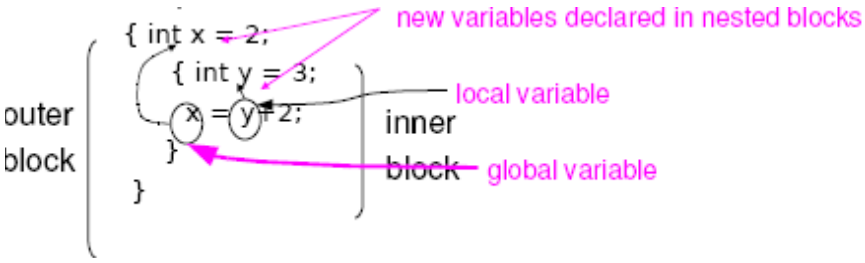
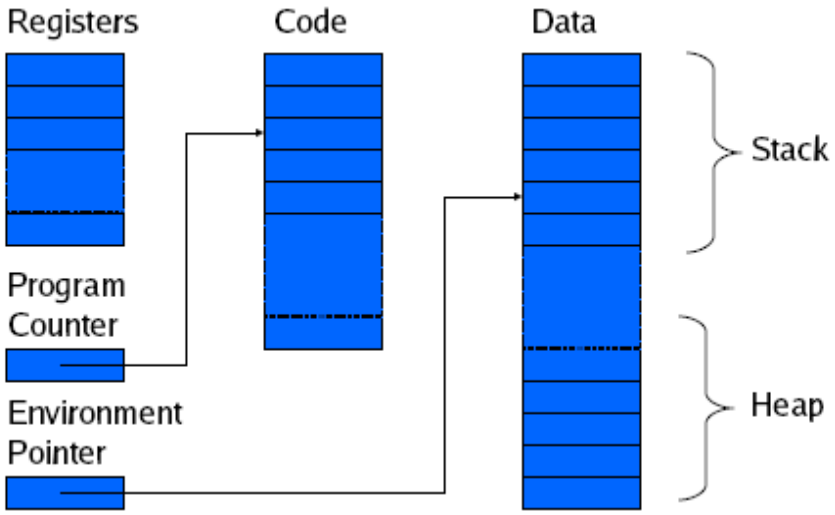
Scope, Function Calls and Storage Management



Quando si entra in un nuovo blocco ad esempio dal blocco esterno a quello interno viene allocato dello spazio sullo stack per gestire la memoria. In caso di uscita dal blocco (interno verso esterno) il blocco di memoria creato viene deallocato. La gestione poco accorta di questo meccanismo può portare alla nascita di problematiche (dangling pointer, oggetti inaccessibili, sovrascritti...)

I blocchi possono essere realizzati in diversi linguaggi `(C | c++ | Java { … }), Algol begin … end, ML let … in … end.`
Possono eesere blocchi inline, oppure associati a funzioni o procedure

Modello della macchina



Particolare attenzione va posta al DataSegment. Composto da

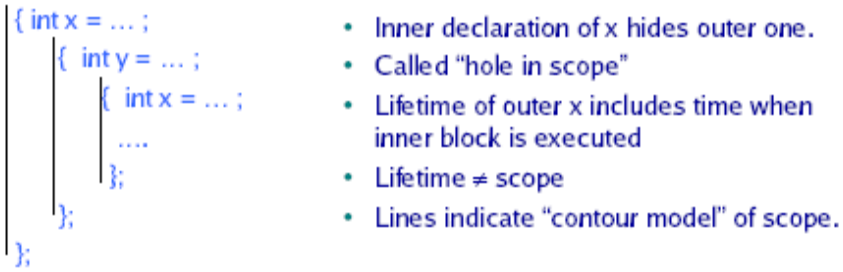
- Stack, un'area di memoria contenente le variabili relative ai blocchi attivi in quel momento.
- Heap, un'area di memoria contenente variabili con diverso lifetime
- Environment pointer, è un registro che punta alla posizione corrente sullo stack

L'inserimento di un nuovo blocco porta alla creazione di un record di ativazione sullo stack. Al contrario l'uscita da un blocco porta alla cancellazione del blocco di attivazione di più recente.

Definizione di Scope (o visibilità o portata): contesto in cui una variabile viene dichiarata all'interno di un programma. Tutte le variabili visibili a tempo di esecuzione in un certo istante sono

contenute nell'ambiente.

