Chapter 10 Scope 作用域

Reusing Names

- Scope is trivial if you have a unique name for everything:

```
fun square a = a * a;
fun double b = b + b;
```

- But in modern languages, we often use the same name over and over:

```
fun square n = n * n;
fun double n = n + n;
```

- How can this work?

变量名在同一个域中不能重复

Definitions

- When there are different variables with the same name, there are different possible bindings for that name
- Not just variables: type names, constant names, function names, etc.
- A definition is anything that establishes a possible binding for a name

Examples

```
fun square n = n * n;
fun square square = square * square;


const
  Low = 1;
  High = 10;
type
  Ints = array [Low..High] of Integer;
var
  X: Ints;
```

Scope

- There may be more than one definition for a given name
- Each occurrence of the name (other than a definition) has to be bound according to one of its definitions
- An occurrence of a name is *in the scope of* a given definition of that name whenever that definition governs the binding for that occurrence

5

Examples

```
- fun square square = square * square;
val square = fn : int -> int
- square 3;
val it = 9 : int
```

- Each occurrence must be bound using one of the definitions
- Which one?
- There are many different ways to solve this scoping problem

6

Blocks

- A block is any language construct that contains definitions, and also contains the region of the program where those definitions apply

```
let
  val x = 1;
  val y = 2;
in
  x+y
end
```

7

Different ML Blocks

- The **`let`** is just a block: no other purpose
- A **`fun`** definition includes a block:

```
fun cube x = x*x*x;
```

- Multiple alternatives have multiple blocks:

```
fun f (a::b::_) = a+b
|   f [a] = a
|   f [] = 0;
```

- Each rule in a match is a block:

```
case x of (a,0) => a | (_,b) => b
```

8

Java Blocks

- In Java and other C-like languages, you can combine statements into one *compound statement* using **{** and **}**
- A compound statement also serves as a block:

```
while (i < 0) {
  int c = i*i*i;
  p += c;
  q += c;
  i -= step;
}
```

9

Nesting 嵌套

- What happens if a block contains another block, and both have definitions of the same name?
- ML example: what is the value of this expression:

```
let
  val n = 1
in
  let
    val n = 2
  in
    n
  end
end
```

10

Classic Block Scope Rule

- The scope of a definition is the block containing that definition, from the point of definition to the end of the block, minus the scopes of any redefinitions of the same name in interior blocks
- That is ML's rule; most statically scoped, block-structured languages use this or some minor variation

11

Example

Scope of this definition is (A – B)

```
let
  val n = 1
in
  let
    val n = 2
  in
    n
  end
end
```

A

B

Scope of this definition is B

12

Labeled Namespaces 命名空间

- A labeled namespace is any language construct that contains definitions and a region of the program where those definitions apply, and also has a name that can be used to access those definitions from outside the construct
- ML has one called a *structure*…

13

ML Structures

```
structure Fred = struct
  val a = 1;
  fun f x = x + a;
end;
```

- A little like a block: **a** can be used anywhere from definition to the end
- But the definitions are also available outside, using the structure name: **Fred.a** and **Fred.f**

内部可任意使用 a、

可在外部使用 Fred.a
Fred.f

14

Other Labeled Namespaces

- Namespaces that are just namespaces:
  - C++ **`namespace`**
  - Modula-3 **`module`**
  - Ada **`package`**
  - Java **`package`**
- Namespaces that serve other purposes too:
  - Class definitions in class-based object-oriented languages

15

Example

```
public class Month {
  public static int min = 1;
  public static int max = 12;
  …
}
```

- The variables **`min`** and **`max`** would be visible within the rest of the class
- Also accessible from outside, as **`Month.min`** and **`Month.max`**
- Classes serve a different purpose too

16

Namespace Advantages

- Two conflicting goals:
  - Use memorable, simple names like **`max`**
  - For globally accessible things, use uncommon names like **`maxSupplierBid`**, names that will not conflict with other parts of the program
- With namespaces, you can accomplish both:
  - Within the namespace, you can use **`max`**
  - From outside, **`SupplierBid.max`**

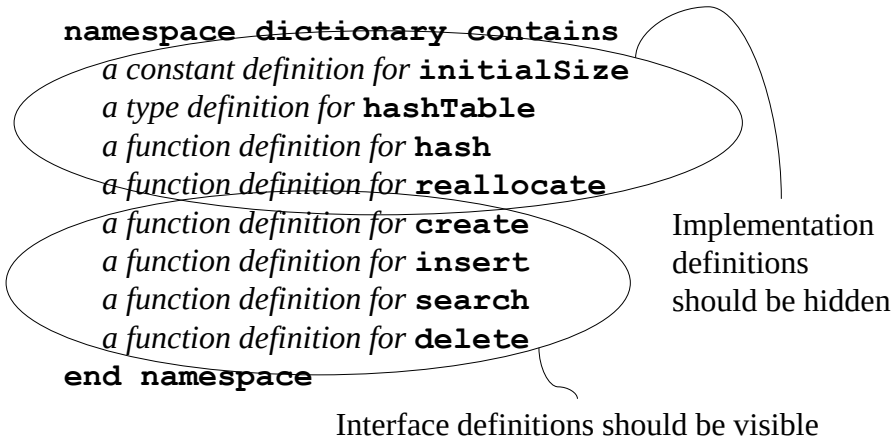
17

Namespace Refinement

- Most namespace constructs have some way to allow part of the namespace to be kept private
- Often a good *information hiding* technique
- Programs are more maintainable when scopes are small
- For example, *abstract data types* reveal a strict interface while hiding implementation details…

