Fundamental Concepts of Name Analysis

- Program Entity: An identifiable entity, e. g. type, function, variable, label, module, package.
- Identifier: A class of tokens that are used to identify program entities; e. g. `minint`
- Name: A composite construct used to identify program entities, usually contains identifiers, e. g. `Thread.sleep`

- Establishing a Binding
  - Explicitly: A definition defines properties of the program entity. There are defining and applied occurrences of a name.
  - Implicitly: There are only applied occurrences of a name. Properties of the program entity may be defined by the context, e.g. variables in PHP

- Static Binding: A binding is established between a name and a program entity. It is valid in a certain area of the program text, the scope of the binding. There the name identifies the program entity. Outside of its scope the name is unbound or bound to a different entity. Scopes are expressed in terms of program constructs like blocks, modules, classes, etc..
- Dynamic Binding: Bindings are established at run-time, e. g. in Lisp.

Scope Rules

Scope rules: a set of rules that specify for a given language how bindings are established and where they hold.

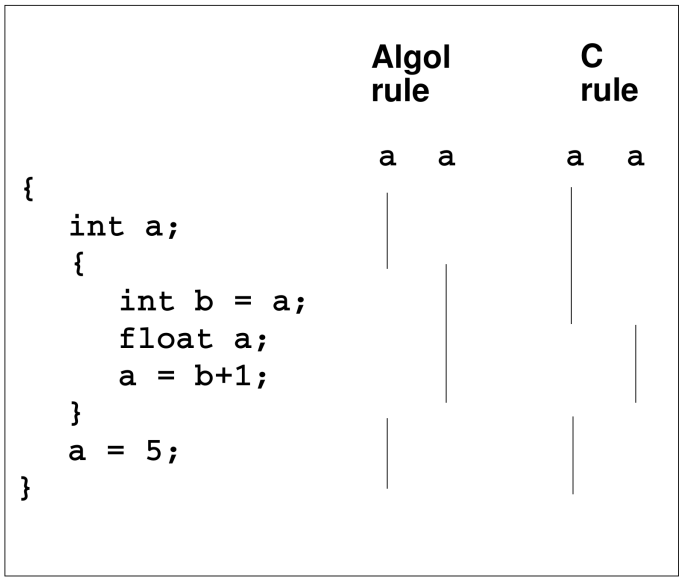
2 variants of fundamental **hiding rules** for languages with nested structures.
Both are based on **definitions that explicitly introduce bindings**:

Algol rule:
The definition of an identifier *b* is valid in the **whole smallest enclosing range**; but **not in inner ranges** that have a **definition of *b***, too.

e. g. in Algol 60, Pascal, Java

C rule:
The definition of an identifier b is valid in the **smallest enclosing range from the position of the definition** to the end; but **not in inner ranges** that have another **definition of b** from the position of that definition to the end.

e. g. in C, C++, Java



Defining Occurrence before Applied Occurrences

The C rule requires that the defining occurrence of a binding precedes all its applied occurrences. In Pascal, Modula and Ada the Algol rule holds. However, an additional rule requires that the defining occurrence of a binding precedes all its applied occurrences.

Consequences

- specific constructs for forward references of functions that call each other recursively:
  - forward function declaration in Pascal
  - function declaration in C before the function definition
- specific constructs for types which may contain references to each other recursively: forward type references allowed for pointer types in Pascal, C, Modula.
- specific rules for labels to allow forward jumps:
  - label declaration in Pascal before the label definition
  - Algol rule for labels in C
- Pascal requires declaration parts to be structured as a sequence of declarations for constants, types, variables and functions, such that the former may be used in the latter. Grouping by coherence criteria is not possible.

The Algol rule is simpler, more flexible and allows for individual ordering of definitions according to design criteria.

Multiple Definitions

Usually a definition of an identifier is required to be unique in each range. That rule guarantees that at most one binding holds for a given identifier in a given range.

Deviations from that rule:

- Definitions for the same binding are allowed to be repeated, e. g. in C:
  `external int maxElement;`
- Definitions for the same binding are allowed to accumulate properties of the program entity, e. g. in LIDO:
  `SYMBOL AppIdent: key: DefTableKey; ...`
  `SYMBOL AppIdent: type: DefTableKey;`
- Separate name spaces for bindings of different kinds of program entities. Occurrences of identifiers are syntactically distinguished and associated to a specific name space, e. g. different name spaces in Java for packages and classes:
  `import Stack.Stack;`
- Overloading of identifiers: different program entities are bound to one identifier with overlapping scopes. They are distinguished by static semantic information in the context, e. g. overloaded functions distinguished by the signature of the call (number and types of actual parameters).