Definizione di Lifetime (ciclo di vita): periodo di tempo durante il quale una determinata locazione di memoria è assegnata a una variabile.

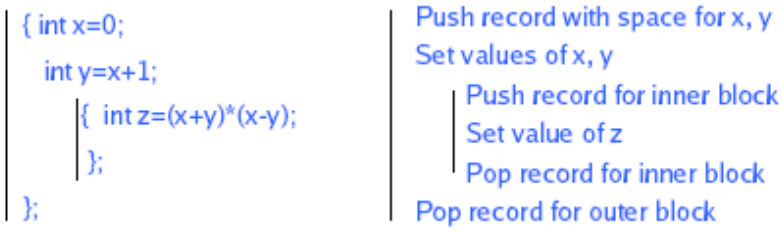


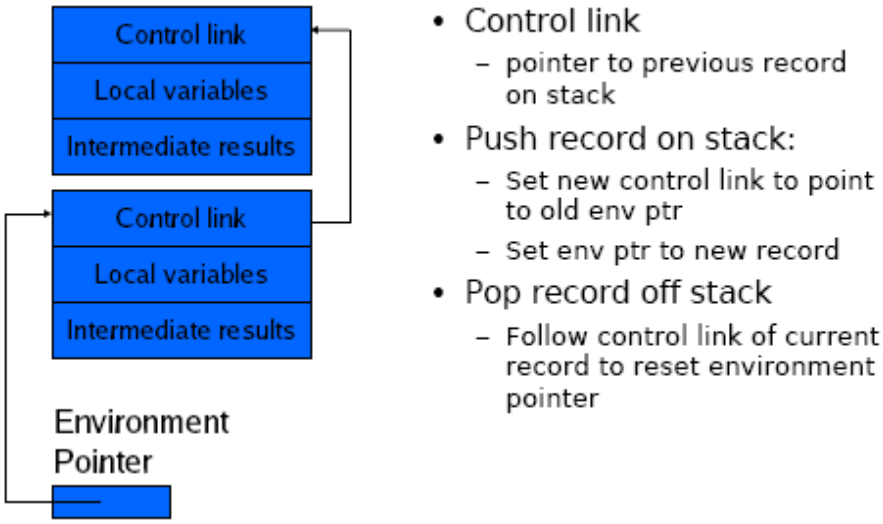
Il blocco più esterno definisce una variabile x. Allo stesso modo il blocco più interno crea una variabile x. Questa situazione viene detta "hole in scope".

Si nota come il ciclo di vita della variabile x perduri anche quando viene chiamato il blocco più interno, ma in tal caso la x più esterna non è visibile. Si nota come il lifetime sia diverso dallo scope.

Gestione In-line Blocks

Activation record: struttura di dati creata dinamicamente sullo stack, essa contiene lo spazio per le variabili locali.





L'Activation record si compone di 3 principali componenti

- Control link, è un puntatore che punta al precedente record di attivazione sullo stack
- Local variables, area di memoria nella quale sono definite le variabile del blocco
- Intermediate result, area di memoria nella quale vengono poste le variabili intemedie (calcoli)

Regole di scope

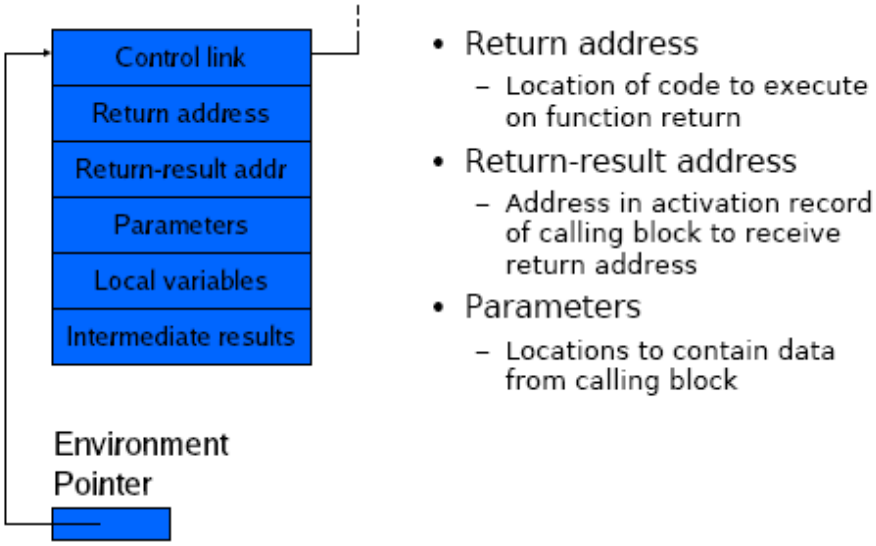
- x, y are local to outer block
- z is local to inner bock
- x, y are global to inner block

```
{ int x=0;
  int y=x+1;
    { int z=(x+y)*(x-y);
    };
};
```

