5. Binding of Names
5.1 Fundamental notions

Program entity: An **identifiable** entity that has **individual properties**, is used potentially at **several places in the program**. Depending on its **kind** it may have one or more runtime instances; e. g. type, function, variable, label, module, package.

Identifiers: a class of tokens that are used to **identify program entities; e. g.** `minint`

Name: a **composite construct** used to **identify a program entity**, usually contains an identifier; e. g. `Thread.sleep`

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, packets

Dynamic binding: Bindings are established in the **run-time** environment; e. g. in Lisp.

A **binding may be established**

- **explicitly by a definition**; it usually **defines properties** of the program entity; we then destinguish **defining and applied occurrences** of a name; e. g. in C: `float x = 3.1; y = 3*x;` or in JavaScript: `var x;`
- **implicitly by using the name**; properties of the program entity may be defined by the context; e. g. bindings of global and local variables in PHP

Lecture Programming Languages and Compilers WS 2011/12 / Slide 501

Objectives:

Repeat and understand notions

In the lecture:

Explanations and examples for

- program entities in contrast to program constructs,
- no, one or several run-time instances,
- bindings established explicitly and imlicitly

Suggested reading:

Kastens / Übersetzerbau, Section 6.2, 6.2.2

5.2 Scope rules

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

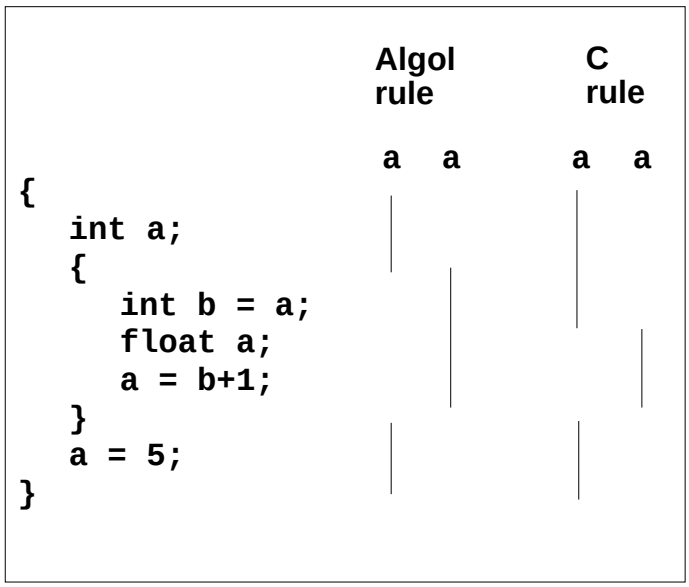
2 variants of fundamental **hiding rules** for languages with nested structures.
Both are based on **definitions that explicitly introduce bindings**:

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



Lecture Programming Languages and Compilers WS 2011/12 / Slide 502

Objectives:

Repeat fundamental hiding rules

In the lecture:

Explanations and examples for

- hiding rules (see "Grundlagen der Programmiersprachen"),
- occurrences of the Algol rule in Pascal (general), C (labels), Java (instance variables).

Suggested reading:

Kastens / Übersetzerbau, Section 6.2, 6.2.2

Defining occurrence before applied occurrences

The **C rule** enforces the defining occurrence of a binding precedes all its applied occurrences.

In Pascal, Modula, Ada the **Algol rule** holds. An **additional rule** requires that the defining occurrence of a binding precedes all its applied occurrences.

Consequences:

- specific constructs for **forward references of functions** which may call each other recursively:
  `forward` function declaration in Pascal;
  function declaration in C before the function definition,
  exemption form the def-before-use-rule in Modula
- 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
- (Standard) **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.

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 503

Objectives:

Understand consequences

In the lecture:

Explanations and examples for the mentioned consequences, constructs and rules.

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 (plain) 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. AG specification language LIDO: association of attributes to symbols:
  `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. in Java bindings of packets and types are in different name spaces:
  `import Stack.Stack;`
  in C labels, type tags and other bindings have their own name space each.
- **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).

Lecture Programming Languages and Compilers WS 2011/12 / Slide 504

Objectives:

Understand variants of multiple definitions

In the lecture:

Explanations and examples for

- the variants,
- their usefulness

Explicit Import and Export

Bindings may be **explicitly imported to or exported from a range** by specific language constructs. Such features have been introduced in languages like Modula-2 in order to support **modular decomposition and separate compilation**.

