Generazione del Codice intermedio

Giovedì 10 Gennaio

Il punto della situazione

- Processo di compilazione:
  - Analisi (lessicale, sintattica, semantica)
  - Sintesi (generazione del codice oggetto)
- La generazione del codice oggetto è la fase più complessa di un compilatore perché dipende:
  - dalle caratteristiche del linguaggio sorgente
  - dalle caratteristiche della ***target machine*** (architettura, sistema operativo, ..)
- La generazione del codice può prevedere, inoltre, una fase di ottimizzazione che può dipendere anche da particolari caratteristiche della ***target machine*** come registri, modelli di indirizzamento, memoria, ….

Generazione del codice intermedio

- Per questo motivo prima di procedere alla fase di generazione del codice oggetto, nella maggior parte dei compilatori è prevista la ***generazione di un codice intermedio*** che sarà poi utilizzato dal back-end per generare il codice oggetto.

- Questa fase farà parte del front-end del compilatore.

Vantaggi

- L'introduzione del codice intermedio permette di ottenere notevoli benefici:
  - **Indipendenza del front-end dal back-end**

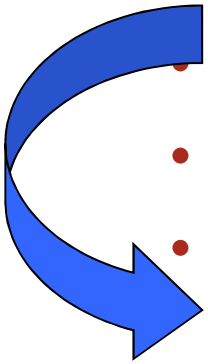
    Nella rappresentazione intermedia non si fa alcuna ipotesi sulla target machine: è possibile quindi definire un back-end per ogni architettura destinataria (re-targeting)
  - **Maggiore semplicità di traduzione nel codice oggetto**

    La rappresentazione intermedia presenta, nella maggior parte dei casi, affinità, in termini di istruzioni, con il linguaggio assembler (che è spesso il linguaggio utilizzato per produrre il codice oggetto)
  - **Maggiore semplicità per le ottimizzazioni**

    Gli algoritmi di ottimizzazione risultano più semplici se operanti su sequenze di istruzioni piuttosto che su strutture più complesse come gli alberi sintattici. Ciò consente di produrre codice più efficiente.

La struttura di un compilatore

Fasi della compilazione



- Analisi lessicale
- Analisi sintattica
- Analisi semantica
- Generazione del codice intermedio

Analisi (front-end)

- Ottimizzazione
- Generazione del codice oggetto

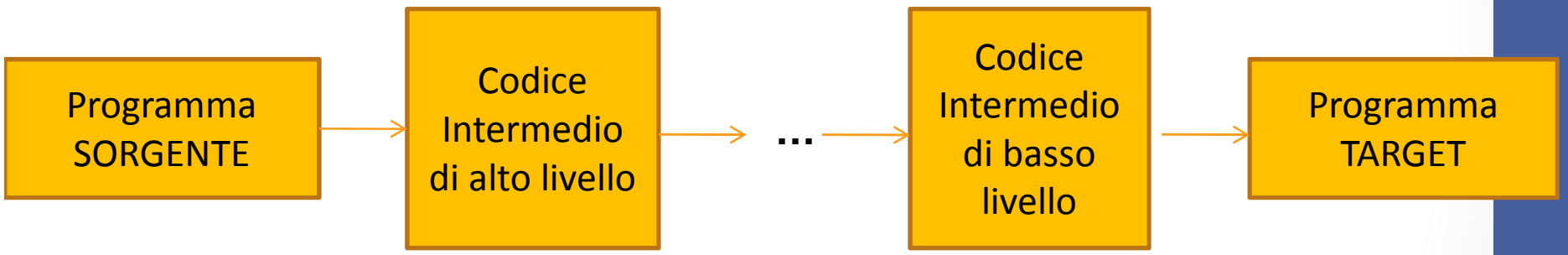
Sintesi (back-end)

Forma del codice intermedio

- Esistono diversi modelli usati per la generazione del codice intermedio.
- Tutti, comunque, si basano sul principio della linearizzazione degli alberi sintattici.
- Il codice intermedio può essere di alto livello o essere più vicino al codice oggetto.
- Può incorporare o no le informazioni contenute nella tabella dei simboli (regole di visibilità, livelli di annidamento,…). Se non lo fa, tali informazioni dovranno essere gestite in fase di generazione del codice oggetto.
- Uno dei più importanti ed usati è il codice a tre indirizzi: ***three-address-code (3AC)***

Più codici intermedi

- Possono esistere vari step di generazione del codice intermedio



Codice a tre indirizzi

- Il nome “codice a tre indirizzi” deriva dal fatto che di solito ogni istruzione contiene tre indirizzi: due per gli operandi e uno per il risultato.