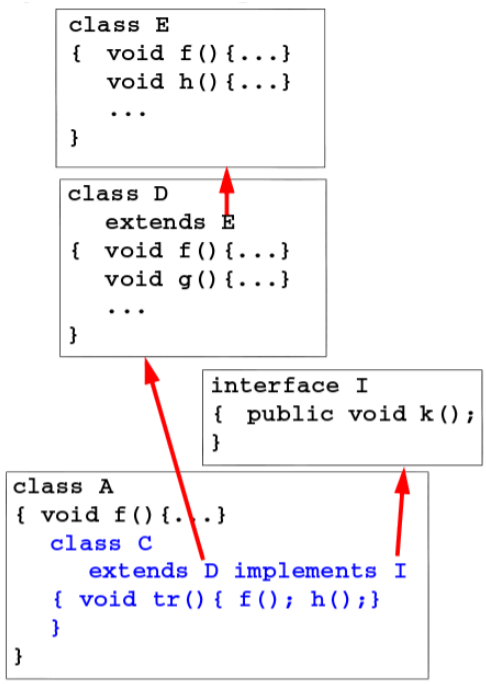
Inherited Bindings

A class provides a set of bindings that consists of the local bindings and those inherited from classes and interfaces. An inherited binding may be hidden by a local definition.

That set of bindings is used for identifying qualified names:

```
D d = new D();
d.f();
d.h();
d.g();
```

A class may be embedded in a context that provides bindings. In Java an unqualified name like `f()` is first tried to be bound in the local and inherited sets, and then in the bindings of the textual context.



An Environment Module for Name Analysis

Program entities are represented by keys. They reference descriptions of properties.

The task of name analysis:

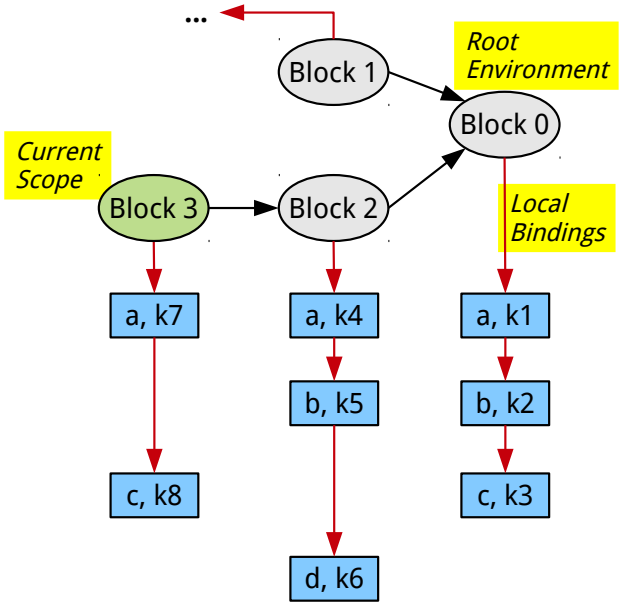
Associate the key of a program entity to each occurrence of an identifier according to the scope rules of the language.

The pair (identifier, key) represents a binding. Bindings that have a common scope are composed to sets.

An environment is a linear sequence of sets of bindings e_1, e_2, e_3, ... that are connected by a hiding relation: a binding (a, k1) in e_i hides a binding (a,k2) in e_j if $i < j$.

Name analysis is implemented using a module that implements environments and operations on them.

Environment Module: Basic Data Structure



- Tree structure for nested environments
- List of local bindings (Id, Key) in each environment
- Types
  - `Environment`
  - `Binding`
- Interface functions
  - Tree Construction: `NewEnv()` , `NewScope(Environment)`
  - Establish binding: `BindIdn(Environment, int)`
  - Lookup binding: `BindingInEnv(Environment, int)` yields `NoBinding` if lookup fails.

Looking up a binding requires linear search through local bindings starting in the current scope and continuing in enclosing environments until a binding is found or the search fails in the root environment

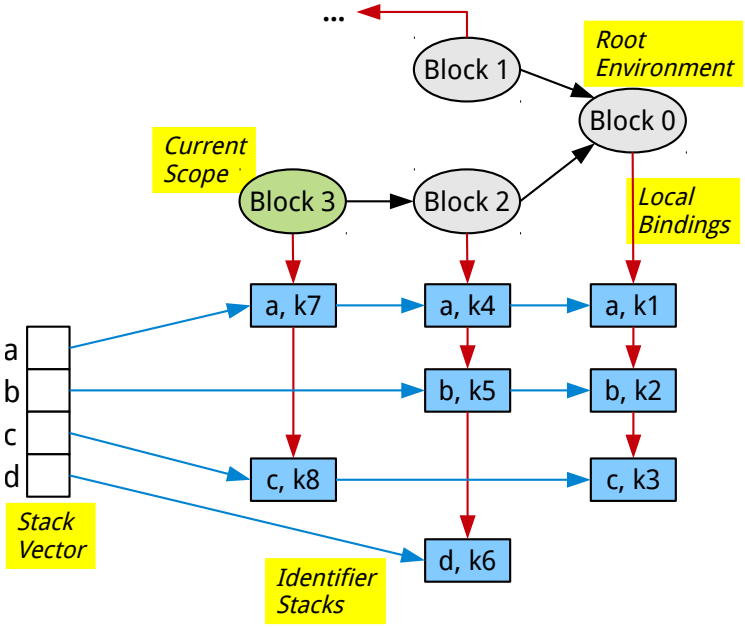
Environment Module: Efficient Data Structure

Using a stack of bindings for each identifier in the program.

