Most commonly-used programming languages are imperative: they consist of a sequence of actions that alter the state of the world. State includes the values of program variables and also the program’s external environment (e.g., files the program reads or writes).

High-level imperative languages mimic the style of the underlying Von Neumann machine architecture, where programs are sequences of instructions that modify the contents of registers and memory locations. This makes it relatively straightforward to compile imperative languages to efficient code:

- High-level variables are mapped to machine locations.
- High-level operations are mapped to (multiple) machine instructions.

Imperative languages are also natural for writing reactive programs that interact with the state of the “real world.” Examples:

- Reading mouse clicks and modifying the contents of a display.
- Controlling a set of relays in an external device.

**Assignment**

The basic primitive stateful operation is typically assignment, which alters a value stored in a location.

In the simplest form, the location is associated with a variable, e.g.,

\[ a := a + 2 \]

(Note: C/C++/Java popularized the use of plain `=` for assignment and `==` for relational equality: a bad idea, because both form expressions and are easy to confuse!)

In most languages, the variable name `a` means different things on the left-hand side (LHS) and right-hand side (RHS) of an assignment.

On the LHS, `a` denotes the location of the variable `a`, into which the value of the RHS expression is to be stored. Here we say `a` is an l-value.

On the RHS, `a` denotes the value currently contained in `a`, i.e., it indicates an implicit dereference operation. Here, `a` is an r-value.

If assignment is an expression, what is its value? Common choices are the value of the RHS, or a special "empty" value (e.g. `():Unit` in Scala).

**Initialization Values**

Most languages require variables to be declared before they are used: this gives them a scope, perhaps a type, and optionally, an initial value specified by an expression.

In fact, it is surely a bug to use any variable as an r-value unless it has previously assigned a value. But many languages permit us to write such code, resulting in runtime errors—either checked (e.g., as in Python) or unchecked (as in C).

The simplest fix is to require an initial value to be given for every declared variable (e.g. as in Scala).

Java takes a slightly more sophisticated approach:

- variables do not need to be initialized at the point of declaration; but
- they must be initialized before they are actually used.
Yet in any reasonably powerful language, checking initialization before use is an uncomputable problem. (Why?)

So the Java language reference manual carefully details a conservative, computable, set of conditions, which every program must meet, that guarantee there will be no uses before definition.

This is called the definite assignment property; just defining it takes 16 pages of the reference manual.

Some programs that do in fact initialize before use will be rejected because they violate the conditions.

Legal example:
```
int a;
if (b) /* b is boolean */
  a = 3;
else
  a = 4;
a = a + 1;
```

Illegal example:
```
int a;
if (b)
  a = 3;
if (!b)
  a = 4;
a = a + 1;
```

Order of stateful operations obviously affects program semantics.

Contrast:
```
x := 10;
x := 20;
y := x
```
```
x := 20;
x := 10;
y := x
```

Notice that we have introduced a sequencing operator, typically written as here with a semicolon (;

For expressions, the meaning of \( e_1; e_2 \) is

- evaluate \( e_1 \)
- throw away the result
- then evaluate \( e_2 \).

Obviously, this is only interesting if \( e_1 \) performs a stateful operation as a side-effect.

In many languages, order of expression evaluation is under-specified (precedence and associativity don’t always fix order).

ANSI C example:
```
a = 0;
b = (a = a + 1) - (a = a + 2);
```

What’s the result in b?

Answer: could be anything! In C, the order of evaluation of the operands to + is undefined. Consequently, the meaning of this program is also undefined, and a compiler is free to return any value it wants.

In fact, this is an illegal C program because of its ambiguous meaning, but many compilers will accept it without even giving a warning.

The impact of evaluation order on program behavior is not always obvious, because side-effects can occur at a distance:
```
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
h(a,f(2)); // = 0 or 2 ??
```

Answer: Depends on evaluation order for function actual parameters, which is language-dependent (and possibly unspecified).

Keeping expression evaluation order or argument evaluation order undefined sometimes lets compiler generate more efficient code.

But most modern languages (e.g., Java) have moved towards precise definition of evaluation order within expressions (typically left-to-right).
In functional languages, ordinary “variables” are immutable, i.e., they are really just names for values (computed just once at runtime), not for locations.

Making variables mutable would fundamentally change the character of a functional language. A gentler approach is to add mutable data structures.

The simplest possible structure is a single mutable cell containing a value. PAPL calls this a box. You can think of a box as a degenerate record with one field or array with one element.