- Il codice a tre indirizzi è una sequenza di istruzioni generalmente del tipo:

$$\mathbf{id} := \mathbf{id}_1 \ \textbf{\textit{operator}} \ \mathbf{id}_2$$

  dove **id, id_1** e **id_2** sono nomi, costanti (tranne **id**), o nomi temporanei generati dal compilatore, ed ***operator*** è un operatore.

- Ad esempio l’espressione **x + y * z** potrebbe essere tradotta:

$$\mathbf{t_1 := y * z}$$
$$\mathbf{t_2 := x + t_1}$$

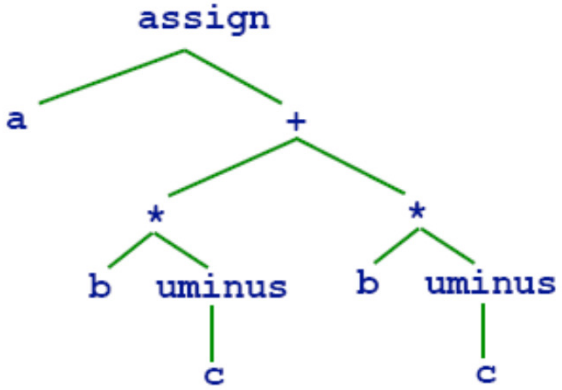
  dove t_1 e t_2 sono nomi temporanei generati dal compilatore.

Codice a tre indirizzi

- Il codice a tre indirizzi consente di linearizzare le rappresentazioni ad albero sintattico.
- Per esempio l'espressione

```
a := b*-c + b*-c
```

può essere descritta come:



Albero sintattico

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

3AC

```
t1 := - c
t2 := b * t1
t3 := t2 + t2
a := t3
```

3AC ottimizzato

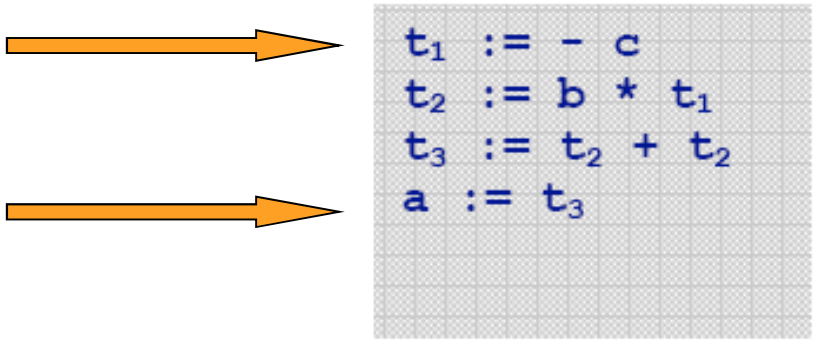
Codice a tre indirizzi

- Ovviamente la forma dell'istruzione:

  $$\mathbf{id} := \mathbf{id}_1 \ \textit{operator} \ \mathbf{id}_2$$

  è insufficiente per rappresentare tutte le caratteristiche di un linguaggio di programmazione.

- Per esempio gli operatori unari, come la negazione o l'assegnazione, richiedono una variante del 3AC che contiene due soli indirizzi.



Non esiste un 3AC standard

- Per rappresentare, quindi, in 3AC tutti i costrutti di un linguaggio di programmazione sarà necessario adattare la forma del 3AC per ogni costrutto.
- Se il linguaggio ha caratteristiche insolite, bisognerà “inventarsi” nuove forme di 3AC.
- Questa è una delle ragioni per cui non esiste una forma standard per il 3AC.