Functions and procedures

In questo caso l'activation record risulta essere più complesso perché composto da

- parametri
- indirizzo di ritorno
- valori ritornati
- locazione nella quale porre il valore ritornato



Lvalue e Rvalue

L:= R (l = left value la locazione | R = right value il valore)

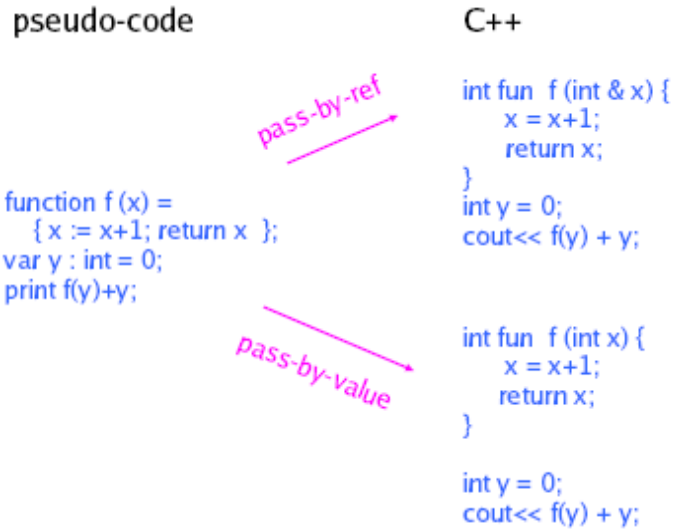
Formalmente, un' espressione che può stare a sinistra di un operatore di assegnamento, ovvero qualcosa in cui può essere scritto un valore, è detta un **l-value** (contrazione di left-value). Una variabile è l'esempio tipico, ma non l'unico, di l-value. Una espressione che può essere solo letta o valutata, ma alla quale non è possibile assegnare un valore, è detta **r-value**. *(tratto da wikipedia)*

NON HO MESSO ML (SLIDE #19/47)

Parameter passing

Il passaggio dei parametri avviente tramite

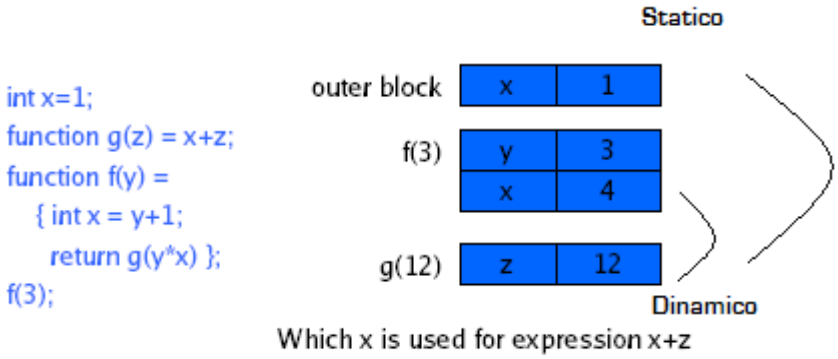
- **Reference**
    - il chiamante pone nell' activation record stack il l-value (l'indirizzo della variabile)
    - in questo modo il valore della variabile può essere modificata perchè i cambiamenti vengono apportati all'indirizzo effettivo dove è allocata la variabile
    - l'indirizzo del parametro attuale viene copiato sul record di attivazione
    - aliasing, si ha quando due variabili diverse si riferiscono alla stessa locazione di memoria.
- **By Value**
    - il chiamante pone nell'activation record il contenuto della variabile
    - la funzione può cambiare il valore nell'activation record, ma non quello effettivo della variabile.
    - Il valore del parametro attuale viene copiato nel record di attivazione come valore del parametro formale



