Semantics and Erroneous Programs

Important part of language specification is distinguishing valid from invalid programs.

Useful to define three classes of errors that make programs invalid.

Of course, even valid programs may not act as the programmer intended!
Static Errors

- **Static errors** can be detected before the program is run (at compile or pre-interpretation time).
  - Includes **lexical** errors, **syntactic** errors, **type** errors, etc.
  - Error checker can give precise feedback about erroneous location in source code.
  - Language semantics are usually defined only for programs that have no static errors.
Checked runtime errors are violations that the language implementation is required to detect and report at runtime, in a clean way.

E.g. in Scala or Java: division by 0, array bounds violations, dereferencing a null pointer

Depending on language, might cause an error message + abort, or raise an exception (which in principle can be caught by program)

Language semantics must specify what runtime errors are checked and how
Unchecked Runtime Errors

Unchecked runtime errors are violations that the implementation does not have to detect.

Subsequent behavior of the computation is arbitrary (language semantics typically silent about this)

No “fail-stop” behavior: error might not be manifested until long after it occurs

E.g. in C: division by 0, array bounds violations, dereferencing a null pointer

Safe languages like Scala, Java, Python have no such errors!
Today: Binding, Scope, Storage

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

Scala `var` vs. `val`
Local Value Bindings

expr ::= num | expr + expr | ... | (expr) | id | let id = expr in expr

(let a = 8 + 5 in a * 3) + 3

binding

use

scope

Letₐ

Add

Mul

Varₐ

Add

Num₈

Num₅

Num₃
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 $(\text{let } a = 8 + 5 \text{ in } a \times 3)$

  but free in $a \times 3$

- We cannot evaluate a free variable
Parallel Scopes

\[
(\text{let } a = 8 + 5 \text{ in } a \times 3) + \\
(\text{let } b = 1 \text{ in } b + 2)
\]

What if both let’s bind \(a\)?
(let a = 8 + 5 in
  let b = a - 10 in
  let c = a * b in
  c + 2)

scope_a

scope_b

scope_a

scope_a&b
Shadowing

(\text{let } a = 8 + 5 \text{ in} \text{ let } a = a - 10 \text{ in} \text{ 36 } + a \text{ ) } + 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 x (+ x 3))
```

```
(@ f (* 13 3))
```
Function parameter scoping

(f x (+ x 3))

\text{scope}_x

\text{Fundef}_{f,x}

\begin{align*}
\text{Add} \\
\text{Var}_x & \quad \text{Num}_3
\end{align*}
Function Name Scoping

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

```ocaml
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.

- Another alternative is to distinguish declarations from definitions. E.g. in C:
  ```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 */
  ```

- Historically, this approach was taken so that compilers could process programs one function in a single forward pass (no longer a common requirement).

- A third alternative is to use explicit syntax to link mutually recursive definitions. E.g. in OCaml:
  ```ocaml
  let rec f(x:float) = g(x +. 1.0)
      and g(y:float) = f(y -. 1.0)
  ```

- Note that all these approaches to recursion break the “up and out” rule for finding bindings.

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

What should happen in the following program?

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

How about this one?

```Scheme
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!
“Static scope”/“Lexical scope”

Better if this program remains erroneous

```plaintext
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 and the Store

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

So evaluating a variable declaration involves two things:

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

2. creating a new **binding** from variable name to location

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
Storage Lifetimes

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

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

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

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.

```c
char *foo() {
    char *p = malloc(100);
    free(p);
    return p;
}
```