**Names and Binding**

One essential part of being a “high-level” language is having convenient **names** for things: operators, variables, constants, types, procedures, classes, etc.

- Allowed syntactic form of names varies for different languages, but intended to be human-readable.

We distinguish **binding** and **use** occurrences of a name.

- A **binding** makes an association between a name and the thing it names.
- A **use** refers to the thing named.

For example, in this ML code:

```ml
fun f (x:int) = 
  if (x > 0) then
    f(x + 1);
  else 0;
```

The first line binds both `f` (as a function) and `x` (as a formal parameter); the second line uses `x`; the third line uses both `f` and `x`.

It is common for some names to be **pre-defined** for all programs in a language, e.g., the type name `int` in the above example. Often the binding of these names is done in a standard library that is implicitly included in all programs.
Scoping Rules

A key characteristics of any binding is its **scope**: in what textual region of the program is the binding active?

- I.e., where in the program can the name be used?
- Alternatively: given a use of the name, how do we find the relevant binding for it?

In most languages, the scope of a binding is based on certain rules for reading the program text. This is called **lexical** scope. (Or **static** scope, because the connection between binding and use can be determined statically, without actually running the program.)

The exact rules for lexical scoping depend on the kind of name and the language.
Sample Scoping Rules

C provides some typical examples:

```c
static int x = 101;
bar (double y) {
    if (y > x)
        bar(y + 1.0);
}
main () {
    bar (3.14);
    { double w; /* inner block */
        w = x + x;
    }
}
```

- The pre-defined type names `int` and `double` are in scope throughout (all) C programs.
- `x` is in scope throughout this C file.
- `bar` is in scope from its point of definition to the end of the C file (including its own body).
- Similarly for `main`.
- `y` is in scope inside the body of `bar`.
- `w` is in scope inside the inner block of `main`.
Name Conflicts

What happens when the same name is bound in more than one place?

- If the bindings are to different kinds of things (e.g., types vs. variables), the language’s concrete syntax often gives a way to distinguish the uses, so no problem arises:

```c
typedef int z; /* z is a synonym for int */
z z = 3;
z w = z + 1;
```

- In this case, we say that types and variables live in different name spaces.

But conflicts can occur if we allow duplicate bindings within a single namespace.

- Some languages disallow this.

- More often, language allows holes in the scope of a binding; these are regions of the program where the binding is hidden by another binding of the same name.

- Sometimes additional rules (such as typing information) is used to determine which binding is meant. Names like this are said to be overloaded.
Hiding in Block-structured Languages

C Example:

```c
int a = 0;
int f(int b) {
    return a+b; /* use of global a */
}
void main() {
    int a = 1;
    print (f(a)); /* use of local a; prints 1 */
}
```

- Under this sort of scoping, we can find the binding relevant to a particular use by looking **up** the program text and **out** through any nesting constructs.

- These lexical scope rules are quite common, but there’s nothing magic about them; other languages may differ.
Dynamic Scope

There’s an alternative approach to scoping which depends on program execution order rather than just the static program text. Under **dynamic scoping**, bindings are (conceptually) found by looking backward through the program execution to find the most recent binding that is still active.

Same C Example:

```c
int a = 0;
int f(int b) {
    return a+b;
}
void main() {
    int a = 1;
    print (f(a));
}
```

- In this case, the use of `a` in `f` would refer to the local declaration of `a` within `main`.
- Global `a` isn’t used at all; result printed is 2.
- Early versions of LISP used dynamic scope, but it was generally agreed to be a mistake.
- Some scripting languages still use it (it is easier to implement than static scope in an interpreter).
**Free Names**

In any given program fragment (expression, statement, etc.) we can ask: which names are used but not bound? These are called the **free** names of the fragment.

The notion of **free** depends both on the name and the fragment. For example, given the C fragment

```c
int f (int x) {
    return x + y;
}
```

we say that `y` is free, but `x` is not. (What other names are free?)

However, in the sub-fragment

```c
return x+y;
```

we say that both `x` and `y` are free.

