Names, binding, scope, type checking

Contents

Names Variables

Points

- What needs a name?
- A variable is a six-tuple
- Aliasing

A name is a handle on an entity in a program. We refer to something by a name if we want to create, use, change, destroy it.

What needs a name?

- constants, variables,
- operators,
- labels,
- types,
- procedures, functions,
- modules, programs,
- files, disks,
- commands, menu items,
- computers, networks, user (login name).

Declaration, use, lifetime, scope of names: these are a major consideration in programming languages.

A name is denoted by an identifier.

☒ Example: identifiers in Ada.

```
‹identifier› ::=
 ‹letter›
 {[‹underline›](‹letter›|‹digit›)}
```

A discussion of forms of identifiers: see the textbook, Section 4.2.

Keywords or restricted words (e.g. `if`, `var`, `type`) provide the syntactic glue in a program. Consider:

`if` *C* `then` *S1* `else` *S2* `end if;`

This is really a triple: *C* - *S1* - *S2*. It might be equally well written as

`(`*C* `|` *S1* `|` *S2*`)` or `{`*C* `->` *S1* `;` *S2*`}`

so long as the compiler and the programmer both understood the deep meaning (that it's a triad).

Similarly,

`while` *C* `loop` *S* `end loop;`

is really a pair: *C* - *S*.

On the other hand, keywords and special words determine the style, the flavour of the language.

An aside remark: it is very easy to change keywords if it is a one-to-one change.

☒ It is a trivial modification of the lexical analyzer to get Pascal to recognize this:

`si` *C* `alors` *S1* `sinon` *S2* `fin si;`

A variable in an imperative language is a six-tuple:

‹name, address, value, type, lifetime, scope›

☒ For example, if we write

```
var x: integer;
```

we decide what will be the name and type of `x`. The place of this declaration in the program decides where and how long `x` is available (scope, lifetime). Its address is determined when its program unit is executing, Finally, using `x` in statements decides what is its current value.

A problem in most programming languages: the same name can be reused in different contexts and denote different entities.

☒ For example:

```
procedure a;
   var b: char;
begin {...} end;
procedure b;
   var a: integer;
begin {...} end;
```

Different types, addresses and values may be associated with such occurrences of a name. Each occurrence has a different lifetime (when and for how long is an object created?), and a different scope (where can the name be used?).

Aliasing is a situation when two names denote the same object (share an address): undesirable!

☒ An example:

```
var y : char;
{...}
procedure p( var x : char );
begin {...} x := y; {...} end;
{...}
begin {...} p( y ); {...} end;
```

Now, `x` and `y` both refer to the same address!

The concept of value is more general: l-value (l for left) and r-value (r for right) are the source and target of an assignment. An example:

```
x := y;
```

l-value is the address of `x`, r-value is the value of `y`. This becomes complicated for array elements that denote addresses which must be evaluated:

```
T[i*2+1] := y;
```

This address depends on the current value of `i`.

Binding

Points

- An informal presentation
- Binding times
- Bindings of a variable
- More on lifetime

Binding is not formally defined. Intuitively, we associate (that is, bind) an attribute with an object. Examples of attributes are name, type, value.

Binding occurs at various times in the life of a program. Three periods are usually considered:

compile time (actually, translation time, because binding happens both in compilers and interpreters);

load time (preparing object code for execution, pulling in the necessary code for built-in operations or those defined in libraries of modules);

run time (between starting the program's execution and its termination).

We also distinguish two kinds of binding, depending on its duration:

static binding is permanent during the life of the program;

dynamic binding is in force during some part of the program's life.

variable → name	compile time described in declarations
variable → address	load time or run time (e.g. Pascal), run time (e.g. Smalltalk) this is usually done implicitly
variable → type	compile time (e.g. Pascal), run time (e.g. Smalltalk) described in declarations
variable → value	run time, load time (initialization) specified in statements, mainly assignment
variable → lifetime	compile time described in declarations
variable → scope	compile time expressed by placement of declarations

More on lifetime: allocation of memory for an object happens at load time or at run time. Two classes of variables are distinguished:

Static variables

Allocation is done once, before the program starts.

Fortran was an important language with such an allocation policy for *all* objects. This is old-fashioned, inflexible, but it also is conceptually simple and inexpensive. Recursion is not possible.

Dynamic variables

Allocation is done after the program has started.

Two possibilities of dynamic allocation:

Explicit allocation and deallocation, that is, the programmer must do it. This is what we do when we use pointers; for example, in Pascal we allocate using new(p), deallocate using dispose(p).