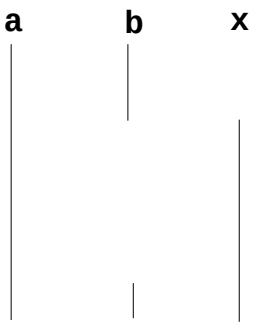
Modula-2 defines two different import/export features

1. **Separately compiled modules**:

```
DEFINITION MODULE Scanner;                    interface of a separately compiled module
   FROM Input IMPORT Read, EOL;                                  imported bindings
   EXPORT QUALIFIED Symbol, GetSym;                              exported bindings
   TYPE Symbol = ...;                                 definitions of exported bindings
   PROCEDURE GetSym;
END Scanner;
IMPLEMENTATION MODULE Scanner BEGIN ... END Scanner;
```

2. **Local modules, embedded in the block structure** establish scope boundaries:

```
VAR a, b: INTEGER;
...
MODULE m;
   IMPORT a;
   EXPORT x;
   VAR x: REAL;
BEGIN ... END m;
...
```



Lecture Programming Languages and Compilers WS 2011/12 / Slide 505

Objectives:

Understand explicit extension of scopes

In the lecture:

Explanations and examples for

- explicit import / export in contrast to implicit hiding,
- scopes related to interfaces,
- import of packets in Java.

Bindings as properties of entities

Program entities may have a property that is a set of bindings,
e. g. the entities exported by a module interface or the fields of a struct type in C:

```
typedef struct {int x, y;} Coord;

Coord anchor[5];
anchor[0].x = 42;
```

The type **Coord** has the bindings of its fields as its property; **anchor[0]** has the type **Coord**; **x** is bound in its set of bindings.

Language constructs like the **with**-statement of Pascal insert such sets of bindings into the bindings of nested blocks:

```
type Coord = record x, y: integer; end;
var  anchor: array [0..4] Coord;
     a, x: real;
begin ...
   with anchor[0] do
      begin ...
         x := 42;
      end;
   ...
end;
```

Bindings of the type **Coord** are inserted into the textually nested scopes; hence the field **x** hides the variable **x**.

Lecture Programming Languages and Compilers WS 2011/12 / Slide 506

Objectives:

Understand bindings as properties

In the lecture:

Explanations and examples for

- sets of bindings,
- used in qualified names,
- used in with-statements,
- name analysis depends on type analysis.

Inheritance with respect to binding

Inheritance is a **relation between object oriented classes**. It defines the basis for **dynamic binding of method calls**. However, **static binding rules** determine the **candidates for dynamic binding** of method calls.

A class has a **set of bindings as its property**.

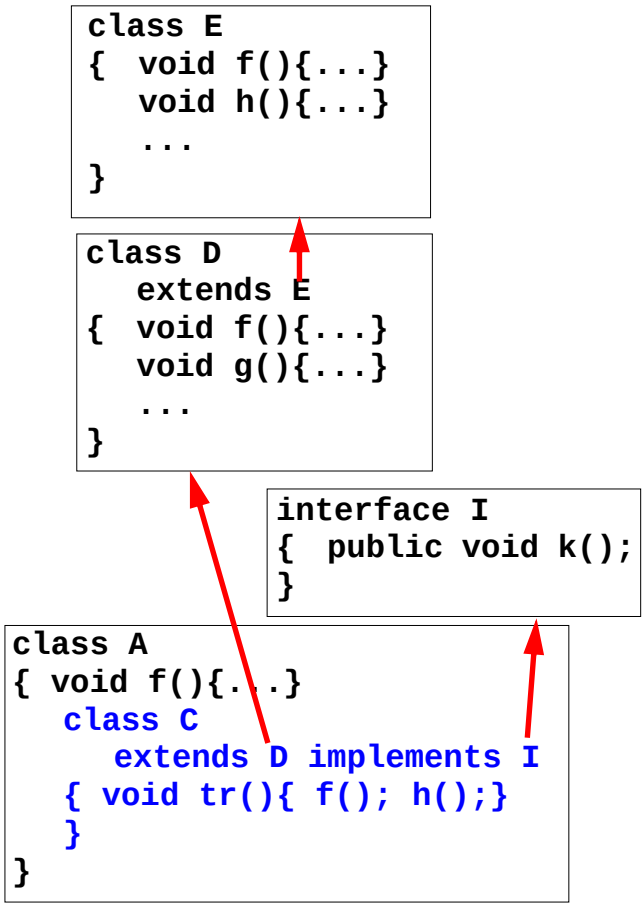
It consists of the bindings **defined in the class** and those **inherited** from classes and interfaces.