The mutable cell is allocated and initialized by a constructor call, e.g.

```plaintext
x = box(42)
```

binds x to a location containing (initially) the value 42.

At an abstract level, we can temporarily ignore the question of where new cells are created, and simply say that the program has an extendible, mutable store, which maps locations to values.

The two operations on boxes are unbox, which gets the current box contents, and setbox, which changes the contents.

Box values are l-values; they must be explicitly dereferenced to get their contents, e.g.

```plaintext
let x = box(40) in
(setbox x (2 + unbox x); unbox x) (* yields 42 *)
```

This approach is obviously more verbose than conventional imperative code, but at least it removes any confusion between l-value and r-value.

Moreover, boxes are also first-class values, e.g.

```plaintext
let incby2 (y:int box) = setbox y (2 + unbox y) in
let z = box(40) in
(incby2(z); unbox z) (* yields 42 *)
```

The ML language provides boxes under the name “references.”

Using the store:

```plaintext
case class BoxV(a:Store.Location) extends Valuedef interpE(expr:Expr,env:Env,st:Store) : (Value,Store) = expr match {
...
case Box(e) => {
  val (v,st1) = interpE(e,env,st)
  val (a,st2) = st1 allocate 1 (BoxV(a),st2 + (a -> v))
}
case UnBox(b) => interpE(b,env,st) match {
  case (BoxV(a),st1) => (st1(a),st1)
  case _ => throw TypeError("Attempt to unbox a non-box")
}
case SetBox(b,e) => interpE(b,env,st) match {
  case (BoxV(a),st1) => {
    val (v,st2) = interpE(e,env,st1)
    (v,st2 + (a -> v))
  }
  case _ => throw TypeError("Attempt to set a non-box")
}
...
}
```

To model the semantics of a language with boxes, we add a Store component to the interpreter. A store is simply an (immutable) record containing

- A map from Locations to Values
- A counter recording the next free Location

```plaintext
object Store {
  type Location = Int
  val empty = Store(0,Map[Location,Value]())
}
case class Store(nextAddr : Store.Location,
    contents : Map[Store.Location,Value]) {
  type Location = Int
def allocate(n:Int) : (Location,Store) =
    (nextAddr,Store(nextAddr+n,contents))
def apply(a:Location) = contents(a) // exception if absent
def +(av:(Location,Value)) = Store(nextAddr,contents + av)
}
```
Modelling Mutable Variables:

```scala
type Env = Map[String,Store.Location]
def interpE(expr:Expr,env:Env,st:Store) : (Value,Store) = {
... case Let(x,d,b) => {
  val (v,st1) = interpE(d,env,st)
  interpE(b,env + (x -> v),st1)
}
...}
def arithBinOp(l: Expr, r:Expr,env:Env,st:Store)
(op: (Int,Int) => Value) = {
  val (lv,st1) = interpE(l,env,st)
  val (rv,st2) = interpE(r,env,st1)
  (lv,rv) match {
    case (NumV(ln),NumV(rn)) => (op(ln, rn),st2)
    case _ => throw TypeError("...")
  }
  ...
}
```

In most imperative programming languages, **variable** names are bound to **locations**, i.e. memory addresses, which in turn contain **values**. So processing a variable declaration (e.g. a parameter or local) conceptually involves two separate operations:

- allocating a new location in the store and perhaps initializing its contents;
- creating a new binding in the environment from the variable name to the location.

Assignment to the variable involves an implicit update to the store, and a use of the variable as an r-value involves an implicit fetch from the store.

The store threading already described works just the same for variables as for boxes!

Key idea: Once created, environment bindings don’t change.

**Semantics of Variables**

So far, our interpreted languages do not have a way to build interesting **data structures**. (Boxes can only store a single value.)

We can remedy this by adding in just one new kind of value, the **pair**. You can think of a pair as a record with two fields, each containing a value — which might be a base value such as integer or another pair.

We write pairs using “infix dot” notation. For example:

```
(1 . ((2 . 3) . 4))
```

corresponds to the structure:

```
1
   4
  /  \\
2 3
```

We can build larger records of a fixed size just by nesting pairs.
We can also build all kinds of interesting arbitrary-sized **recursive** structures using pairs. For example, to represent **lists** we can use a pair for each link in the list. The left field contains an element; the right field points to the next link, or is 0 to indicate end-of-list.

Example:

```
[1,2,3]
(1.(2.(3.0)))
```

Note that for programs to detect when they’ve hit the end of a list, they’ll need a way to distinguish the value 0 from any pair value.

