Part of being a “high-level” language is letting the programmer name things:

- variables
- constants
- types
- functions
- classes
- modules
- fields
- operators
- ...

Generically, we call names identifiers

An identifier binding makes an association between the identifier and the thing it names

An identifier use refers to the thing named

The scope of a binding is the part of the program where it can be used
Scala Example

```scala
object Printer {
  def print(expr: Expr) : String = unparse(expr).toString()

  def unparse(expr: Expr) : SExpr = expr match {
    case Num(n) => SNum(n)
    case Add(l,r) => SList(SSym('+')::unparse(l)::unparse(r)::Nil)
    case Mul(l,r) => SList(SSym('*')::unparse(l)::unparse(r)::Nil)
    case Div(l,r) => SList(SSym('/')::unparse(l)::unparse(r)::Nil)
  }
}
```

- Identifier syntax is language-specific
- Usually unbounded sequence of alpha|numeric|symbol(?)
- Further rules/conventions for different categories
- Identifiers are distinct from keywords! Some identifiers are pre-defined
Names, values, variables

- Most languages let us bind \textit{variable} names to memory cells (the \textit{store}) that contain \textit{values}.
- Name gives access to cell for read or update.
- Many languages also let us bind names \textit{directly} to (immutable) values computed by expressions.
- Sometimes (confusingly) also called “variables.”
- They let us \textit{share} expressions to save repeated writing and, maybe, evaluation.

Scala \texttt{var} vs. \texttt{val}
Local Value Bindings

\[
expr ::= num \mid expr + expr \mid \ldots \mid (expr) \mid id \mid let id = expr in expr
\]

(\text{let } a = 8 + 5 \text{ in } a * 3) + 3
Bound vs. Free

- A variable use $x$ is **bound** if it appears in the scope of a binding for $x$
- Otherwise, it is **free**
- Bound and free are relative to an enclosing subexpression, e.g.
  
  $a$ is bound in $(\text{let } a = 8 + 5 \text{ in } a * 3)$ but free in $a * 3$
- We cannot evaluate a free variable
Parallel Scopes

(\texttt{let } a = 8 + 5 \texttt{ in } a * 3) +
(\texttt{let } b = 1 \texttt{ in } b + 2)

What if both let’s bind a?
Nested Scopes

(let a = 8 + 5 in
  let b = a - 10 in
  a * b) + 2

scope_a

scope_b

scope_a

scope_a&b
(let \(a = 8 + 5\) in
let \(a = a - 10\) in
\(36 + a\)) + 3

"Nearest enclosing binding" wins
Functions and parameters

- Consider adding functions with parameters to our expression language

- We give names to these parameters
  - The scope of a parameter is the function body
  - The value of each parameter is provided at the function call (or “application”) site

```
@ f (* 13 3)
```

```
(f x (+ x 3))
```

### Diagram

**Declaration AST**
- function name
- formal parameter
- body

**Application AST**
- function name
- actual parameter
Function parameter scoping

\[(f \; x \; (+ \; x \; 3))\]
Function Name Scoping

- Typically, we want to allow functions to be recursive.
- Scope of function’s name includes its own body.

```
letfun f x = if x = 0 then 1 else x*f(x-1) in f(42)
```
Mutually Recursive Definitions

```plaintext
letrec f(x) = g(x + 1)
and g(y) = f(y - 1)
in f(2) + g(4)
```

Many earlier languages were designed to be compiled by a single pass through the source code and therefore use forward declarations

```c
void g (double y); /* declares g but doesn't define it */
void f(double x) { g(x+1.0); }
void g(double y) { f(y-1.0); } /* definition is here */
```

In some languages, all top-level definitions are (implicitly) treated as mutually recursive.
“Dynamic Scope”

What should happen in the following program?

```plaintext
letfun f(x) = x + y
in f(42)
```

How about this one?

```plaintext
letfun f(x) = x + y
in let y = 2
    in f(42)
```

One possible answer: let the value of y “leak” into f

This is an example of “dynamic scope”  Bad idea!

Why?
“Static scope”/“Lexical scope”

Better if this program remains erroneous

```
letfun f(x) = x + y
in let y = 2
  in f(42)
```

Looking at a function declaration, we can always determine if and where a variable is bound without considering the dynamic execution of the program!

Some scripting languages still use dynamic scope, but as programs get larger, its dangers become obvious.
Re-using names

What happens when the same name is bound twice in the same scope?

If the bindings are to different kinds of things (e.g. types vs. variables), can often disambiguate based on syntax, so no problem arises (except maybe readability):

```scala
type Foo = Int
val Foo : Foo = 10
val Bar : Foo = Foo + 1
```

Here we say that types and variables live in different name spaces

If the bindings are in the same namespace, typically an error. But sometimes additional info (such as types) can be used to pick the right binding; this is called **overloading**
Named scopes: modules, classes

Often, the construct that delimits a scope can itself have a name, allowing the programmer to manage explicitly the visibility of the names inside it.

