Programmazione Avanzata

2024-2025

Introduzione

Il linguaggio di programmazione ideale facilita la scrittura di programmi succinti e chiari. Questo ne permette la comprensione, modifica e mantenimento durante l'intero ciclo di vita. Aiuterà inoltre i programmatori a gestire l'interazione tra le componenti di un sistema software complesso. A un software è richeisto di essere affidabile, manutenibile ed efficiente. Ad un linguaggio di programmazione si chiede di essere scrivibile, ovvero di permettere la stesura di una soluzione in modo non contorto; leggibile, ovvero di permettere di riconoscere la correttezza o gli errori direttamente dalla sintassi, senza eseguire; semplice, ovvero facile da apprendere e applicare; sicuro, ovvero contenere protezioni contro la scrittura di codice malevolo; robusto, ovvero resistente ad eventi indesiderati.

Agli inizi, la programmazione era effettuata direttamente in codice macchina per ottenere programmi piccoli ed efficienti. È negli anni '50 che emergono i primi due linguaggi, Fortran e Cobol. Il primo permette di scrivere programmi in forma matematica, il secondo è adatto all'uso bancario. Le necessità dei programmatori sono cambiate nel corso degli anni. Funzionalità e paradigmi un tempo considerati inefficienti, come la ricorsione e la programmazione ad oggetti, sono oggi diventati la norma. Le caratteristiche del linguaggio sono, in ogni caso, definite al punto d'incontro tra le necessità umane del programmatore e le necessità tecniche dell'architettura di Von Neumann della macchina sottostante. Possiamo distinguere diversi paradigmi. La programmazione procedurale sceglie la *routine* come unità base per la modularizzazione. La programmazione imperativa si basa su istruzioni, definite a passaggi, che modificano valori. La programmazione funzionale segue un approccio simile a quello matematico, basato su espressioni e funzioni. La programmazione a oggetti si basa sul concetto di classe come unità base. La programmazione *abstract data type* usa i tipi astratti come unità base.La programmazione dichiarativa cerca di definire il problema tramite regole, invece di descrivere i passaggi per trovare la soluzione.

Computabilità

Un programma per computer è interpretabile come funzione matematica dello stato della macchina prima dell'esecuzione e degli ingressi forniti dall'utente. Esso può implementare solo funzioni computabili, ovvero in grado di produrre un risultato. Questi può essere impossibile da raggiungere per errori nella funzione, oppure a causa di un tempo di esecuzione infinito. Alcune funzioni possono essere computabili in principio, ma non in pratica (se il tempo di computazione eccede limiti materiali). Si definisce *funzione parziale* una funzione definita solo per certi argomenti. Usando le definizioni matematiche:

- funzione computabile: $f: A \to B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano le seguenti condizioni:
  - $< x, y > \in f$ e $< x, z > \in f \ \to \ y = z$
  - $\forall x \in A, \quad \exists y \in B \ / \ < x, y > \in f$
- funzione parziale $f: A \to B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano la seguente condizione:
  - $< x, y > \in f$ e $zx, z > \in f \ \to \ y = z$

Possiamo fornire una definizione alternativa di computabilità. Una funzione è computabile se esiste un algoritmo che permetta di produrre il risultato desiderato per qualsiasi ingresso appartenente al dominio. Anche quando l'algoritmo esiste, la sua implementabilità dipende dal linguaggio di programmazione scelto. La classe delle funzioni computabili sui numeri naturali coincide con la classe delle funzioni parziali ricorsive. Questo perché la ricorsione è essenziale nella computazione, e perché la maggior parte delle funzioni è parziale.