Indirizzi e istruzioni

- Un indirizzo può essere:
  - un nome;
    nell'implementazione tale nome è sostituito da un puntatore alla TS;
  - una costante;
    può essere di vario tipo; il compilatore potrebbe operare opportune conversioni di tipo;
  - indirizzo temporaneo;

    è utile per le ottimizzazioni usare variabili temporanee

    con nomi diversi

Indirizzi e istruzioni

- Ogni istruzione può essere contrassegnata da una label simbolica. Le istruzioni possono essere:
  - Assegnazioni della forma **x=y op z**, dove op è un'operazione logica o aritmetica;
  - Assegnazioni della forma **x=op y**, dove op è un'operazione unaria;
  - Istruzioni di copia **x=y**;
  - Salto incondizionato **goto L**, dove L è la label di un'istruzione;
  - Salti condizionati della forma **if x goto L** o **ifFalse x goto L**;
  - Salti condizionati della forma **if x relop y goto L**;

Indirizzi e istruzioni

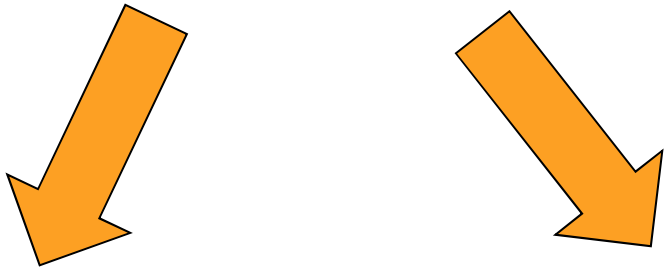
- Chiamate di procedure: **param x** per i parametri, **call p,n** per le procedure, **y=call p,n** per le funzioni, **return y** è opzionale;

  param x_1
  param x_2
  …
  param x_n
  call p,n

  genera una chiamata della procedura $p(x_1,x_2,\ldots,x_n)$
- Copie indicizzate della forma **x=y[i]** e **x[i]=y**; (la prima pone x al valore della locazione che si trova i unità dopo y, la seconda pone uguale a y il contenuto della locazione che si trova i unità dopo x.
- Assegnazioni di indirizzi della forma **x=&y** (pone il valore di x uguale alla locazione di y)
- Assegnazioni di puntatori della forma ***x=y e x=*y**;

Esempio

```
do i=i+1;
while (a[i]<v);
```



ETICHETTE SIMBOLICHE

```
L: t1=i+1
   i=t1
   t2=i*8
   t3=a [t2 ]
   if t3 < v goto L
```

INDICI DI POSIZIONE

```
100: t1=i+1
101: i=t1
102: t2=i*8
103: t3=a [t2 ]
104: if t3 < v goto 100
```

La dimensione di ogni elemento dell'array ha dimensione pari a 8 unità

Esempio in linguaggio SimplePas

Programma per il calcolo del fattoriale

```
read x; { input an integer }
if 0 < x then { don't compute if x <= 0
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

Tree address code

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Esempio di codice 3AC

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

if_false: istruzione di salto condizionale a 2 indirizzi (usata per tradurre sia il costrutto if che il repeat)

label: istruzione a un indirizzo, segnano la posizione di indirizzi di salto;

halt: istruzione senza indirizzi che serve per marcare la fine del codice

Strutture dati per l'implementazione del 3AC

- Il 3AC, normalmente, non è implementato in forma testuale ma:
  - ogni istruzione in 3AC è realizzata attraverso un record con campi per gli operatori ed operandi;
  - la sequenza delle istruzioni in 3AC è implementata attraverso un vettore o una lista concatenata.

Implementazione delle istruzioni in 3AC

- Le istruzioni vengono implementate con record con 4 campi (uno per l'operatore e tre per gli indirizzi degli operandi).
- Questi record vengono chiamati **quadruple**.
- Per quelle istruzioni che hanno bisogno di meno indirizzi i campi relativi saranno lasciati "vuoti".

Esempio

- Consideriamo l'espressione:

```
a := b*-c + b*-c
```

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Codice 3AC

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Quadruple

Esempio

Tree address code

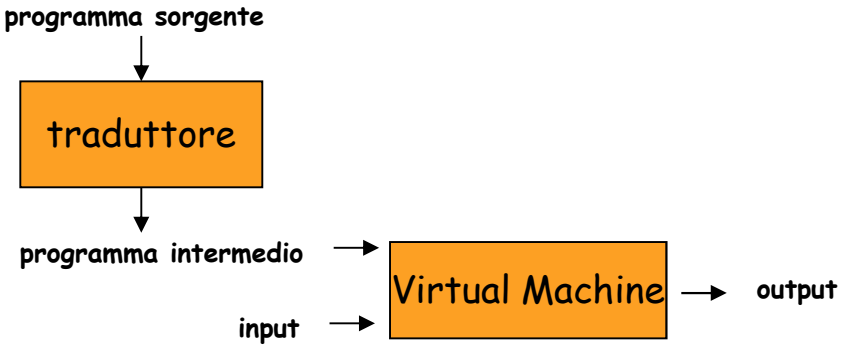
```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Quadruple

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

Codice per macchina virtuale

- Codice oggetto in forma intermedia (è considerato codice intermedio):
  - ***p-code*** prodotto dal Pascal p-compiler per una ***Virtual Stack Machine***
  - ***bytecode*** prodotto dai compilatori Java per una ***Virtual Java Machine***
  - ***Common Intermediate Language (CIL)*** della piattaforma .NET eseguito dal ***Common Language Runtime (CLR)***, la macchina virtuale .NET progettata da Microsoft per funzionare con qualsiasi sistema operativo.



Un esempio: il P-Code

- Questo codice è stato progettato negli anni 70-80 per essere eseguito da una ***Virtual Stack Machine (P-machine).***
- Contiene descrizioni e informazioni specifiche legate alla struttura della P-machine
- Supponiamo che la ***P-machine*** sia costituita da:
  - una memoria per il codice
  - una memoria per le variabili
  - uno stack per i dati temporanei
  - una serie di registri per la gestione dello stack e il supporto dell'esecuzione.

Un esempio di P-Code

- Consideriamo l'espressione: **2*a+(b-3)**

- Una versione in P-code per questa espressione è la seguente:

```
ldc 2          ; load constant 2
lod a          ; load value of variable a
mpi            ; integer multiplication
lod b          ; load value of variable b
ldc 3          ; load constant 3
sbi            ; integer subtraction
adi            ; integer addition
```

Un esempio di P-Code

- Consideriamo l'assegnazione: **x := y+1**
  che corrisponde alle seguenti istruzioni in P-code:

```
lda x          ; load address of x
lod y          ; load value of y
ldc 1          ; load constant 1
adi            ; add
sto            ; store top to address
               ; below top & pop both
```

```
read x; { input an integer }
if 0 < x then { don't compute if x <= 0
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

P-code per il programma per il calcolo del fattoriale

