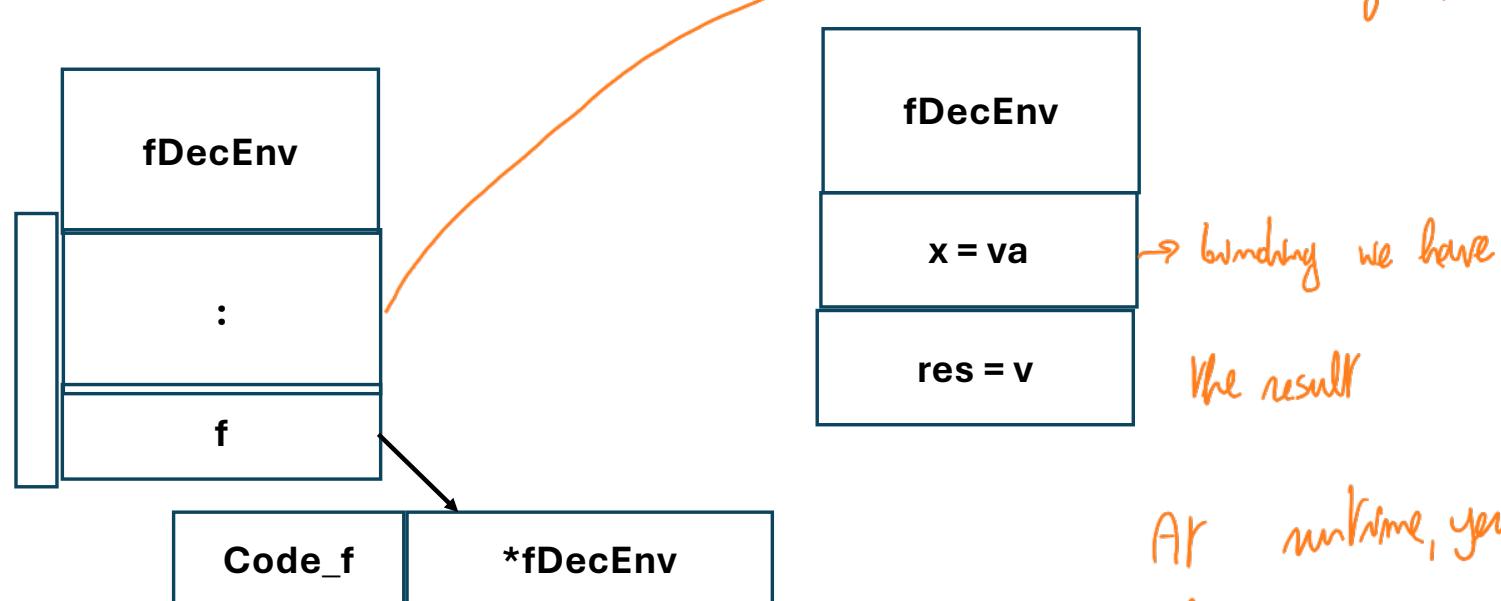
$(Fun(x, e), env \Downarrow Closure("x", e, env))$



Pointer to the beginning of  
the stack so you use base+offset

# Function Application

$$\begin{array}{c}
 \textcircled{1} \\
 \downarrow \quad \downarrow \\
 \begin{array}{c}
 \textit{Den}(“f”, env) \Downarrow \textit{Closure}(“x”, body, fDecEnv) \\
 (\textit{arg}, env) \Downarrow \textit{va} \quad (\textit{fDecEnv}[x = \textit{va}], \textit{bod}) \Downarrow \textit{v} \\
 \hline
 (\textit{Apply}(\textit{Den}(“f”), \textit{arg}), \textit{env}) \Downarrow \textit{v}
 \end{array}
 \end{array}$$



At runtime, you take only  
the fDecEnv so no local  
references are clear

# The interpreter

- Question: how do we represent definition of a function? It looks to me that this is just to lookup the function in the environment

```
let rec eval((e: exp), (env: evT env)) =
```

```
match e with
```

```
| ...
```

```
| Fun(i, a) -> Closure(i, a, env)
```

When encountering a function, we create a closure and this is how to interpret it

```
| Apply(Den(f), eArg) ->
```

```
    let fclosure = lookup env f in
```

lookup value in function name to get a closure

```
(match fclosure with
```

check if you have a closure (functional value)

```
    | Closure(arg, fbody, fDecEnv) ->
```

```
        let aVal = eval eArg env in
```

→ extend environment with value of actual env.

```
        let aenv = bind fDecEnv arg aVal in
```

→ binds in the decEnv the argument

```
        eval fbody aenv
```

```
    | _ -> failwith("non functional value"))
```

```
| Apply(_,_) -> failwith("Application: not first order function") ;;
```

↓ apply as not in the case described

Great question! The reason we use `eval eArg env` instead of `lookup eArg env` is because `eArg` is an **expression**, not necessarily a variable name. Let's break it down.

## 1. What is `eArg`?

- `eArg` represents the argument **expression** passed to the function.
- It could be a **constant**, a **variable**, or even another expression.

For example:

ocaml

Copia Modifica

```
let f = fun x -> x + 3
let y = 5
let result = f (y * 2)
```

When we encounter `Fun(i, a)`, which represents a function definition:

- `i` is the function's **parameter**.
- `a` is the **body** of the function.
- `env` is the **current environment** at the time of definition.

The result of evaluating a function definition is **not** running the function but instead creating a **closure**:

ocaml

Copia Modifica

```
| Fun(i, a) -> Closure(i, a, env)
```

This means that when we define a function, we **store**:

1. The **parameter** (`i`).
2. The **function body** (`a`).
3. The **environment at the time of definition** (`env`).

This is key to **static scoping**, because the function "members" the environment where it was defined, not where it is called.

## Dynamic scope

discipline (ex. JS). Value of function is not as a closure,  
but what is called efun: Just parameters and body.  
And work with current environment.

```
type evT = | Int of int | Bool of bool | Unbound  
           | Funval of efun
```

and efun = ide \* exp

The definition of efun shows that the functional abstraction  
contains only the code of the declared function

No need for environment. We use

The env at call

time

The body of the function will be evaluated in the obtained  
environment by binding the formal parameters to the actual  
parameter values in the environment in which the application  
takes place

## Operational Semantics

$$(Fun("x", e), env) \Downarrow Funval("x", e)$$

↑ ① ↓  
you care about this  
↑ body

$$\frac{(Den("f"), env) \Downarrow Funval("x", e) \quad (arg, env) \Downarrow va, (e, env[x = va]) \Downarrow v}{(Apply(Den "f"), env) \Downarrow v}$$

Apply(Den("f"), arg)

$\text{fun}("x", \text{body}) \Downarrow \text{Closure}("x", \text{body}, \text{env})$

# Interpreter

```
| Fun(arg, ebody) -> Funval(arg,ebody)
| Apply(Den(f), eArg) ->
  let fval = lookup env f in
    (match fval with
      | Funval(arg, fbody) ->
        let aVal = eval eArg env in
          let aenv = bind env arg aVal in
            eval fbody aenv
      | _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function");;
```

*actual environment, not declarative environment*

In Okmull everything is a function of Fun, Let, Apply etc. You do not declare variables, you declare functions and variables via let statements! So binding is done via let.

## Readings

- Notes on operational semantics are available on TEAMS