An **inherited binding may be hidden** by a local definition.

That set of bindings is used for identifying qualified names (cf. **`struct`** types):

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

A class may be **embedded in a context** that provides bindings. An unqualified name as in **`f()`** is bound in the **class's local and inherited** sets, and **then** in the **bindings of the textual context** (cf. **`with`**-statement).



Lecture Programming Languages and Compilers WS 2011/12 / Slide 507

Objectives:

Understand inheritance relation

In the lecture:

The example is used to explain

- inheritance hierarchy,
- hiding via inheritance,
- binding of qualified and unqualified names,
- nested classes,
- relation to dynamic method calls.

5.3 An environment module for name analysis

The compiler represents a **program entity by a key**. It references a description of the entity's properties.

Name analysis task: Associate the **key of a program entity to each occurrence of an identifier** according to **scope rules** of the language (consistent renaming).
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, k) in e_i hides a binding (a,h) in e_j if i < j.

Scope rules can be modeled using the concept of **environments**.

The **name analysis task** can be **implemented** using a **module** that implements **environments** and operations on them.

Lecture Programming Languages and Compilers WS 2011/12 / Slide 508

Objectives:

Understand the name analysis task

In the lecture:

Explanations and examples for

- environments,
- use of environments to model scope rules.

Environment module

Implements the abstract data type **Environment**:
hierarchically nested sets of **Binding**s **(identifier, environment, key)**
(The binding pair (i,k) is extended by the environment to which the binding belongs.)

Functions:

NewEnv ()	creates a new Environment e, to be used as root of a hierarchy
NewScope (e_1)	creates a new Environment e_2 that is nested in e1. Each binding of e_1 is also a binding of e_2 if it is not hidden there.
BindIdn (e, id)	introduces a binding (id, e, k) if e has no binding for id; then k is a new key representing a new entity; in any case the result is the binding triple (id, e, k)
BindingInEnv (e, id)	yields a binding triple (id, e_1, k) of e or a surrounding environment of e; yields NoBinding if no such binding exists.
BindingInScope (e, id)	yields a binding triple (id, e, k) of e, if contained directly in e, NoBinding otherwise.

Lecture Programming Languages and Compilers WS 2011/12 / Slide 509

Objectives:

Learn the interface of the Environment module

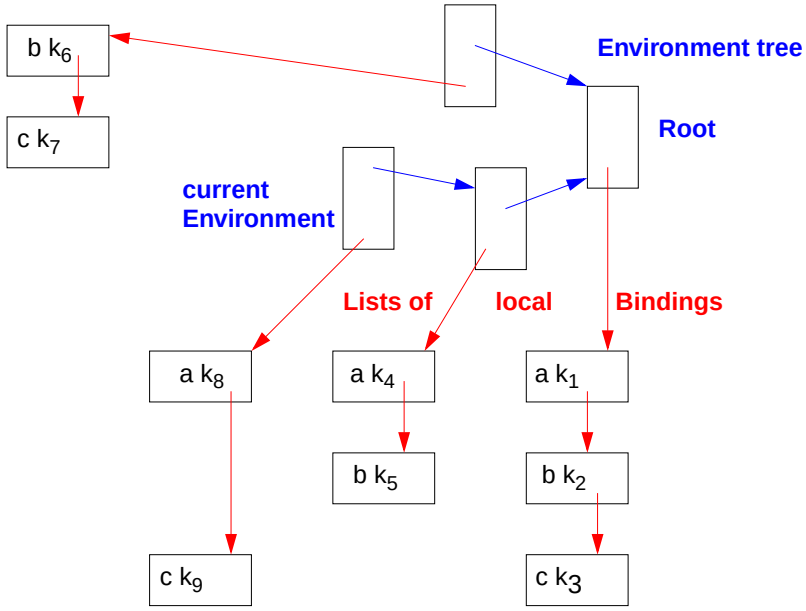
In the lecture:

- Explain the notion of Environment,
- explain the examples of scope rules,
- the module has further functions that allow to model inheritance, too.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Data structure of the environment module (1)



k_i: key of the defined entity

Lecture Programming Languages and Compilers WS 2011/12 / Slide 510

Objectives:

An search structure for definitions

In the lecture:

Explanations and examples for

- the environment tree,
- the binding lists.
- Each search has complexity O(n) in the number of definitions n.