Un'altra definizione di computabilità si basa sul concetto di macchina di Turing. Una macchina di Turing è un sistema bicomponente. Il primo elemento è un nastro, diviso in celle di memoria, sul quale sia possibile leggere, scrivere e muoversi di una cella per volta. Il secondo elemento è un controllore a stati finiti, che opera sul nastro per leggerlo, per scriverci o per muovervisi di una cella. Una funzione sui numeri naturali è computabile con un metodo efficace se e solo se è computabile da una macchina di Turing. Questo teorema è dimostrabile con tre dimostrazioni: Alonso Church, Alan Turing e Calcolo Lambda. Tutti i linguaggi di programmazione sono *Turing-complete*.

In una quarta definizione, la computabilità è riconducibile all'*halting problem*, che consiste nel determinare se un programma terminerà in corrispondenza di un certo ingresso. Possiamo associare il problema ad una funzione f_{halt} a

doppio ingresso (programma, input) che ritorna *halt* o $\neg halt$ se il programma termina o meno. Supponendo di avere un programma in grado di risolvere il problema, ovvero che abbia lo stesso output di f_{halt}, possiamo utilizzarlo per creare un programma che a volte non termina.

Compilatori, calcolo lambda, semantica denotativa, divisione dei linguaggi

Un compilatore traduce il programma in istruzioni macchina, mentre un interprete traduce ed esegue allo stesso tempo. Il compilatore è divisibile in componenti. Il *lexical analyzer* raggruppa le istruzioni in *token*. Il *syntax analyzer* o *parser* raggruppa i *token* in espressioni, *statement* e dichiarazioni, in vase a regole grammaticali. Il prodotto del *parser* è il *parse tree*, una struttura dati che rappresenta il programma. Il *semantic analyzer* applica regole e procedure aggiuntive in base al contesto delle espressioni, come ad esempio il *type checking*, producendo un *augmented parse tree*. L'*intermediate code generator* produce una prima versione non ottimizzata del codice, in un formato chiamato *intermediate representation*. Il *code optimizer* elimina sottoespressioni, sostituisce variabili duplicate, rimuove istruzioni inutilizzate e rimpiazza le chiamate a funzioni brevi con il rispettivo codice (*inlining*) quando esso è più efficiente. Il *code generator* converte il codice intermedio in codice macchina per il *target* desiderato.

Distinguiamo tra sintassi (il testo di un programma) e semantica (la funzionalità che rappresenta). Una grammatica è composta da un simbolo iniziale, un insieme di simboli non temrinali, un insieme di terminali e un insieme di regole di produzione. Essa fornisce un metodo per definire un insieme infinito di espressioni. I non terminali sono i simboli utilizzati per esprimere la grammatica, mentre i terminali sono i simboli che appaiono nel linguaggio. Una grammatica è detta *ambigua* se la stessa espressione ammette più di un *parse tree*. I linguaggi umani uniscono ambiguità, frasi imperative, dichiarative, e interrogative. I linguaggi imperativi uniscono dichiarazioni e assegnamenti. Nella *semantica denotazionale* un programma è una funzione matematica da stato a stato. Lo stato è una funzione matematica che rappresenta i valori della memoria in un determinato stato dell'esecuzione di un programma.

Il lambda calculus è una notazione per descrivere la computazione, composta da tre parti. La prima è una notazione per descrivere le funzioni. La seconda è un meccansimo di prova per descrivere equazioni tra epsressioni. La terza è un insieme di regole di calcolo chiamate riduzioni. I due concetti principali del calcolo lambda sono le astrazioni lambda, per cui se M è un'espressione, $\lambda x.M$ è la funzione ottenuta trattando M come una funzione di x, e l'applicazione, ovvero l'anteposizione di un'espressione davanti ad un'altra per ottenere la composizione. Ad esempio, in $(\lambda x.x)M = M$, applichiamo una funzione identità all'espressione M. Un linguaggio di programmazione è interpretabile come un'applicazione del calcolo lambda, ovvero l'unione del calcolo lambda puro con tipi di dati aggiuntivi. Introduciamo ora il concetto di assegnazione delle variabili. Si dice libera una variabile che non è dichiarata nell'espressione (recuperare la definzione di algebra e logica). Nel calcolo lambda, al contrario di quanto avvenga nei linguaggi di programmazione procedurali come C, l'assegnamento di variabili non ha alcun effetto secondario ed è puramente funzionale.