Pairs and lists built from them are the entire basis of the LISP language and its descendents.

---

**Boxed vs. Unboxed**

For example, consider an array of 100 integers. In an **unboxed** representation, the array would be represented directly by 100 words holding the array contents. In a **boxed** representation, the array would be indirectly represented by an implicit 1-word pointer to 100 consecutive locations holding the array contents.

The language’s choice of representation makes a big difference to the semantics of operations on the data, especially if values are mutable. Key questions include:

- What does assignment mean?
- How does parameter passing work?
- What do equality comparisons mean?

---

**Pragmatics of Large Values**

Real machines are very efficient at handling small, fixed-size chunks of data, especially those that fit in a single machine **word** (e.g. 16-64 bits), which usually includes:

- Numbers, characters, booleans, enumeration values, etc.
- Memory addresses (locations).

But often we want to manipulate larger pieces of data, such as records and arrays, which may occupy many words.

There are two basic approaches to representing value:

- The **unboxed** representation holds the actual contents of the value, using as many machine words as necessary.
- The **boxed** representation **implicitly** allocates storage (the “box”) for the contents in memory, and then represents the value by the **location** of that storage.

(Note: this use of the term “boxed” applies to our interpreter’s box values, but also to many other kinds of values, including much larger ones.)

---

**Unboxed Representation Assignment Semantics**

Earlier languages often used unboxed representations for records and arrays. For example, in Pascal and related languages,

```pascal
TYPE Employee =
    RECORD
        name : ARRAY (1..80) OF CHAR;
        age : INTEGER;
    END;
```

specifies an unboxed representation, in which value of type Employee will occupy 84 bytes (assuming 1 byte characters, 4 byte integers).

The semantics of assignment is to copy the entire representation. Hence

```pascal
VAR e1,e2 : Employee;
    e1.age := 91;
    e2 := e1;
    e1.age := 19;
    WRITE(e1.age, e2.age);
```

prints 19 followed by 91.
**UNBOXED REPRESENTATION PROBLEMS**

Assignment using the unboxed representation has appealing semantics, but two significant problems:

- Assignment of a large value is expensive, since lots of words may need to be copied.
- Since compilers need to generate code to move values, and (often) allocate space to hold values temporarily, they need to know the size of the value.

These problems make the unboxed representation unsuitable for value of arbitrary size. For example, unboxed representation can work fine for pairs of integers, but not for pairs of arbitrary values that might themselves be pairs.

**Boxed Representation Assignment Semantics**

Most modern languages (e.g. Java, Python, Haskell) implicitly allocate memory for all objects (i.e. objects, records, constructed values) that are larger than a word, and represents values of these types by references (pointers, locations) into memory.

As a natural result, these languages use so-called reference semantics for assignment and argument passing. Scala example:

```scala
case class emp(var name:String, var age:Int)
object Bat {
  def main(argv:Array[String]) = {
    val e1 = emp("fred",91)
    val e2 = e1
    e1.age = 19
    println(e1.age + " " + e2.age)
  }
}
```

prints 19 19

**Explicit Pointers**

Many languages that use unboxed semantics also have separate explicit pointer types to enable programmers to construct recursive data structures, e.g. in C++

```cpp
struct Intlist {
  int head;
  Intlist *tail;
};
Intlist *mylist = new Intlist;
mylist->head = 42;
mylist->tail = mylist;
delete mylist;
```

In C/C++, struct and class instances are fundamentally unboxed, i.e. they use unboxed assignment semantics. But the programmer usually boxes them explicitly (using `new` or `malloc`) and manipulates them via explicit pointers.
Languages typically provide some form of built-in equality testing on values. So when are two (large) values equal?

- One possible definition is: when their contents are equal, field by field. This **structural equality** is the only sensible definition for unboxed values.
- For boxed values, we could instead say: when their locations are identical. This **reference equality** is more efficient, but might be less useful.

Note that reference equality implies structural equality, but not vice-versa.

Some languages provide both, under different names. They may also provide a standard way for the programmer to define equality on a given type in an ad-hoc way. For example, in Scala, the `eq` operator gives reference equality, whereas the `==` operator invokes a user-defined `equals` method; for case classes, the latter is pre-defined to be structural equality.