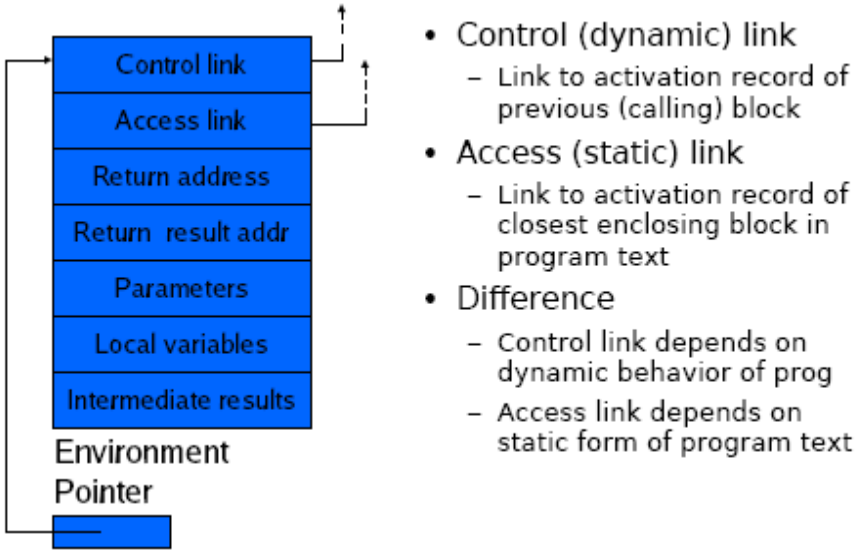
Access to global variables

Se una variabile X compare nel corpo della funzione, ma non è dichiarata dentro la procedura, significa che essa è stata dichiarata all'esterno della funzione. Tale situazione porta a dire che X si trova in un altro record di attivazione diverso da quello corrente. Partendo quindi dal presupposto che X sia stata dichiarata in un blocco, si utilizzano due diverse tecniche per accedere a tale variabile.

- **Static scope**, nel caso di variabili con lo stesso nome ma in blocchi diversi ci si riferisce al blocco più esterno
- **Dynamic scope**, si punta al blocco più recentemente usato rispetto a quello che invoca l'accesso alla variabile



Record di attivazione nel caso di scope statico



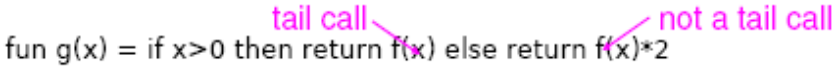
- **Control link,** puntatore all'activation record parent.
- **Access link,** puntatore all'activation record outer

Tail recursion

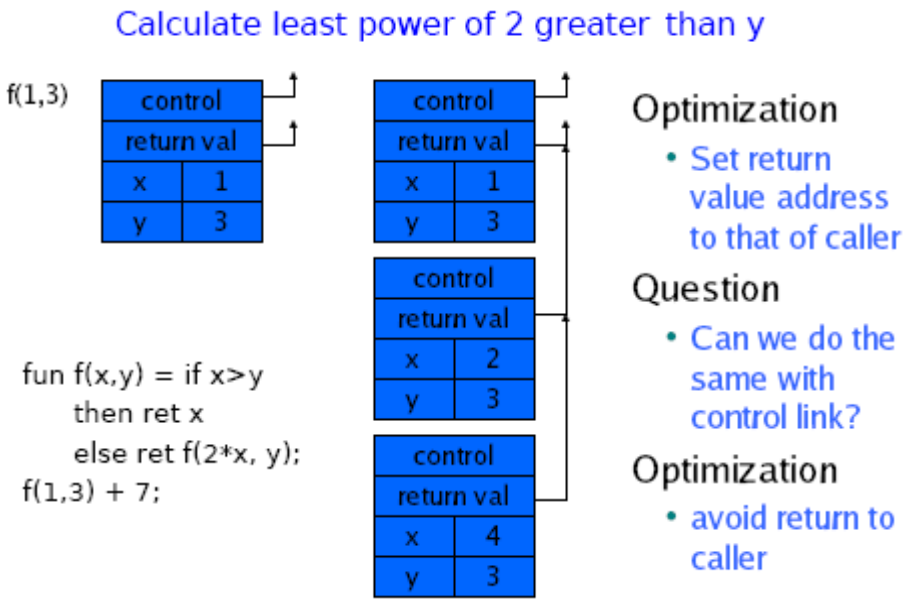
Ottimizzazione messa in atto dal compilatore, ad esempio quando si usano funzioni ricorsive al fine di ridurre la memoria usata per il record di attivazione.

Siano date due funzione f() e g(). Si dice *tail call* se g() chiama f() senza nessun altra computazione.