Gestione della memoria

Lo *stack* serve soltanto per la ricorsione, mentre l'*heap* serve soltanto per le strutture dati dinamiche. Quando queste due funzionalità non sono necessarie, la memoria può essere gestita staticamente. Ad esempio il linguaggio FORTRAN, non permettendo la ricorsione, non aveva un record di attivazione. La memoria occupata da un programma era stimabile in modo esatto al momento della compilazione. I linguaggi moderni, invece, permettendo la ricorsione e l'allocazione dinamiche, necessitano di record di attivazione e di una gestione più complessa della memoria. Molti linguaggi sono *block-structured*, ovvero le variabili sono accessibili solo all'interno del loro blocco (variabili locali) o di sottoblocchi interni ad esso. Nell'analisi del ciclo di vita di variabili, distinguiamo tra *scope*, ovvero la regione di spazio (a livello di blocchi) in cui la variabile è attiva, e *lifetime*, ovvero tempo di allocazione. Queste due metriche possono non corrispondere. Ad esempio, se dichiaro una variabile in un blocco interno che abbia lo stesso nome di quella esterna, ottengo un *"hole in scope"* in cui la variabile esterna è ancora attiva ma non accessibile. Ogni blocco vede come globali tutte le variabili dichiarate in blocchi di livello superiore. Lo spazio in memoria viene allocato all'ingresso nel blocco, e deallocato all'uscita. I blocchi possono essere legati a funzioni, *inline*, oppure legati a istruzioni quali controllo di flusso e cicli. C e C++ non permettono la dichiarazione di funzioni locali innestate.

Viene creato un record di attivazione ad ogni ingresso in un blocco. Ogni record di attivazione contiene, in ordine, un *control link* ovvero un puntatore al record precedente sullo stack, delle variabili locali e dei risultati intermedi. L'indirizzo nell'*environment pointer* è al *control link* del record in cima allo *stack*. Una macchina standard contiene dei registri standard, il codice e un registro chiamato *program counter* o *instruction pointer* con l'indirizzo dell'istruzione corrente, i dati (*stack* e *heap*) e un *environment pointer* o *stack pointer* con l'indirizzo della cima dello stack. Le chiamate a funzioni richiedono il passaggio dei parametri, il salvataggio dell'indirizzo di ritorno, il salvataggio di variabili locali e risultati intermedi, e l'allocazione di spazio per il valore di ritorno. I parametri delle funzioni possono essere valutati al momento del passaggio, oppure essere lasciati come funzioni per la *lazy evaluation*, in base al tipo di linguaggio. Il passaggio può avvenire per *reference* o per valore. Il passaggio per valore, richiedendo di copiare il valore del parametro, è più sicuro ma più lento. Riduce però il problema dell'*aliasing*, ovvero il puntamento allo stesso indirizzo di memoria da parte di più variabili. In base alla terminologia il valore effettivo di una variabile può prendere il nome di *R-value* mentre il suo indirizzo è denominato *L-value*.

```c
#include <cstdio>

void f() {
    int x = 5;
    int y = 3;
}

int main(int argc, char *argv[]) {
    f();
    printf("Hello World!");
    return 0;
}
```

```
f():
        sub     sp, sp, #16 ; due byte in più: stack pointer e frame pointer
        mov     w8, #5
        str     w8, [sp, #12]
        mov     w8, #10
        str     w8, [sp, #8]
        add     sp, sp, #16
        ret

main:
        sub     sp, sp, #48
        stp     x29, x30, [sp, #32]
        add     x29, sp, #32
        mov     w8, wzr
        str     w8, [sp, #12]
        stur    wzr, [x29, #-4]
        stur    w0, [x29, #-8]
        str     x1, [sp, #16]
        bl      f()
        adrp    x0, .L.str
        add     x0, x0, :lo12:.L.str
        bl      printf
        ldr     w0, [sp, #12]
        ldp     x29, x30, [sp, #32]
        add     sp, sp, #48
        ret

.L.str:
        .asciz  "Hello World!"
```