---

**CALL-BY-VALUE**

- As we've seen already, each actual parameter is **evaluated** to a **value** before call.
- On entry to the function, each formal parameter is bound to a freshly-allocated location, and the actual parameter value is **copied** into that location. (This is much like processing the declaration and initialization of a local variable.)
- Updating the formal parameter doesn’t affect actuals in **calling** procedure.
  ```c
  double hyp(double a, double b) {
    a = a * a;
    b = b * b;
    return sqrt(a+b);
  }
  ```
  - Semantics are just like for assignment.
  - Simple; easy to understand!

---

**PROCEDURE PARAMETER PASSING**

When we apply a function in an imperative language, the formal parameters get bound to locations containing values.

- How is this done and which locations are used?
- Do we pass addresses or contents of variables from the caller?
- How do we pass actual values that aren’t variables?
- What does it mean to pass a large value like an array?

Two main approaches:

- call-by-value (CBV)
- call-by-reference (CBR)

---

**PROBLEMS WITH CALL-BY-VALUE (1)**

- Can be inefficient for large unboxed values:

Example (C): Calls to `dotp` copy 20 doubles

```c
typedef struct {double a1, a2, ..., a10;} vector;

double dotp(vector v, vector w) {
  return v.a1 * w.a1 + v.a2 * w.a2 + ... + v.a10 * w.a10;
}
vector v1, v2;
double d = dotp(v1, v2);
```
**Problems with Call-by-Value (2)**

- Cannot affect calling environment directly.

Example: calls to `swap` have no effect:

```c
void swap(int i, int j) {
    int t;
    t = i; i = j; j = t;
}
...swap(a[p], a[q]);
```

(Of course, perhaps this is usually a good thing!)

- Can at best return only one result (as a value), though this might be a record.

**Call-by-Reference**

- Pass the existing location of each actual parameter.
- On entry, the formal parameter is bound directly to this location. Thus, it can be dereferenced to get the value, but it can also be updated.
- If actual argument doesn’t have a location (e.g., `(x + 3)`), either:
  - Evaluate it into a temporary location and pass address of temporary, or
  - Treat as an error.
- Now `swap`, etc., work fine!
- Lots of opportunity for aliasing problems, e.g.,

  ```c
  PROCEDURE matmult(a,b,c: MATRIX)
  ...
  (* sets c := a * b *)
  matmult(a,b,a) (* oops! *)
  ```

- **Call-by-value-result** (a.k.a. copy-restore) addresses this problem, but has other drawbacks.

**Hybrid Methods; Records and Arrays**

How might we combine the simplicity of call-by-value with the efficiency of call-by-reference, especially for large unboxed values?

- In Pascal, Ada, and similar languages, where records and arrays are both unboxed, the programmer can specify (in the procedure header) for each parameter whether to use call-by-value or call-by-reference.

- In C/C++, record (struct or class) values are unboxed, but arrays are boxed. C++ allows per-argument calling mode specification. C always uses call-by-value, but programmers can take the address of a variable explicitly, and pass that to obtain CBR-like behavior:

  ```c
  swap(int *a, int *b) {
      int t;
      t = *a; *a = *b; *b = t;
  }
  swap (&a[p], &a[q]);
  ```

Of course, it is the programmer’s responsibility to make sure that the address remains valid (especially when it is returned from a function).

**Values can be References**

In a language like Java or Python that boxes large values such as records or arrays, these values are already pointers (or references).

Thus, even if the language uses call-by-value, the values that are passed are actually references. Calls don’t cause any actual copying of the large values.

But it is a mistake (which some otherwise good authors make) to say that these languages use “call-by-reference.” (If they did, they would pass a reference to the reference!)

For example, the first interpreter in this week’s homework uses call-by-value, but boxes (and pairs) are among the possible values.
INFORMAL SEMANTICS

- Grammars can be used to define the legal programs of a language, but not what they do! (Actually, most languages place further, non-grammatical restrictions on legal programs, e.g., type-correctness.)
- Language behavior is usually described, documented, and implemented on the basis of natural-language (e.g., English) descriptions.
- Descriptions are usually structured around the language’s grammar, e.g., they describe what each nonterminal does.
- Natural-language descriptions tend to be imprecise, incomplete, and inconsistent.

EXAMPLE: FORTRAN DO-LOOPS

“DO \ n \ i \ = \ m_1, m_2, m_3
Repeat execution through statement \( n \), beginning with \( i = m_1 \), incrementing by \( m_3 \), while \( i \) is less than or equal to \( m_2 \). If \( m_3 \) is omitted, it is assumed to be 1. \( m \)'s and \( i \)'s cannot be subscripted. \( m \)'s can be either integer numbers or integer variables; \( i \) is an integer variable.”
Consider:
```
  DO 100 I = 10,9,1
  ...  
  100 CONTINUE
```

How many times is the body executed?