- Local binding hides global ones
- No search, O(1) lookup.

Changing the current scope requires stack adjustment:

- moving towards the root: pop a set of bindings
- moving towards the leaves: push a set of bindings



Name Analysis using Attribute Grammars

Context	Symbol	Computation
Program	Root	`Root.Env = NewEnv();`
Nested Block	Range	`Range.Env = NewScope(INCLUDING (Range.Env, Root.Env));`
Defined Identifier Occurrence	IdDefScope	`IdDefScope.Bind = BindIdn(INCLUDING Range.Env, IdDefScope.Symb);`
Applied Identifier Occurrence	IdUseEnv	`IdUseEnv.Bind = BindingInEnv(INCLUDING Range.Env, IdUseEnv.Symb);`

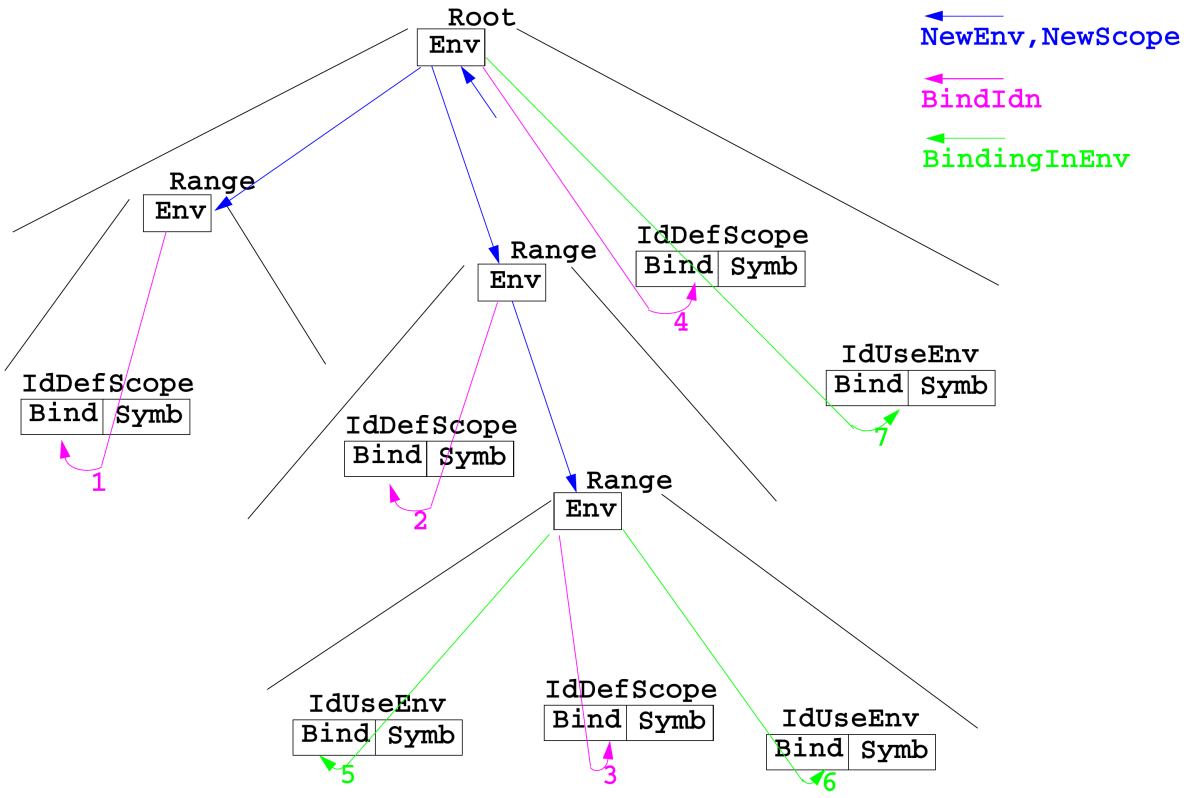
Preconditions for specific scope rules

- Algol rule: all `BindIdn()` of all surrounding ranges before any `BindingInEnv()`
- C rule: `BindIdn()` and `BindingInEnv()` in textual order

Typical semantic checks

- No applied occurrence without a valid defining occurrence
- At most one definition for an identifier in a range
- No applied occurrence before its defining occurrence (Pascal)

Attribute Computations for Name Analysis



Eli Module for Name Analysis

Library modules provides class symbols and computations for name analysis. These computational roles can be inherited by tree symbols:

Name Analysis for the Basic SetLan Version

```
~O~<SetLanName.specs~>~{~-
   $/Name/CScope.gnrc:inst     // we use C-like scope rules
~}

~O~<SetLanName.lido~>~{~-
   ATTR Sym: int;    // the name's symbol table index
   CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;

   SYMBOL Program    INHERITS RootScope END;
   SYMBOL VarNameDef INHERITS IdDefScope, IdentOcc END;
   SYMBOL VarNameUse INHERITS IdUseEnv, IdentOcc, ChkIdUse END;
~}
```

This is the complete specification (without checking for multiple definitions). It can easily be extended for nested scopes and additional identifier occurrences.