In questa situazione chiamare f() o g() sarebbe la stessa cosa.

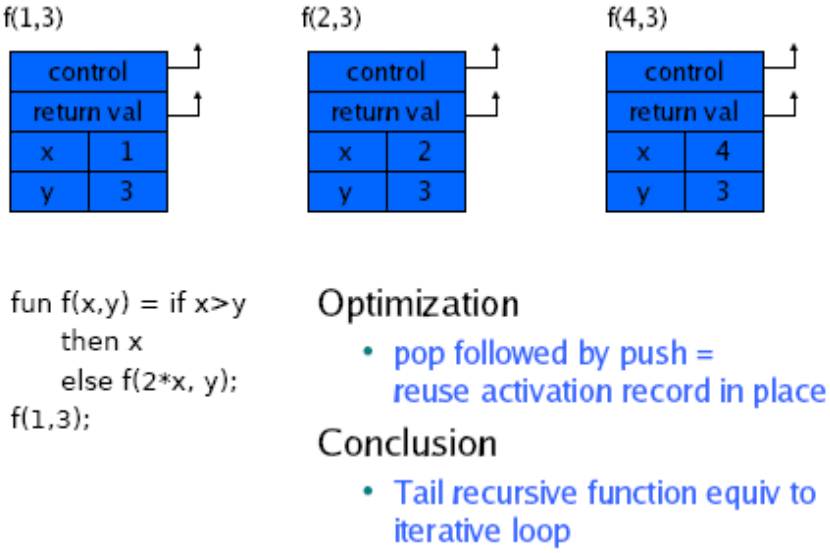


Il vantaggio della tail recursion è quello di usare sempre lo stesso record di attivazione per tutte le chimate ricorsive.



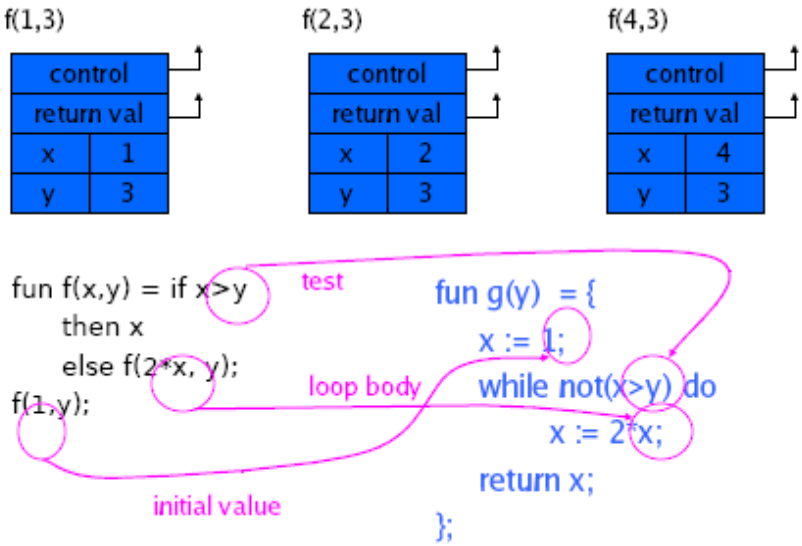
Al termine della terza chiamata, il record di attivazione passa il risultato al secondo che a sua volta passa il risultato al primo e poi al main.

La tail recursion ci permette di evitare questo passaggio, cioè evita la creazione di tre distinti record di attivazione. Infatti viene creato il primo record di attivazione, poi la successiva chiamata alla funzione ricorsiva esegue un pop sullo stack togliendo il record di attivazione e ponendo al suo posto il nuovo record di attivazione. In questo modo si avrà un solo record di attivazione attivo alla volta.



Tail recursion as Iteration

La tail recursion se scritta correttamente può essere usata anche per scrivere un loop, infatti grazie al fatto che usa un solo record di attivazione essa può essere paragonata in tutti i sensi ad un classico ciclo. Le due versioni portano ad avere lo stesso risultato seguendo le stesse operazioni.


f(1,3)
control
return val
x
1
y
3
f(2,3)
control
return val
x
2
y
3
f(4,3)
control
return val
x
4
y
3
fun f(x,y) = if x>y
then x
else f(2*x, y);
f(1,y);
test
loop body
initial value
fun g(y) = {
x := 1;
while not(x>y) do
x := 2*x;
return x;
};