The **meaning** of a fragment clearly must depend on the values of its free names. To handle this, we usually give semantics to fragments relative to an **environment**.
Environments

An **environment** is just a mapping from names to their meanings, e.g., a map from variable names to values.

- We write \( E(x) \) means the result of looking up \( x \) in environment \( E \). (This notation is because an environment is (almost) just like a **function** taking a name as argument and returning a meaning as result.)

- We write \( E + \{x \mapsto v\} \) for the environment obtained from existing environment \( E \) by **extending** it with a new binding from \( x \) to \( v \). If \( E \) already has a binding for \( x \), this new binding hides it, but doesn’t replace it. (That’s why \( E \) isn’t quite like a function.)

Two less frequent operations:

- We write \( E/\{x := v\} \) to mean the environment obtained from \( E \) by **updating** the binding for \( x \) to \( v \). \( E \) must have a binding for \( x \) already; if it has more than one, the most recent extension is updated.

- We write \( E - \{x\} \) to mean the environment obtained from \( E \) by **dropping** the bindings for \( x \), which must be the most recent extension.

The **domain** of an environment, \( \text{dom}(E) \), is the set of names bound in \( E \).
Bindings, Values, Locations

So far, we’ve been deliberately vague about what gets associated to a name by a binding, because it varies a lot, depending on the kind of name.

For example:

- **function** names usually get bound to descriptions of the function’s parameters and body.

- **type** names get bound to a description of the type, including its possible values.

- **constant** names get bound to a value (either at compile time or at run time). “Variables” in purely functional languages work act like constants.

In most imperative programming languages, **variable** names are bound to **locations**, i.e. memory addresses, which in turn contain **values**. So declaring a variable typically involves two separate operations:

- creating a new location (e.g., on the stack or in the heap) and perhaps initializing its contents;

- creating a new binding from the variable name to the location.

For simplicity, we will temporarily omit this level of indirection, and say that variables are bound directly to values. (This is why we need to be able to **update** the environment.) But soon we’ll be more honest.
Formal Operational Semantics

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 local construct.

\[
\text{exp ::= } \text{var} \\
\quad | \text{int} \\
\quad | \text{'}\text{'}\text{'}+\text{' '}\text{exp}\text{' '}\text{exp}\text{' '}\text{'}\text{'} \\
\quad | \text{'}\text{'}\text{'}\text{local}'\text{ '}\text{var}\text{' '}\text{exp}\text{' '}\text{exp}\text{' '}\text{'}\text{'} \\
\quad | \text{'}\text{'}\text{'}:\text{'}\text{=}\text{'}\text{ '}\text{' '}\text{var}\text{' '}\text{exp}\text{' '}\text{'}\text{'} \\
\quad | \text{'}\text{'}\text{'}\text{if}'\text{ '}\text{exp}\text{' '}\text{exp}\text{' '}\text{exp}\text{' '}\text{'}\text{'} \\
\quad | \text{'}\text{'}\text{'}\text{while}'\text{ '}\text{exp}\text{' '}\text{exp}\text{' '}\text{'}\text{'} \\
\quad | \text{etc.}
\]

Informally, the meaning of (local x e₁ e₂) is: evaluate e₁ to a value v₁, create a new binding for x initialized to v₁, and evaluate e₂ in the resulting environment.
State Machine

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

- the current environment $E$, which maps each in-scope variable to an integer value $v$.
- the current expression $e$, to be evaluated.

We give the state transitions in the form of judgments:

$$
\langle e, E \rangle \downarrow \langle v, E' \rangle
$$

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

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.
Evaluation Rules

\[
\frac{x \in \text{dom}(E)}{\langle x, E \rangle \downarrow \langle E(x), E \rangle} \text{ (Var)}
\]

\[
\frac{}{\langle i, E \rangle \downarrow \langle i, E \rangle} \text{ (Int)}
\]

\[
\frac{\langle e_1, E \rangle \downarrow \langle v_1, E' \rangle \quad \langle e_2, E' \rangle \downarrow \langle v_2, E'' \rangle \quad v = v_1 + v_2}{\langle (+ \ e_1 \ e_2), E \rangle \downarrow \langle v, E'' \rangle} \text{ (Add)}
\]