Implicit allocation (when a block is entered) and deallocation (when a block is exited).

Scope

Points

- Blocks and block structure
- Anonymous blocks
- Nesting diagrams
- Call graphs
- Dynamic scoping

Blocks and block structure

We group declarations and statements in order to

- keep steps of a non-elementary activity together (e.g., all steps of a sorting algorithm),
- ensure proper interpretation of names.

Names are bound to various elements of a program. We refer to names in statements.

The **scope** of a name `N` means all places in the program where `N` denotes the same object.

Blocks can be **nested**. Names introduced in a block are called **local bindings**. A name referred to, but not declared, in a block must have been declared in a surrounding block.

Nesting of blocks is possible in Pascal, Ada and (in some form) in C. It is not allowed in Fortran.

A program, procedure or function consists of a heading and a *named* block. This is opposed to anonymous blocks, available in Algol 60, Algol 68, PL/I, Ada and C -- but not in Pascal.

Anonymous blocks

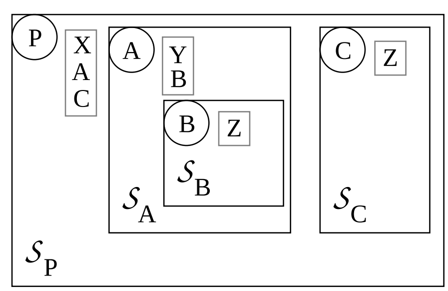
An anonymous block is like a procedure defined (without a name) and immediately called *once*.

Such blocks are useful when a computation is required once and auxiliary variables are only needed in this computation. We do not want to declare them outside the block. (A procedure might as well be used to achieve the same effect.)

☒ An example (written in Ada):

```
declare DELTA, SQRT_DELTA: real;
begin
 DELTA := B*B-4.0*A*C;
 if DELTA < 0.0 then
   NO_OF_ROOTS := 0;
 elsif DELTA = 0.0 then
   NO_OF_ROOTS := 1;
   ROOT1 := (-B)*0.5/A;
 else
   SQRT_DELTA := sqrt(DELTA);
   NO_OF_ROOTS := 2;
   ROOT1 := (-B-SQRT_DELTA)*0.5/A;
   ROOT2 := (-B+SQRT_DELTA)*0.5/A;
 end if;
end;
```


A diagram that shows nesting (and all locally defined names) in a program:

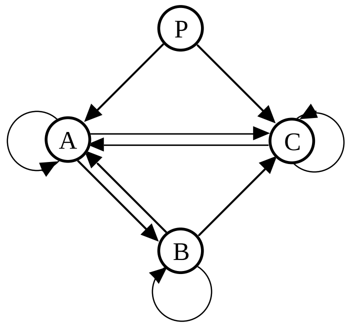


The same in Pascal (assume all the necessary `forward` declarations):

```
program P;
   var X: integer;
   procedure A;
      var Y: char;
      procedure B;
         var Z: Boolean;
      begin 𝒮B end;
   begin 𝒮A end;

   procedure C;
      var Z: integer;
   begin 𝒮C end;
begin 𝒮P end;
```

A call graph shows which program units can call other units (little loops show possible recursion).

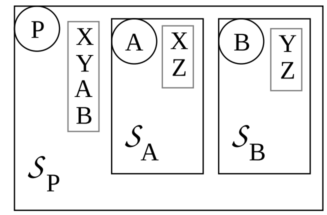


Visibility, or the Referencing Environment

P
A
B
$\mathcal{S}_B$ ⟵ P.X, P.A.Y, P.A.B.Z
$\mathcal{S}_A$ ⟵ P.X, P.A.Y
C
$\mathcal{S}_C$ ⟵ P.X, P.C.Z
$\mathcal{S}_P$ ⟵ P.X

Read P.X as "variable X declared in P".

Hole-in-scope (hiding): the same name is used in an enclosing block and a nested block.



The visible variables in $\mathcal{S}_P$: P.X, P.Y
The visible variables in $\mathcal{S}_A$: P.A.X, P.Y, P.A.Z
The visible variables in $\mathcal{S}_B$: P.X, P.B.Y, P.B.Z

That is, P.X is visible everywhere in P except in $\mathcal{S}_A$, where P.A.X hides it and itself becomes visible.

P.Y is visible everywhere in P except in $\mathcal{S}_B$, where P.B.Y hides it and itself becomes visible.