```
lda x       ; load address of x
rdi         ; read an integer, store to
            ; address on top of stack (& pop it)
lod x       ; load the value of x
ldc 0       ; load constant 0
grt         ; pop and compare top two values
            ; push Boolean result
fjp L1      ; pop Boolean value, jump to L1 if fals
lda fact    ; load address of fact
ldc 1       ; load constant 1
sto         ; pop two values, storing first to
            ; address represented by second
lab L2      ; definition of label L2
lda fact    ; load address of fact
lod fact    ; load value of fact
lod x       ; load value of x
mpi         ; multiply
sto         ; store top to address of second & pop
lda x       ; load address of x
lod x       ; load value of x
ldc 1       ; load constant 1
sbi         ; subtract
sto         ; store (as before)
lod x       ; load value of x
ldc 0       ; load constant 0
equ         ; test for equality
fjp L2      ; jump to L2 if false
lod fact    ; load value of fact
wri         ; write top of stack & pop
lab L1      ; definition of label L1
stp
```

Confronto tra P-code e 3AC

- Il P-code in alcuni aspetti è più vicino, rispetto al 3AC, agli attuali linguaggi assemblativi.
- Le istruzioni in P-code richiedono meno indirizzi (uno o zero indirizzi).
- P-code è meno compatto del 3AC in termini di numero di istruzioni.
- P-code non è “self-contained” in quanto le istruzioni operano implicitamente sullo stack (e questi indirizzi impliciti sullo stack sono proprio gli indirizzi che mancano nelle istruzioni).
- Il vantaggio dell’uso dello stack sta nel fatto che il compilatore non ha bisogno di assegnare nomi ai valori che necessitano in ogni punto del codice così come accade nelle istruzioni in 3AC.
- Si possono usare strutture dati analoghe a quelle usate per il 3AD

Tecniche per la generazione del codice intermedio

- La generazione del codice intermedio (o direttamente il target code senza codice intermedio) può essere vista come il calcolo di un attributo.
- Infatti, se il codice generato è visto come un attributo stringa, allora questo codice diventa un attributo sintetizzato che può essere definito usando una grammatica con attributi.
- Il codice intermedio potrà quindi essere generato direttamente durante il parsing.

Considerazioni

- La generazione del codice intermedio come calcolo di attributi sintetizzati è semplice.
- Di fatto non è una tecnica molto usata nei moderni compilatori per diverse ragioni:
  - la concatenazione di stringhe è un'operazione che ha un notevole peso computazionale e che richiede una notevole quantità di memoria;
  - nel caso di grammatiche con attributi ereditati la generazione del codice è più complessa.
- Ecco perché spesso la generazione del codice intermedio avviene operando, con appositi moduli, direttamente sull'albero sintattico.

Criteri generali per il C.I. (if then)

- Gli operatori booleani si traducono in salti.

```
if (x<100 || x>20 && x!=y) x=0;
```

```
        if x<100 goto L2
        ifFalse x>20 goto L1
        ifFalse x!=y goto L1
L2:     x=0
L1:
```

Il compilatore può ottimizzare la valutazione, calcolando la porzione minima necessaria a stabilire se tutta l'espressione sia vera o falsa.

Criteri generali per il C.I. (if then else)

- Gli operatori booleani si traducono in salti.

```
if (x<100) x=0;
else x=1;
```

```
        if x<100 goto L2
        ifFalse x<100 goto L1
L2:     x=0
        goto L3
L1:     x=1
L3:
```

Criteri generali per il C.I. (while)

- Gli operatori booleani si traducono in salti.

```
while (x<100) x=x+1;
```

```
L3:    if x<100 goto L2
       ifFalse x<100 goto L1
L2:    t1=x+1
       x=t1
       goto L3
L1:
```

Criteri generali per il C.I. (assegnazione di valori booleani)

- `x=a<b && c<d;`

```
        ifFalse a<b goto L1
        ifFalse c<d goto L1
        t=true
        goto L2
L1:     t=false
L2:     x=t
```

Criteri generali per il C.I. (switch)

- Switch statements

```
switch (E) {
    case V1:S1
    case V2:S2
          …
    case Vn-1:Sn-1
    default: Sn
}
```