\[
\frac{\langle e_1, E \rangle \downarrow \langle v_1, E' \rangle \quad \langle e_2, E' + \{x \mapsto v_1\} \rangle \downarrow \langle v_2, E'' \rangle}{\langle (\text{local } x \ e_1 \ e_2), E \rangle \downarrow \langle v_2, E'' - \{x\} \rangle} \text{ (Local)}
\]

\[
\frac{x \in \text{dom}(E)}{\langle (:= \ x \ e), E \rangle \downarrow \langle v, E'/\{x := v\} \rangle} \text{ (Assgn)}
\]

\[
\frac{\langle e_1, E \rangle \downarrow \langle v_1, E' \rangle \quad v_1 \neq 0 \quad \langle e_2, E' \rangle \downarrow \langle v_2, E'' \rangle}{\langle (\text{if } e_1 \ e_2 \ e_3), E \rangle \downarrow \langle v_2, E'' \rangle} \text{ (If-nonzero)}
\]

\[
\frac{\langle e_1, E \rangle \downarrow \langle 0, E' \rangle \quad \langle e_3, E' \rangle \downarrow \langle v_3, E''' \rangle}{\langle (\text{if } e_1 \ e_2 \ e_3), E \rangle \downarrow \langle v_3, E''' \rangle} \text{ (If-zero)}
\]

\[
\frac{\langle e_1, E \rangle \downarrow \langle v_1, E' \rangle \quad v_1 \neq 0 \quad \langle e_2, E' \rangle \downarrow \langle v_2, E'' \rangle}{\langle (\text{while } e_1 \ e_2), E'' \rangle \downarrow \langle v_3, E''' \rangle} \text{ (While-nonzero)}
\]

\[
\frac{\langle e_1, E \rangle \downarrow \langle 0, E' \rangle}{\langle (\text{while } e_1 \ e_2), E \rangle \downarrow \langle 0, E' \rangle} \text{ (While-zero)}
\]
Notes on the Rules

• This notation has similarities to axiomatic semantics: the notion of derivation is essentially the same, but the content of judgments is different.

• The structure of the rules guarantees that at most one rule is applicable at any point.

• The environment 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

\[
\text{Value eval(Exp e, Env env)} \{ \ldots \}
\]

returns a value \( v \) and has side effects on \( \text{env} \) such that

\[
\langle e, \text{env}_{\text{before}} \rangle \downarrow \langle v, \text{env}_{\text{after}} \rangle
\]

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

Question: how deep can the derivation tree get?
Representing Environments

To implement operational semantics as an interpreter, we need to choose a concrete representation for environments that supports the operations.

In particular, we want to make it easy to extend an environment with new bindings — which may hide existing bindings — while still keeping the old environment around. (This is useful so that we can enter and exit scopes easily.)

A simple approach is to use a singly-linked list, which is always searched from, and extended at, its head.

```c
int a = 0; /* env1 */
{
    int b = 1;
    int a = 2; /* env2 */
    a = a + b;
}

a = a + 1;
```

A more efficient approach is to use a balanced tree or hash table.
Pairs

So far our languages have just dealt with integer values. Real languages have more interesting values, such as records, unions, arrays, etc.

We’ll start by adding 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 an integer or another pair.

How much storage do we need for a pair? Potentially unbounded amounts!

So concretely, we’ll think of pairs as heap-allocated entities, represented by a pointer. When one pair “contains” another, it really contains a pointer to the other.

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

\[(1 \cdot ((2 . 3) . 4))\]

corresponds to the structure:

```
1
  |
  v
  4
  |
  v
  2 3
```
Lists and Trees

It turns out that we can build all kinds of interesting **recursive** structures using pairs.

**Lists:** 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:

\[ [1, 2, 3] \]

\[(1.(2.(3.0)))\]

Trees: Easy to encode binary trees with integer values at the leaves. (Other kinds of trees would require more complicated encodings.)

Example:

\[ ((1.2).((3.4).5)) \]