There is no hole in the scope of P.A.Z or P.B.Z because they have disjoint areas of visibility.

So far we saw static scoping (lexical scoping). It allows us to determine the use of every variable in a program **statically**, without executing it.

Dynamic scoping is an alternative with unclear advantages. The idea is to search for a name in a chain of *called* procedures, starting from the main program. This chain is built according to the visibility rules, but regardless of nesting.

⊠

```
program P;
   var X: integer;
   procedure A;
   begin
      X := X + 1;
      print(X);
   end;
   procedure B;
      var X: integer;
   begin
      X := 17;
      A;
   end;
begin
   X := 23;
   B;
end;
```


The main program calls B, then B calls A. When A refers to X, which X is it?

- With static scoping, it is X in P, the enclosing block of procedure A. The number printed will be 24.
- With dynamic scoping (search is done up the chain of calls), it is X in B, the most recently entered block with a declaration of X. The number printed will be 18.

The dynamic scoping rule has been used in APL, SNOBOL-4 and in classic Lisp. It is not used much today. Even Common Lisp, a recent standard, has adopted static scoping.

Dynamic scoping is easier to implement than static scoping, but:

- it is not possible to understand at compile time which variables refer to which objects, that is, type checking is impossible at compile time;
- internal variables are not protected in the way we have come to expect: in our example X is local in B but A can nevertheless access it.

Constants
•
Variable initialization

Points
- A few (and easily found in the textbook).

(Read Sections 4.11-4.12 of the textbook.)

Type checking

Points
- Operations and operands
- Strong typing
- Conversion and coercion

With few exceptions, operations in programming languages take operands (arguments) of well-defined, specific types.

Examples of operations

- Boolean (comparison, conjunction, disjunction etc.),
- arithmetic (+, -, *, /, sin, tan, exp etc.),
- string (concatenation, substring etc.),
- assignment (yes, this is an operation with two arguments),
- passing a parameter to a procedure.

Type checking ensures that an operator—when it is applied—gets arguments that it can handle. Type checking is done at compile time or at run time.

A type error occurs when an argument of an unexpected type is given to an operation. This too can be signalled at compile time or at run time.

A programming language has **strong typing** when all type errors can be discovered at compile time. This is not easy to achieve even in a strict language such as Pascal.

☒ Problem: subtypes of enumerated types.

```
type
   day = (mo,tu,we,th,fr,sa,su);
   workday = (mo,tu,we,th,fr);
var x : day; y : workday;
{...}
y := x; {???}
```

We also have records with variants (examples will be considered later in the course).

A relaxed definition of strong typing: all errors can be discovered, preferably at compile time.

It is interesting that no popular programming language features perfect strong typing. There always are small infractions. ML is perfect, but not very popular.

Strict typing of operands is elegant and desirable, but it may be impractical. What happens if two operands make some sense together?

☒ Example:

```
var
   x : real; n: integer;
{...} n := 2; {...}
x := n * 3.14;
```

This multiplication may be rejected by a strict compiler. We can use a conversion (also known as a type cast):

```
x := float(n) * 3.14;
```

We can also rely on an implicit conversion called coercion: automatically change the form of an operand with a slightly inappropriate type. For example, represent the value of n as a float and use this new (but equal) value in the multiplication.

C is an example of a language with exaggerated use of coercion.

Type compatibility

Points
- When can composite objects be compared?
- Compatibility by name and by structure

Two objects of different primitive types can only be compared if one type may be converted into the other—for example, integer into real.

When can composite objects of two different types be compared?

☒ In all these cases X and Y are comparable:

```
X, Y : T;
X : T;  Y : T;
X, Y : record F: real end record;
```

Here, however, X and Y cannot be compared:

```
X : record F: real end record;
Y : record F: real end record;
```

These are anonymous declarations (in Ada such declarations are consistently *elaborated*).

```
type τ1 is
   record F: real end record;
X : τ1;
type τ2 is
   record F: real end record;
Y : τ2;
```

Those were all examples of compatibility by name. There may also be compatibility by structure.

Two types are compatible by structure if their primitive components (or components of those components, or etc.) are pairwise the same.

☒ Consider this:

```
type PERSON is record
   AGE: integer;
   MARRIED: boolean;
end record;

type HOUSE is record
   NO_BEDROOMS: integer;
   GARAGE: boolean;
end record;
```

These types should not be compatible, because the programmer obviously meant for them to be different! We should not be able, for example, to assign `HOUSE`s to `PERSON`s.

Summary

....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....