```
valuta E in t
    goto test
L1:codice per S1
    goto next
L2:codice per S2
    goto next
          …
Ln:codice per Sn
test:if t=V1 goto L1
      if t=V2 goto L2
      …
     goto Ln
next:
}
```

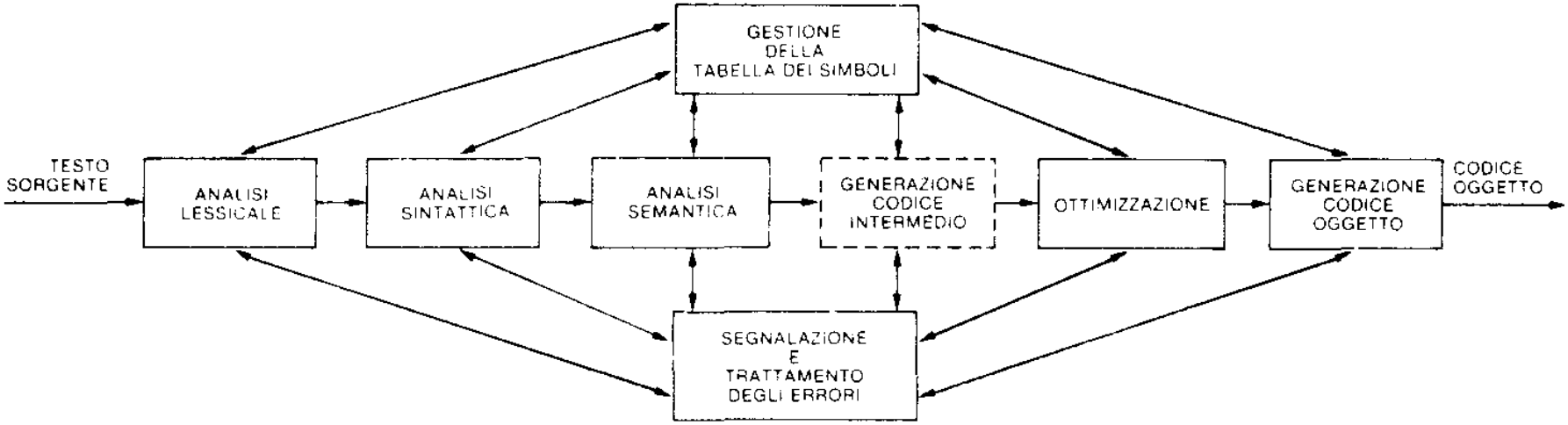
Criteri generali per il C.I. (chiamata di funzione)

- `n=f(a)`

```
t1=a
param t1
t2=call f,1
n=t2
```

Fase di sintesi

- In generale, a questo punto è possibile passare alla fase della generazione del codice oggetto (target code) che può essere direttamente in linguaggio macchina o, come accade spesso, in linguaggio assemblativo.
- In questa fase è possibile prevedere eventuali azioni di ottimizzazione del codice.



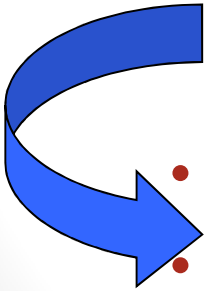
GENERAZIONE DEL CODICE OGGETTO

La struttura di un compilatore

Fasi della compilazione

- Analisi lessicale
- Analisi sintattica
- Analisi semantica
- Generazione del codice intermedio

Analisi (front-end)



- Ottimizzazione
- Generazione del codice oggetto

Sintesi (back-end)

Ottimizzazione del codice

- La traduzione di un programma in codice intermedio e la successiva generazione del codice, producono un codice oggetto solitamente inefficiente rispetto a quello che potrebbe scrivere direttamente nel linguaggio assemblativo un esperto programmatore.
- Ecco perché molti compilatori cercano di applicare delle trasformazioni migliorative, dette ottimizzazioni, allo scopo di rendere il codice prodotto più efficiente.
- Le trasformazioni possono operare a livello del codice intermedio (ottimizzazioni *indipendenti dalla macchina*).
- Esistono anche ottimizzazioni *dipendenti dalla macchina* operate dal *post-ottimizzatore* direttamente sul codice oggetto a valle del generatore.

Esempi di ottimizzazioni indipendenti dalla macchina

- **Propagazione dei valori (copy propagation)**
  - Sostituzione di una variabile con il più recente valore ad essa assegnato.
- **Ripiegamento delle costanti (constant folding)**
  - In un'espressione in cui gli argomenti sono costanti si sostituisce il risultato.
- **Riduzione dei salti**
  - Per esempio un salto ad un'altra istruzione di salto.
- **Eliminazione del codice morto**
  - Eliminazione di codice che non può mai essere eseguito perché irraggiungibile.
- **Eliminazione di sottoespressioni comuni**
  - Sostituzione di espressioni già calcolate.

Esempi di ottimizzazioni indipendenti dalla macchina

- **Eliminazione degli assegnamenti ridondanti o inutili**
  - **x=x+0; x=x*1; x=x-0;…**
- **Estrazione del codice invariante da un ciclo**
- **Riduzione del costo delle operazioni**
  - Sostituzione di un'operazione costosa per mezzo di una più veloce. Per esempio x^3 -> x*x*x 2*x -> x+x x/2->x *0,5
- **Espansione in linea delle chiamate di procedure o eliminazione delle ricorsioni in coda (dove l'ultima operazione è la chiamata a procedura).**