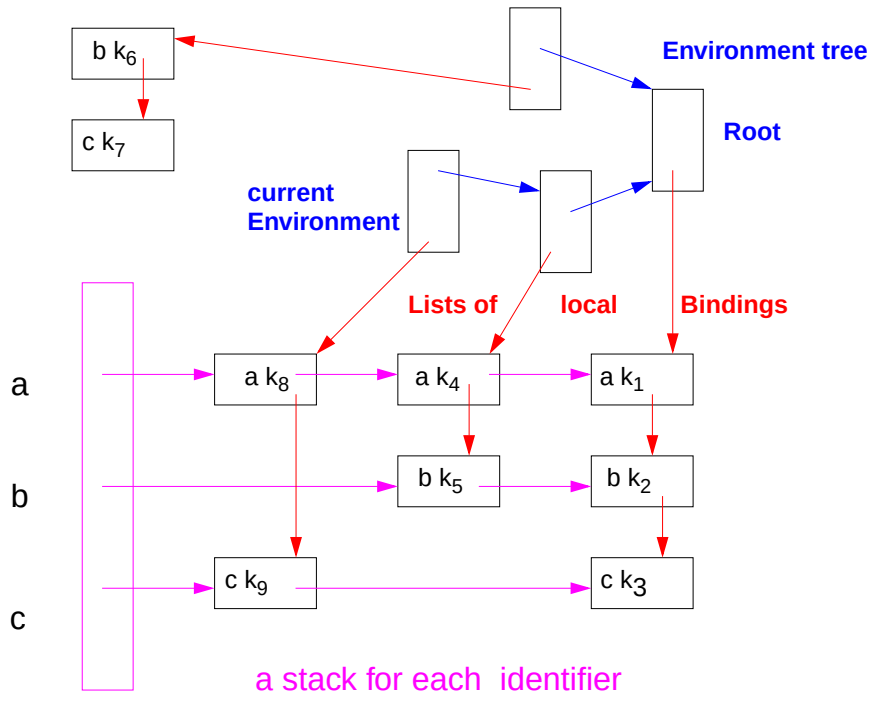
Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Questions:

- How is a binding for a particular identifier found in this structure?
- How is determined that there is no valid binding for a particular identifier and a particular environment.

Data structure of the environment module (2)



vector of stacks indexed by identifier codes

k_i: key of the defined entity

Lecture Programming Languages and Compilers WS 2011/12 / Slide 510a

Objectives:

An efficient search structure

In the lecture:

Explanations and examples for

- the concept of identifier stacks,
- the effect of the operations,
- O(1) access instaed of linear search,
- how the current environment is changed using operations Enter and Leave, which insert a set of bindings into the stacks or remove it.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Questions:

- In what sense is this data structure efficient?
- Describe a program for which a linear search in definition lists is more efficient than using this data structure.
- The efficiency advantage may be lost if the operations are executed in an unsuitable order. Explain!
- How can the current environment be changed without calling Enter and Leave explicitly?

Environment operations in tree contexts

Operations in tree contexts and the order they are called can **model scope rules**:

`Root` context:

```
Root.Env = NewEnv ();
```

Range context that may contain definitions:

```
Range.Env = NewScope (INCLUDING (Range.Env, Root.Env));
```

accesses the next enclosing Range or Root

defining occurrence of an identifier `IdDefScope`:

```
IdDefScope.Bind = BindIdn (INCLUDING Range.Env, IdDefScope.Symb);
```

applied occurrence of an identifier `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

The resulting **bindings are used for checks and transformations**, e. g.

- 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).

