Nomi e ambiente

Blocchi e regole di scope

Nomi

Meccanismi di astrazione

- astrarre dalla macchina fisica
- nascondere dettagli, mettere in evidenza le parti importanti

fondamentali per gestire la complessità del software

L'uso dei nomi: un meccanismo di astrazione base
già presente in assembly

Nome: sequenza di caratteri usata per denotare qualcos'altro.

Permettono una rappresentazione sintetica, astratta, mnemonica di:

- valori (costanti)
- locazioni di memoria (variabili)
- pezzi di codice (procedure)
- operatori (funzioni base)
- ...

Sintassi

I nomi possono essere sequenze di caratteri significativi,

non solo identificatori, ma anche simboli semplici

```
+   -   <=   ++
```

Esempio

```
const pi = 3.14;
int x = pi + 1;
void f(){...};
```

Nomi e oggetti denotabili

- Bisogna distinguere tra:
  - nome (stringa di caratteri),
  - l'oggetto rappresentato, denotato.
- In linguistica: significante e significato
- Oggetto denotabile: oggetto associabile a un nome,
  - linguaggi programmazione diversi hanno oggetti denotabili diversi

Legame, ambiente.

- Legame (binding): associazione esistente tra nome e oggetto
- Ambiente (environment): insieme dei legami esistenti
  dipende da:
  - uno specifico punto del programma

  può dipendere:
  - dal codice eseguito in precedenza, storia del programma.

Creazione del binding

Possiamo separare tra:

- nomi definiti dal linguaggio
  - tipi primitivi, operazioni primitive, costanti predefinite;
- nomi definiti dal programmatore,
  - variabili, parametri formali, procedure (in senso lato), tipi definiti dall'utente, etichette, moduli, costanti definite dall'utente, eccezioni.

Creazione del binding

Il binding può essere creato in vari momenti:

- definizione del linguaggio,
- scrittura del codice,
- caricamento del programma in memoria,
- esecuzione,
- ...

Ma distinguiamo principalmente tra:

- binding statico prima dell'esecuzione della prima istruzione
- binding dinamico durante l'esecuzione

Ma anche l'oggetto denotato viene incrementalmente definito in vari momenti,

- es. dichiarazione di una variabile

```
int a
int myArray[size];
```

Ambiente e store:

- Il valore di una variabile x dipende da due funzioni:
  - ambiente: definisce quale locazione di memoria contiene il dato di x
  - store (memoria): determina il dato effettivo.
- Accesso al valore in due passaggi.
- I comandi possono modificare lo store (assegnazione), ma non l'ambiente.
- Ambiente modificabile attraverso dichiarazioni.
- Alcuni linguaggi non prevedono l'esistenza di uno store (funzionali puri).

Dichiarazione:

Nomi e legami sono quasi sempre definiti attraverso

Dichiarazioni: meccanismo (implicito o esplicito) col quale si crea un legame (si modifica l'ambiente)

```
int x = 0;
typedef int T;
int inc (T x) {
    return x + 1;
}
```

Attraverso più dichiarazioni,
lo stesso nome può denotare oggetti distinti
in punti diversi del programma

- Python, e altri linguaggi tipati dinamicamente, un'eccezione: assegno una variabile senza dichiararla, dichiarazione implicita, declared on first use

Blocchi

Nei linguaggi moderni l'ambiente è strutturato

- Blocco: regione del programma che può contenere dichiarazioni locali a quella regione

`{...}`	`C, Java`
`begin ... end`	`Algol, Pascal`
`(...)`	`Lisp`
`let...in...end`	`Scheme, ML`

Possono essere:

- associati a una procedura
- anonimi (o in-line)

Vantaggi dei blocchi

Gestione locale dei nomi:

```
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

- definire nomi locali indipendenti da altre dichiarazioni.
- strutturare il programma,

Con un'opportuna allocazione della memoria (vedi dopo):

- ottimizzano l'occupazione di memoria
- permettono la ricorsione

Annidamento

I blocchi possono essere annidati:

```
{
    int x = 0;
    int y = 2;
    {
        int x = 1;
        x = y;
    }
    write(x);
}
```

Regola di visibilità:

- una dichiarazione è visibile nel blocco di definizione
  e in tutti quelli annidati
  - a meno di mascheramento: una nuova dichiarazione per lo stesso nome nasconde, maschera, la precedente

Esempio

```
A:{
    int a = 1;
    B:{
        int b = 2;
        int c = 3;
        C:{
            int c=4;
            write(a+b+c);
        }
        D:{
            write(a+b+c);
        }
    }
}
```

Utilizzabilità di una dichiarazione

Dov'è utilizzabile una dichiarazione, all'interno del blocco in cui essa compare?

- tipicamente a partire dalla dichiarazione e fino alla fine del blocco non posso usare una dichiarazione prima di averla scritta
- in alcuni linguaggi (Modula3, Python, Haskell) in tutto il blocco forward reference

Dichiarazioni come:

```
{const a = b;
 const b = 3;
...
}
```

possono generare errore oppure no (con la forward reference)

Validità di una dichiarazione,

- a partire dalla dichiarazione
- in tutto il blocco

Istruzioni del tipo:

```
const a = 1;
...
procedure foo;
    const b = a;
    const a = 2
```

- generano errore in Pascal, C#, dove validità (tutto il blocco) e utilizzabilità (dalla dichiarazione) non coincidono
- tipicamene a `b` viene assegnato `1` (C, Java)
- può essergli assegnato `2` (Modula3, Python)

Suddividiamo l'ambiente

L'ambiente (in uno specifico blocco) può essere suddiviso in

- ambiente locale: associazioni create all'ingresso nel blocco
  - variabili locali
  - parametri formali
- ambiente globale: relativo al programma principale
  - dichiarazioni esplicite di variabili globali
  - associazioni esportate da moduli ecc.
- ambiente non-locale, non-globale:
  - associazioni ereditate da altri blocchi

Operazioni sull'ambiente

Tipicamente avvengono all'ingresso, uscita dei blocchi

- Creazione associazione nome-oggetto denotato (naming)
  - entrata in un blocco, dichiarazione locale in blocco
- Distruzione associazione nome-oggetto denotato (unnaming)
  - uscita da blocco con dichiarazione locale
- Riferimento oggetto denotato mediante il suo nome (referencing)
  - uso di un nome, nel codice,
- Disattivazione associazione nome-oggetto denotato
  - entrata in un blocco con dichiarazione che maschera
- Riattivazione associazione nome-oggetto denotato
  - uscita da blocco con dichiarazione che maschera

Operazioni sugli oggetti denotabili

- Creazione
- Accesso
- Modifica (se l'oggetto è modificabile)
- Distruzione

Creazione e distruzione di un oggetto non coincidono con creazione e distruzione dei legami per esso

Alcuni oggetti denotabili (come: costanti, tipi) non vengono né creati né distrutti.

- creazione, distruzione fanno riferimento a dati in memoria (variabili), codice (procedure)

Eventi base in un binding

Una ricapitolazione dei punti precedenti:

1. Creazione di un oggetto
2. Creazione di un legame per l'oggetto
3. Riferimento all'oggetto, tramite il legame (per accesso, modifica)
4. Disattivazione di un legame
5. Riattivazione di un legame
6. Distruzione di un legame
7. Distruzione di un oggetto

- 1-7 tempo di vita dell'oggetto
- 2-6 tempo di vita dell'associazione

Tempo di vita

La vita di un oggetto non necessariamente coincide con la vita dei legami per quell'oggetto

Vita dell'oggetto più lunga di quella del legame:

- passaggio per riferimento, di una variabile in una procedura

```
var A:integer;
procedure P (ref X:integer);
   begin ...
   end;
...
P(A);
```

L'esecuzione di P crea un legame tra X e un oggetto esistente prima e dopo l'esecuzione.

Tempo di vita

- Stesso esempio in C

```
int A;
void P(int *X){
   ...
}
...
P(&A);
```

- Un ulteriore esempio di oggetto che sopravvive al legame:
  - mascheramento
    in questo caso il legame si disattiva ma non sparisce completamente.

Tempo di vita

Vita dell'oggetto più breve di quella del legame:

- puntatore ad area di memoria dinamica deallocata.