```
int gcd(int u, int v)
{if (v==0) return u;
else return gcd(v,u%v);}
```

```
int gcd(int u, int v)
{begin:
if (v==0) return u;
else
     {int t1=v, t2=u%v;
      u=t1; v=t2;
      goto begin;}
}
```

Ricordiamo che

- Il problema di generare un codice target ottimo è indecidibile
- Molti problemi che si incontrano in questa fase, come l'allocazione dei registri, sono computazionalmente intrattabili
- Ci si deve accontentare di tecniche euristiche che generino un buon codice non necessariamente ottimo

Generatori di codice oggetto

- Un generatore di codice ha 3 obiettivi principali:
  - selezione delle istruzioni (scelta delle istruzioni in linguaggio macchina per implementare il C.I.)
  - allocazione e assegnazione dei registri (stabilire quali valori mantenere in quali registri)
  - ordine delle istruzioni (decidere in quale ordine devono essere eseguite le istruzioni)

Input per un generatore di codice oggetto

- L'input è il codice intermedio, unitamente alle informazioni contenute nella tabella dei simboli.
- Può essere il codice a tre indirizzi, rappresentato come quadruple o triple
- Può essere il codice per macchine virtuali, come il p-code o il bytecode, …
- Rappresentazioni lineari di alberi
- …

Il programma target

- E’ strettamente dipendente dall’architettura a livello dell’instruction set della macchina target.
- Architetture più comuni
  - RISC (reduced instruction set computer) – molti registri, istruzioni a 3 indirizzi, insieme semplice di istruzioni
  - CISC (complex instruction set computer) – pochi registri, istruzioni a due indirizzi, istruzioni complesse
  - a stack
- Spesso è espresso in linguaggio assembly.

Codice macchina assoluto e rilocabile

- Un programma in codice macchina assoluto ha il vantaggio di poter essere caricato in memoria in un posizione fissa e immediatamente e rapidamente eseguito
- Un programma in codice rilocabile consente di compilare separatamente vari sottoprogrammi. Tali oggetti rilocabili devono poi essere composti insieme mediante il linking e caricati in memoria mediante il loader. Ciò dà un vantaggio in termini di flessibilità e di riuso del codice.

Selezione delle istruzioni

- La complessità di quest'attività è influenzata da
  - Il livello di astrazione del codice intermedio
  - La natura dell'instruction set
  - La qualità del codice oggetto che si vuole ottenere (velocità e dimensione)

Selezione delle istruzioni

- Può essere modellizzato come un problema di matching tra gli alberi utilizzati per rappresentare il codice intermedio e le istruzioni del linguaggio target.
- Esistono algoritmi di programmazione dinamica per cercare sequenze di codice vicino all'ottimo
- Il problema di trovare il codice ottimo è indecidibile.

Allocazione dei registri

- I registri sono gli elementi di memorizzazione più veloce, ma in genere non sono tanti.
  - Allocazione dei registri: si seleziona l'insieme delle variabili che risiederanno nei registri
  - Assegnazione dei registri: Si seleziona lo specifico registro per ogni variabile
- Il problema di assegnazione ottimo dei registri alle variabili è NP-completo
- Dipende dall'architettura

Ordine di valutazione

- La scelta di un ordine con cui eseguire le istruzioni può richiedere l'uso di un numero inferiore di registri
- La scelta del miglior ordine è un problema NP-completo

Linguaggio target

- Considereremo un modello di una semplice macchina target, che dispone di n registri di uso generale. Tipiche istruzioni sono
  - Operazioni di caricamento o load
    `LD dest, addr` carica in dest il valore contenuto in addr
  - Operazioni di store
    `ST x,r` salva in x il valore del registro r