EXPERIMENTAL SEMANTICS

Try it and see!
Implementation becomes standard of correctness.
This is certainly precise: compiler source code becomes specification.
But it is:
- difficult to understand;
- awkward to use;
- subject to accidental change;
- wholly non-portable.

FORMAL SEMANTICS

Aims:
- Rigorous and unambiguous definition in terms of a well-understood formalism, e.g., logic, naive set theory, etc.
- Independence from implementation. Definition should describe how the language behaves as abstractly as possible.

Uses:
- Provably-correct implementations.
- Provably-correct programs.
- Basis for language comparison.
- Basis for language design.
(But usually not basis for learning a language.)

Main varieties: Operational, Denotational, Axiomatic.
Each has different purposes and strengths. In this course, we’ll mostly focus on operational semantics, with brief looks at the others.
OPERATIONAL SEMANTICS

Define behavior of language on an abstract machine.

Abstract machine should be much simpler than real machines, since otherwise a compiler for a real machine would be just as good!

Typical mechanisms:

- Characterize the state of the abstract machine (typically as an environment mapping variables to values) and give a set of evaluation rules describing how each syntactic construct affects the state.
- Define a simple Von Neumann-style stack machine and describe how each syntactic construct can be compiled into stack machine instructions.

Some useful things to do with an operational semantics:

- Build an implementation for a real machine by interpreting or compiling the abstract machine code.
- Explicate the meaning of a language feature by proving that it has the same behavior as a combination of simpler features.
- Prove that correctly typed programs cannot “dump core” at runtime.

SEMANTICS FROM INTERPRETERS

In the homework, we’re building definitional interpreters for small languages that display key programming language constructs.

Our goal is to study the interpreter code in order to understand implementation issues associated with each language.

In addition, the interpreter serves as a form of semantic definition for each language construct. In effect, it defines the meaning of the language in terms of the semantics of Scala.

(Of course, you’ll also be learning more about the semantics of Scala as we go!)

SEMANTICS AND ERRONEOUS PROGRAMS

An important part of a language specification is distinguishing valid from invalid programs.

It is useful to define three classes of errors that make programs invalid. (Of course, even valid programs may behave differently than the programmer intended!)

Static errors are violations of the language specification that can be detected at compilation time (or, in an interpreter, before interpretation begins)

- Includes: lexical errors, syntactic errors (caught during parsing), type errors, etc.
- Compiler or interpreter issues an error pinpointing erroneous location in source program.
- Language semantics are usually defined only for programs that have no static errors.

CHECKED RUNTIME ERRORS

Checked runtime errors are violations that the language implementation is required to detect and report at runtime, in a clean way.

- Examples in Scala or Python: division by zero, array bounds violations, dereferencing a null pointer.
- Depending on language, implementation may issue an error message and die, or raise an exception (which can be caught by the program).
- Language semantics must specify behavior precisely.

Unchecked runtime errors are violations that the implementation need not detect.

- Subsequent behavior of the computation is arbitrary. (Error is often not manifested until much later in execution.)
- Examples in C: division by zero, dereferencing a null pointer, array bounds violations.
- Language semantics probably don’t specify behavior.
- Safe languages like Python, OCaml, and Java have no such errors!
So far, we’ve presented operational semantics using interpreters. These have the advantage of being precise and executable. But they are not ideally compact or abstract.

Another way to present operational semantics is using state transition judgments, for appropriately defined machine states.

For example, consider a simple language of imperative expressions, in which variables must be defined before use, using a let or fun construct.

\[
\text{exp} := \text{var} \mid \text{int} \\
\mid '(\ '+\ \text{exp}\ '+\ ')' \\
\mid '(\ 'let\ 'var\ \text{exp}\ '+\ ')' \\
\mid '(\ ':=\ 'var\ '+\ ')' \\
\mid '(\ 'ifnz\ 'exp\ '+\ ')' \\
\mid '(\ 'fun\ 'var\ '+\ ')' \\
\mid '(\ '@\ 'exp\ '+\ ')' \\
\mid \text{etc.}
\]

For simplicity, we assume that the only values are integers and closures; ifnz expects an integer condition and tests whether it is non-zero.