OCaml modules

```ocaml
module Env = struct
  type env = (string * int) list
  let empty : env = []
  let rec lookup (e:env) (k:string) : int = ...
end
let e0 : Env.env = Env.empty in Env.lookup e0 "abc"
```

Java classes

```java
class Foo {
  static int x;
  static void f(int x);
}
int z = Foo.f(Foo.x)
```
Semantics via Environments

An environment is a mapping from names to their bindings.

The environment at a program point describes all the bindings in scope at that point.

Environment is extended when binding constructs are evaluated.

Environment is consulted to determine the meaning of names during evaluation.
Environments for everything

- Environments can hold binding information for all kinds of names

- A variable name is (typically) bound to location [in the store] containing the variable

- A value (constant) name may be bound directly bound to the value [environment = store]

- A function name is bound to description of the function’s parameters and body

- A type name is bound to a type description, including the layout of its values

- A class name is bound to a list of the class’s content

- Etc.
Variables, Environment, Store

In most imperative languages, variable names are bound to locations, which in turn contain values.

So creating a variable involves two things:

1. **allocating** a new store location (and possibly initializing its contents)

2. creating a new **binding** from the variable name to that location
Initialization Values

Many languages require variables to be declared before they are used: this gives them a scope, perhaps a type, and (maybe) an initial value given by an expression.

It is surely a bug to use any variable as an r-value unless it has been previously assigned a value.

But many languages let us write such code, resulting in runtime errors—either checked (e.g. as in Python) or unchecked (e.g. as in C).

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

- Java takes a more sophisticated approach
- Variables do not need to be initialized at the point of declaration, but
- They must be initialized before they are used

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

Legal example:

But checking initialization before use is uncomputable in general! (Why?)

A legal Java program
Definite Assignment

So the Java definition carefully details a conservative, computable, set of conditions, which every program must meet, that guarantee the absence of uses before definition.

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

Having these rules in the Java definition ensures portability.

Being conservative means that some programs that actually do initialize before use will be rejected.

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

an illegal Java program
Describing the Store

- In most languages, there are other ways to allocate storage too, such as explicit new operations or implicit boxing operations.

- Simplistic store model: mutable map from locations to values.

- Better models require distinguishing different classes of storage based on the lifetime of the data.
Storage Lifetimes

1. Typical computations use far more memory locations in total than they use at any one point.

2. So most language implementations support re-use of memory locations that are no longer needed.

3. The lifetime of every object should cover all moments when the object is being used.

4. Otherwise, we get a memory safety bug.
Storage Classes: Static Data

- **Lifetime** = Entire Execution

- Typically used for *global* variables and constants

  - If language has no recursion, can also be used for function-local variables

- **Fixed** address known before program executes

- **No** runtime allocation/deallocation costs
Storage Classes: Stack Data

- **Nested** Lifetimes (last allocated is first deallocated)
- Typically used for function-**local** variables (and internal control data for function calls)
  - Works because function call lifetimes also nest
- Allocation/deallocation are very **cheap** (just adjust the stack pointer)
- Produces good **locality** for caches, virtual memory
Storage Classes: Heap Data

- **Arbitrary** Lifetimes

- Typically used for explicitly allocated objects

- Some languages implicitly heap-allocate other data structures, e.g. bignums, closures, etc.

- Allocation/deallocation are relatively expensive

- Run-time library must decide where to allocate

- Deallocation can be done manually (risking memory bugs) or by a garbage collector
Scope, Lifetime, Memory Safety

- **Lifetime** and **scope** are closely connected.

- For a language to be **memory safe**, it suffices to make sure that in-scope identifiers never point (directly or indirectly) to deallocated objects.

- For **stack-allocated local** variables, this happens naturally.
  - Stack locations are deallocated only when function returns and its local variables go out of scope forever.

- For **heap** data, easiest to enforce safety using a garbage collector (GC).
  - GC typically works by recursively **tracing** all objects reachable from names that are currently in scope (or that might come back into scope later).
  - Only **unreachable** objects are deallocated, making their locations available for future re-allocation.
Explicit Deallocation

Many older languages (notably C/C++) support explicit deallocation of heap objects.

Somewhat more efficient than GC.

But makes language unsafe: "dangling pointer" bug occurs if we deallocate an object that is still in use [unchecked runtime error].

Converse problem: "space leak" bug occurs if we don’t deallocate an unneeded object.

Not a safety problem, but may unnecessarily make program run slower or crash with "out of memory" error.