  - Operazioni di calcolo
    `Op dest, src1, src2`
  - Salti incondizionati
    `BR L` branch su etichetta L
  - Salti condizionati
    `Bcond r , L` branch su L se il valore del registro r soddisfa la condizione cond

Indirizzi nel codice target

- Ogni programma in esecuzione ha a disposizione uno spazio di indirizzamento logico partizionato in 4 aree per dati e codice:
  - Code: definita staticamente, contiene il codice target eseguibile, la dimensione del codice target può essere calcolata a compile-time
  - Static: definita staticamente, contiene le costanti globali e gli altri dati generati dal compilatore, la dimensione può essere calcolata a compile-time
  - Heap: gestita dinamicamente, dedicata agli oggetti creati e distrutti durante l'esecuzione. La dimensione non è determinata a compile-time
  - Stack: gestito dinamicamente, dedicato alla memorizzazione dei record d'attivazione corrispondenti alle chiamate delle procedure. La dimensione non è determinata a compile-time

Generatori di codice oggetto: creazione di blocchi base

- Molto generatori di codice partizionano le istruzioni in C.I in blocchi base che saranno sicuramente eseguiti in sequenza
- Ogni blocco base è una sequenza di istruzioni a 3 indirizzi
- Il flusso di controllo può entrare nel blocco solo attraverso la prima istruzione.
- I blocchi base diventano nodi di un grafo, chiamato diagramma di flusso.

Algoritmo di partizionamento in blocchi base

- Si individuano le istruzioni “leader” che costituiscono l’inizio di un blocco base
- La prima istruzione del C.I. è leader
- Ogni istruzione di destinazione di un salto incondizionato o condizionato è un leader
- Ogni istruzione immediatamente successiva a un salto è un leader

Esempio

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Semplice generazione del codice per un blocco base

- Assumiamo che le istruzioni macchina abbiano la forma
  - LD reg, mem
  - ST mem, reg
  - OP reg, reg, reg
- Per ogni registro si ha un descrittore di registro che tiene traccia dei nomi delle variabili il cui nome si trova in quel momento nel registro.
- Per ogni variabile si ha un descrittore di indirizzo che tiene traccia della locazione in cui si trova il valore corrente di quella variabile

Istruzioni macchina per le operazioni

- Per ogni istruzione a tre indirizzi x=y+z
  - Si selezionano i registri per x, y, e z. Siano Rx, Ry, Rz
  - Se y non si trova già in Ry si genera l'istruzione LD Ry,y', dove y' indica una delle locazioni associate a y
  - Se z non si trova già in Rz si genera l'istruzione LD Rz,z'
  - Si genera l'istruzione ADD Rx,Ry,Rz

Istruzioni macchina per la copia

- Per le istruzioni del tipo x=y
  - Si procede in modo analogo e si generano le istruzioni del tipo LD Ry,y

Completamento del blocco base

- Per ogni variabile x il cui descrittore indica che la locazione associata non è quella della memoria assegnata a x, si genera ST x, R dove R è il registro dove risulta il valore di x alla fine del blocco

Esempi di ottimizzazioni dipendenti dalla macchina

- Le ottimizzazioni dipendenti dalla macchina possono essere suddivise in due classi:
  - Ottimizzazioni relative ad una attenta selezione delle istruzioni e all'efficiente allocazione dei registri
  - Ottimizzazioni relative all'uso di particolari istruzioni in linguaggio macchina (ottimizzazioni a feritoia o peephole optimization) :
    - Eliminazione operazioni load/store ridondanti
    - Eliminazione del codice irraggiungibile
    - Eliminazione di salti superflui
    - Semplificazioni algebriche (x=x+0)
    - Uso di modalità d'indirizzamento con auto-incremento o auto-decremento, nel caso che la macchina target lo consenta.