```
int *X, *Y, z;
X = (int *) malloc (sizeof (int));
Y = X;
*Y = 5;
free (X);
z = *Y;
```

- Dangling reference: riferimenti pendenti, causa di errori

Regole di scope

In presenza di procedure
le regole di scope diventano più complesse

```
int x=10;
void incx () {
    x=x+1;
}
void foo () {
    int x = 0;
    incx();
}
int main () {
foo();
write(x);
}
```

- quale `x` incrementa `incx`?
  - si stampa 10 o 11?

Due alternative

Un nome non-locale alla procedura `incx` fa riferimento a:

- alla prima dichiarazione in blocco che include sintatticamente `incx`
- all'ultima dichiarazione "eseguita" prima di `incx`

Scope statico

Prima alternativa:

- un nome non locale è determinato dai blocchi che testualmente lo racchiudono a partire da quelli più interni:

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{   int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Scope statico

Prima alternativa:

- un nome non locale è determinato dai blocchi che testualmente lo racchiudono a partire da quelli più interni:

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{   int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Stampa: 2 0 5

Scope dinamico

Seconda alternativa

- un nome non locale è determinato dalla sequenza di blocchi attivi, a partire da quelli attivati più recentemente

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{   int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Scope dinamico

Seconda alternativa

- un nome non locale è determinato dalla sequenza di blocchi attivi, a partire da quelli attivati più recentemente

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{   int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Output: 2 3 2

Scope statico: indipendenza dalla posizione

```
int x=10;
void incx () {
   x++;     }
void foo (){
   int x = 0;
   incx();  }
foo();
```

- `incx` è dichiarato nello scope della `x` più esterna,
- `incx` è chiamato nello scope della `x` più interna,
- `incx` può essere chiamata in molti contesti diversi,
- l'unico modo per `incx` di comportarsi in modo uniforme è che il riferimento a `x` nelle due chiamate di incx sia sempre quello più esterno (scope statico).

Scope statico: indipendenza dai nomi locali

```
int x=10;
void incx () {
   x++;    }
void foo (){
   int y =0;
   incx();   }
foo();
```

In questa seconda versione `foo` usa una variabile locale diversa

- `y` invece di `x`

Questa modifica di nomi di variabili locali:

- modifica il comportamento del programma con lo scope dinamico
- non ha alcun effetto in scope statico.

Scope dinamico: specializzare una funzione

Supponiamo che “visualizza” sia una procedura che

- rende a colori sul video un certo testo
- usa come parametro una costante non-locale “colore”

Con scope dinamico posso modificare il suo comportamento con:

```
{
const colore = rosso;
visualizza(testo);
}
```

Tutte le identificatori non locali diventano dei parametri impliciti della procedura:

- maggiore flessibilità nell’uso delle procedure
  - simulo il meccanismo dei “valori di default”
- maggiore difficoltà nel determinare il comportamento della procedura

Scope statico vs dinamico

Scope statico (statically scoped, lexical scoping):

- informazione completa dal testo del programma
- le associazioni sono note a tempo di compilazione
- principi di indipendenza
- più complesso da implementare ma più efficiente
- Algol, Pascal, C, Java, Scheme, ...
- Usato in praticamente tutti i linguaggi recenti

Scope dinamico (dynamically scoped):

- informazione derivata dall'esecuzione
- spesso causa di programmi meno "leggibili' '
- più flessibile: modificare il comportamento di una procedura al volo
- più semplice da implementare, ma meno efficiente
  - esistono implementazioni efficienti ma piuttosto complesse
- R, Ruby (come opzione), Lisp (alcune versioni), Perl, APL, Snobol, ...

Scope statico ma
permette di usare una variabile senza prima dichiararla

- forward reference

```
>>> def f():
...     write(x)
...
>>> x = "global"
>>> f()
global
```

Python

- l'assegnazione di una variabile comporta la sua dichiarazione

```
def f():
    write(x)
def g():
    x = "local"
    f()
x = "global"
g()
```

- stampa

  global

Aliasing

Nomi diversi denotano lo stesso oggetto, causato da:

- passaggio dei parametri a una procedura per riferimento:

```
int x = 2;
int foo (ref int y){
    y = y + 1;
    x = x + y;}
foo(x);
write (x);
```

- puntatori:

```
int *X, *Y;
X = (int *) malloc (sizeof (int));
*X = 0;
Y = X;
*Y = 1;
write (*X);
```

Overloading

Lo stesso nome può avere significati diversi a seconda del contesto (accade spesso nei linguaggi naturali)

Normalmente, per i nomi di alcune funzioni predefinite:

- somma '+' è un classico esempio:
  - somma tra interi
  - somma floating-point
  - concatenazione tra stringhe

Il contesto, tipo degli argomenti, determinano il significato corretto.

Determinare l'ambiente

L'ambiente è dunque determinato da:

- regola di scope (statico o dinamico)
- regole specifiche, per esempio:
  - quando è visibile una dichiarazione nel blocco in cui compare?

Più avanti discuteremo:

- regole per il passaggio dei parametri
- regole di binding (shallow o deep)
  - intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X

Costrutto let in Scheme

Scheme permette dichiarazioni attraverso diversi costrutti:

- `let`
- `let*`
- `letrec`

con diversa validità delle dichiarazioni.

- `let`: non ricorsiva, creazione in blocco del nuovo ambiente,
- `let*`: non ricorsiva, creazione sequenziale di una serie di ambienti,
- `letrec`: ricorsive e mutuamente ricorsive.

Esempio

- Qual è la valutazione di:

```
(let ((a 1))
  (let ((a 2)
        (b a))
    b))
```

- e di:

```
(let ((a 1))
  (let* ((a 2)
         (b a))
    b))
```

Esempio

- Qual è la valutazione di:

```
(let ((a 1))
  (let ((a 2)
        (b a))
    b))
```

- e di:

```
(let ((a 1))
  (let* ((a 2)
         (b a))
    b))
```

rispettivamente 1 e 2

Mutua ricorsione (di funzioni o tipi)

forza l'utilizzo di un nome prima che questo venga dichiarato,

per essere possibile, devo permettere eccezioni al vincolo:

- un nome deve essere dichiarato prima di essere usato (utilizzabilità delle dichiarzioni)

Java: dichiarazione di metodi

```
{void f(){
    ...
    g(); // g non ancora dichiarato
    ...
}
void g(){
     ...
     f();
     ...
}
```

Mutua ricorsione di definizione di tipo

Pascal per tipi puntatore:

```
type lista = ^elem;
     elem = record
              info : integer;
              next : lista;
            end
```

posso definire un tipo puntatore prima di aver definito il tipo puntato

In C:

```
struct child {
  struct parent *pParent;
};
struct parent {
  struct child *children[2];};
```

posso definire `struct` con campi puntatore a tipi non ancora definiti

Dichiarazioni incomplete di tipo:

- Forward declaration, in C:

```
typedef struct elem element;
struct elem {
    int info;
    element *next;
}
```

Dichiaro `element` un tipo `struct` non ancora definito

```
typedef struct child ch;
struct child {
  struct parent *pParent;
};
struct parent {
  ch *children[2];
};
```

Analogamente per `ch`,

Dichiarazioni incomplete di tipo:

In Ada

```
type elem;
type lista is access elem;
type elem is record
              info: integer;
              next: lista;
     end
```

Dichiarazioni incomplete di funzione

In C:

```
void eval_parent(struct parent p);  // solo dichiarazione

void eval_child(ch c){
    ...
    eval_parent(p2);
}
void eval_parent(struct parent p){  // definizione completa
    ...
    eval_child(c2);
}
```

La prima dichiarazione incompleta di eval_parent
serve in eval_child a sapere come usare eval_parent:
che tipo di parametri passare, che risultato restituisce

Moduli e information hiding

- Moduli: ulteriore meccanismo per definire l'ambiente, gestire i binding.
- Programmi di grosse dimensioni pongono il problema di nascondere parte dei nomi.
  - usare per caso lo stesso nome in due parti del codice porta a comportamenti imprevedibili
  - evitare conflitti tra nomi porta a un sovraccarico cognitivo

I blocchi annidati (e procedure) risolvono il problema ma non completamente.