18

Example: An Abstract Data Type



Two Approaches

- In some languages, like C++, the namespace specifies the visibility of its components
- In other languages, like ML, a separate construct defines the interface to a namespace (a *signature* in ML)
- And some languages, like Ada and Java, combine the two approaches

Namespace Specifies Visibility

namespace dictionary contains
  private:
  a constant definition for **initialSize**
  a type definition for **hashTable**
  a function definition for **hash**
  a function definition for **reallocate**
  public:
  a function definition for **create**
  a function definition for **insert**
  a function definition for **search**
  a function definition for **delete**
end namespace

21

Separate Interface

interface dictionary contains
  a function type definition for **create**
  a function type definition for **insert**
  a function type definition for **search**
  a function type definition for **delete**
end interface

namespace myDictionary implements dictionary contains
  a constant definition for **initialSize**
  a type definition for **hashTable**
  a function definition for **hash**
  a function definition for **reallocate**
  a function definition for **create**
  a function definition for **insert**
  a function definition for **search**
  a function definition for **delete**
end namespace

22

Scoping with Primitive Namespaces

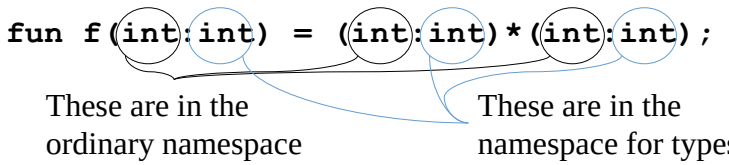
```
- val int = 3;
val int = 3 : int
```

- It is legal to have a variable named **`int`**
- ML is not confused
- You can even do this (ML understands that **`int*int`** is not a type here):

```
- fun f int = int*int;
val f = fn : int -> int
- f 3;
val it = 9 : int
```


Primitive Namespaces

- ML's syntax keeps types and expressions separated
- ML always knows whether it is looking for a type or for something else
- There is a separate namespace for types



Primitive Namespaces

- Not explicitly created using the language (like primitive types)
- They are part of the language definition
- Some languages have several separate primitive namespaces
- Java: packages, types, methods, variables, and statement labels are in separate namespaces

25

When Is Scoping Resolved?

- All scoping tools we have seen so far are static
- They answer the question (whether a given occurrence of a name is in the scope of a given definition) at compile time
- Some languages postpone the decision until runtime: *dynamic scoping*

26

Dynamic Scoping

- Each function has an environment of definitions
- If a name that occurs in a function is not found in its environment, its *caller's* environment is searched
- And if not found there, the search continues back through the chain of callers
- This generates a rather odd scope rule…

每个函数都有一个定义环境.

如果该环境中找不到，就到上级环境中去找.

主动调用

27

Classic Dynamic Scope Rule

- The scope of a definition is the function containing that definition, from the point of definition to the end of the function, along with any functions when they are called (even indirectly) from within that scope—minus the scopes of any redefinitions of the same name in those called functions

28

Static Vs. Dynamic

- The scope rules are similar
- Both talk about *scope holes*—places where a scope does not reach because of redefinitions
- But the static rule talks only about regions of program text, so it can be applied at compile time
- The dynamic rule talks about runtime events: "functions when they are called…"

Example

```
fun g x =
  let
    val inc = 1;
    fun f y = y+inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
in
  h x
end;
```

What is the value of `g 5` using ML's classic block scope rule?

Block Scope (Static)

```
fun g x =
  let
    val inc = 1;
    fun f y = y+inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```

With block scope, the reference to **inc** is bound to the previous definition in the same block. The definition in **f**'s caller's environment is inaccessible.

g 5 = 6 in ML

31

Dynamic Scope

```
fun g x =
  let
    val inc = 1;
    fun f y = y+inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```

With dynamic scope, the reference to **inc** is bound to the definition in the caller's environment.

g 5 = 7 if ML used dynamic scope

32

Where It Arises

- Only in a few languages: some dialects of Lisp and APL
- Available as an option in Common Lisp
- Drawbacks:
  - Difficult to implement efficiently
  - Creates large and complicated scopes, since scopes extend into called functions
  - Choice of variable name in caller can affect behavior of called function

33

Questions?

34