To evaluate this language, we choose a machine state consisting of:

- the current environment \( E \), which maps each in-scope variable to a location \( l \).
- the current store \( S \), which maps each location \( l \) to an integer value \( v \).
- the current expression \( e \), to be evaluated.

We give the state transitions in the form of judgments:

\[ \langle e, E, S \rangle \Downarrow \langle v, S' \rangle \]

Intuitively, this says that evaluating expression \( e \) in environment \( E \) and store \( S \) yields the value \( v \) and the (possibly) changed store \( S' \).

Values \( v \) are either integers \( i \) or function closures \( [x, e, E] \), where \( x \) is the formal parameter, \( e \) is the body, and \( E \) is the static environment at the point of definition.

To describe the machine’s operation, we give rules of inference that state when a judgment can be derived from judgments about sub-expressions.

The form of a rule is

\[
\frac{\text{premises}}{\text{conclusion}} \quad \text{(Name of rule)}
\]

We can view evaluation of the program as the process of building an inference tree.

This particular formulation, where each expression results in a value based on the values of the subexpressions, is called big-step semantics.
### Evaluation Rules (1)

\[
\begin{align*}
& l = E(x) \quad v = S(l) \quad \text{(Var)} \\
\langle x, E, S \rangle & \Downarrow \langle v, S \rangle \\
\langle i, E, S \rangle & \Downarrow \langle i, S \rangle \quad \text{(Int)} \\
\langle e_1, E, S \rangle & \Downarrow \langle i_1, S' \rangle \quad \langle e_2, E, S' \rangle & \Downarrow \langle i_2, S'' \rangle \quad \text{(Add)} \\
\langle e_1, E, S \rangle & \Downarrow \langle v_1, S' \rangle \quad i_1 \notin \text{dom}(S') \\
\langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle & \Downarrow \langle v_2, S'' \rangle \quad \text{(Let)} \\
\langle e, E, S \rangle & \Downarrow \langle v, S' \rangle \quad l = E(x) \\
\langle := x e, E, S \rangle & \Downarrow \langle v, S' + \{l \mapsto v\} \rangle \quad \text{(Assign)} \\
\end{align*}
\]

### Evaluation Rules (2)

\[
\begin{align*}
\langle e_1, E, S \rangle & \Downarrow \langle i_1, S' \rangle \quad i_1 \neq 0 \quad \langle e_2, E, S' \rangle & \Downarrow \langle v_2, S'' \rangle \quad \text{(If-zero)} \\
\langle (\text{ifnz } e_1 e_2 e_3), E, S \rangle & \Downarrow \langle v_2, S'' \rangle \\
\langle e_1, E, S \rangle & \Downarrow \langle 0, S' \rangle \quad \langle e_3, E, S' \rangle & \Downarrow \langle v_3, S'' \rangle \quad \text{(If-zero)} \\
\langle (\text{ifnz } e_1 e_2 e_3), E, S \rangle & \Downarrow \langle v_3, S'' \rangle \\
\langle \text{fun } x e \rangle, E, S \rangle & \Downarrow \langle [x, e, E], S \rangle \quad \text{(Fun)} \\
\langle e_1, E, S \rangle & \Downarrow \langle x', E', S' \rangle \quad \langle e_2, E, S' \rangle & \Downarrow \langle v', S'' \rangle \quad l \notin \text{dom}(S') \\
\langle e', E' + \{x \mapsto l\}, S'' + \{l \mapsto v'\} \rangle & \Downarrow \langle v, S''' \rangle \quad \text{(Appl)} \\
\langle @ e_1 e_2, E, S \rangle & \Downarrow \langle v, S''' \rangle \\
\end{align*}
\]

### Notes on the Rules

- The structure of the rules guarantees that at most one rule is applicable at any point.
- The store relationships constrain the order of evaluation.
- If no rules are applicable, the evaluation gets stuck; this corresponds to a runtime error in an interpreter.

We can view the interpreter for the language as implementing a bottom-up exploration of the inference tree. A function like

\[
\text{def eval(Exp e, Env env) : Value = \{ .... \}}
\]

returns a value \(v\) and has side effects on a global store such that

\[
\langle e, \text{env}, \text{store before} \rangle \Downarrow \langle v, \text{store after} \rangle
\]

The implementation of \texttt{eval} dispatches on the syntactic form of \(e\), chooses the appropriate rule, and makes recursive calls on \texttt{eval} corresponding to the premises of that rule.

Question: how deep can the derivation tree get?