In C++ è possibile salvare *reference*. Ad esempio, `int &x = y` crea in `x` una *reference* non modificabile a `y`. Passando per *reference* i parametri per le funzioni, al contrario di quello che avviene con il passaggio per valore, si creano *side effects*. Una versione ibrida è il passaggio per puntatore. L'indirizzo viene passato per copia, ma poi tramite di esso si può modificare la variabile originale come se fosse una *reference*. Per verificare se un linguaggio supporta veramente il passaggio per reference si può provare a costruire una funzione per scambiare il contenuto di due variabili. Non possiamo infatti scambiare il contenuto di due variabili passate per copia, perché la modifica avverrebbe solo sulle copie locali, e verrebbe in ogni caso deallocata al termine dell'esecuzione della funzione. In C++ è possibile scambiare variabili passate per reference. In Java questo non funziona con i tipi base, che sono sempre passati per copia.

```c
int f(int a, int b) {
    if (b==0)
        return a;
    else
        return f(b, a%b);
}

int main(int argc, const char *argv[]) {
    f(15,10);
    return 0;
}
```

```
f:
```

```
        sub     sp, sp, #32
        stp     x29, x30, [sp, #16]   ; store pair, x30=return address
        add     x29, sp, #16    ; x29=frame pointer
        str     w0, [sp, #8]
        str     w1, [sp, #4]
        ldr     w8, [sp, #4]
        cbnz    w8, .LBB0_2
        b       .LBB0_1
.LBB0_1:
        ldr     w8, [sp, #8]
        stur    w8, [x29, #-4]
        b       .LBB0_3
.LBB0_2:    ; il risultato è il resto (divide intero, rimoltiplica, sottrae)
        ldr     w0, [sp, #4]
        ldr     w8, [sp, #8]
        ldr     w10, [sp, #4]
        sdiv    w9, w8, w10
        mul     w9, w9, w10
        subs    w1, w8, w9
        bl      f
        stur    w0, [x29, #-4]
        b       .LBB0_3
.LBB0_3:
        ldur    w0, [x29, #-4]
        ldp     x29, x30, [sp, #16]
        add     sp, sp, #32
        ret

main:
        sub     sp, sp, #48
        stp     x29, x30, [sp, #32]
        add     x29, sp, #32
        mov     w8, wzr
        str     w8, [sp, #12]
        stur    wzr, [x29, #-4]
        stur    w0, [x29, #-8]
        str     x1, [sp, #16]
        mov     w0, #15
        mov     w1, #10
        bl      f
        ldr     w0, [sp, #12]
        ldp     x29, x30, [sp, #32]
        add     sp, sp, #48
        ret
```

Il compilatore può ottimizzare il salvataggio dei valori di ritorno tenendoli in un registro invece di assegnarli allo stack.

Soluzione vecchia (ottimizzata meglio):

Indirizzo	Contenuto	Descrizione
0xF174	00 00 00 0A	10
0xF178	00 00 00 0F	15
0xF17C		Return value
0xF180	...	x29: stack pointerframe pointer
0xF188	...	x30: link registerreturn address

Indirizzo	Contenuto	Descrizione
154	00 00 00 05	5
158	00 00 00 0A	10
15c	...	Return value
160	...	SP

Indirizzo	Contenuto	Descrizione
168	...	LR

Indirizzo	Contenuto	Descrizione
134	00 00 00 00	0 (b)
138	00 00 00 05	5 (a)
13c	...	Return value
140	...	x29 sp
148	...	x30 lr / ra