Lecture Programming Languages and Compilers WS 2011/12 / Slide 511

Objectives:

Apply environment module in the program tree

In the lecture:

- Explain the operations in tree contexts.
- Show the effects of the order of calls.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.1

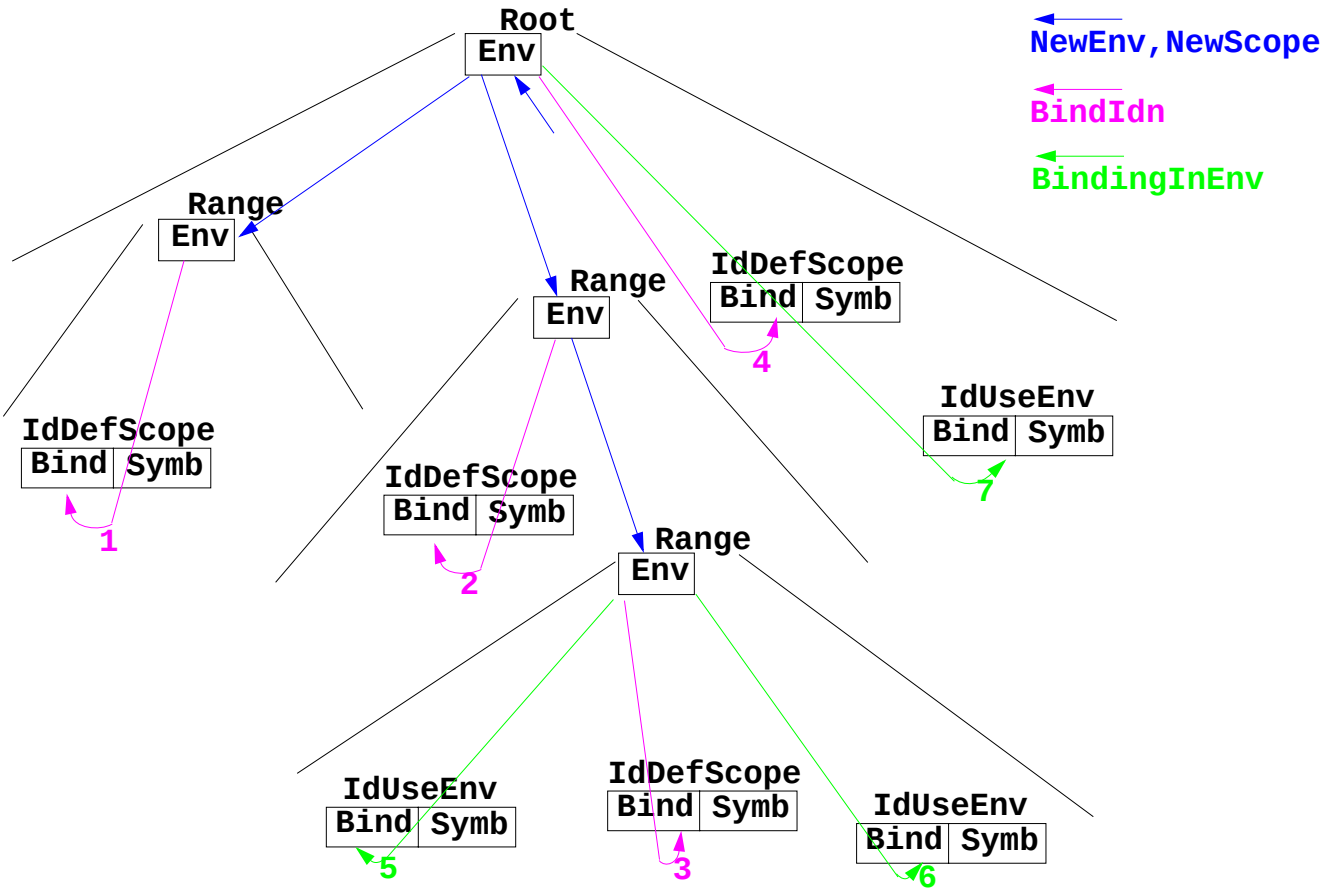
Assignments:

Use Eli module for a simple example.

Questions:

- How do you check the requirement "definition before application"?
- How do you introduce bindings for predefined entities?
- Assume a simple language where the whole program is the only range. There are no declarations, variables are implicitly declared by using their name. How do you use the operations of the environment module for that language?

Attribute computations for binding of names



Lecture Programming Languages and Compilers WS 2011/12 / Slide 512

Objectives:

Understand dependences for name analysis

In the lecture:

- Identify the computations of the environment structure (blue), insertion of bindings in environments (magenta), lookup of a binding in an environment (green);
- order for Algol rules: (4 before 7) and (2, 3, 4 before 5, 6)
- order for C rules: 1, 2, 5, 3, 6, 4, 7