GENERAZIONE CODICE INTERMEDIO

GENERAZIONE DEL CODICE SORGENTE

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.

STRUTTURA DELLE FASI DI COMPILAZIONE

Analisi lessicale

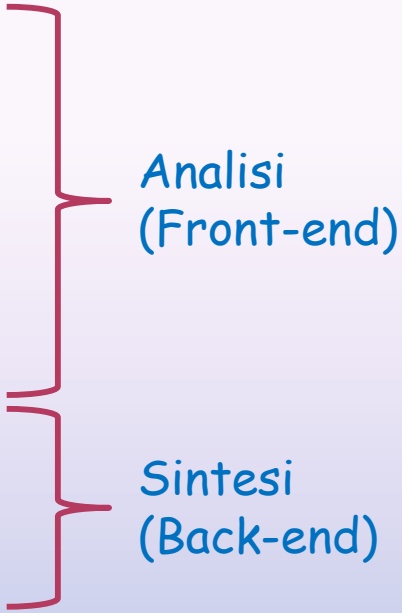
Analisi sintattica

Analisi semantica

Generazione del codice intermedio

Ottimizzazione

Generazione del codice oggetto



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 ad alto livello o essere più vicino al codice oggetto.

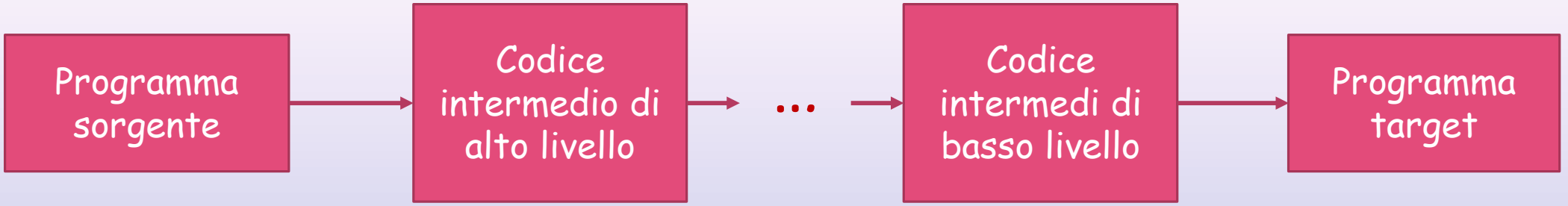
Può incorporare o no le informazioni contenute nella tabella dei simboli (regole di visibilità, livelli di annidamento,...).

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:

```
id:= id1 operator id2
```

Dove `id`, `id1` e `id2` 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:

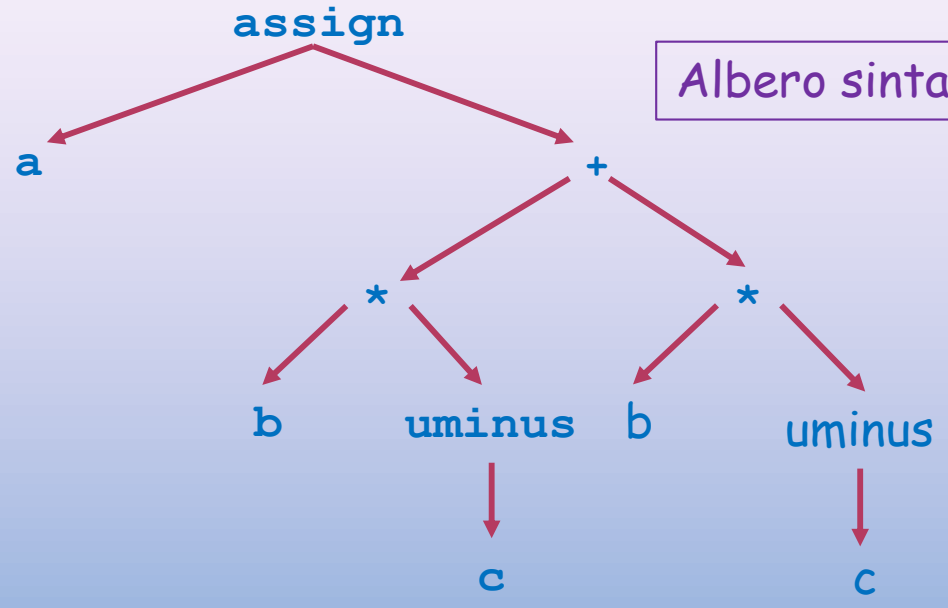
```
t1:=y*z
```

```
t2:=x+t1
```

Dove `t1` e `t2` 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

3AC

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

3AC Ottimizzato

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

CODICE A TRE INDIRIZZI

Ovviamente la forma dell'istruzione:

Id := id1 operator id2

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.

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

NON ESISTE UN 3AC STANDARD

Quindi per rappresentare 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 x1
param x2
…
param xn
call p,n
```

genera una chiamata della procedura p(x1,x2,…,xn)

- Copie indicizzate della forma **`x=y[i]`** e **`x[i]=y`**; (la prima pone il valore di x uguale a quello del valore contenuto nella locazione che si trova i unità dopo y, la seconda pone uguale a y il contenuto della locazione x[i] 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;`**

ETICHETTE - ESEMPIO

```
do i=i+1;
while (a[8*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
```

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 (CONTINUA)

```
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

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
```

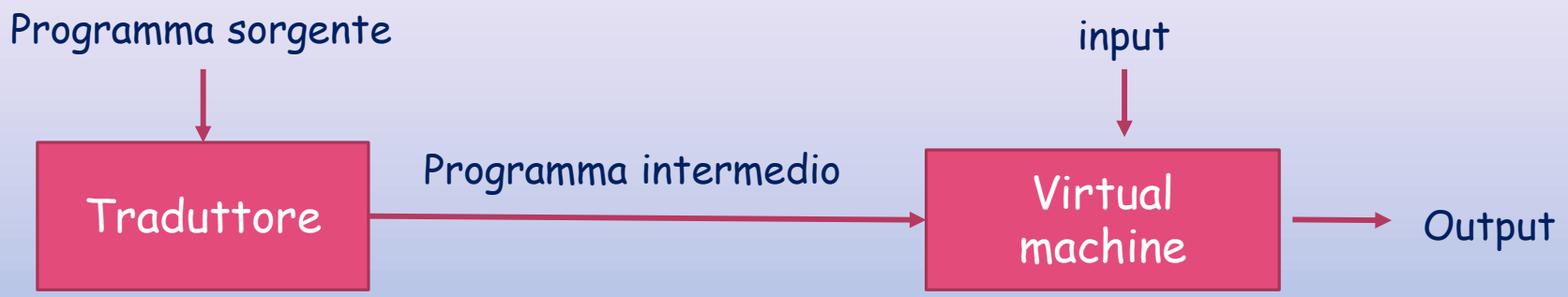
Quadruple

```
(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

Esempi di 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



TECNICHE PER LA GENERAZIONE DEL CODICE INTERMEDIO (C.I.)

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:
```

CRITERI GENERALI PER IL C.I. (IF THEN ELSE)

```
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:…
```

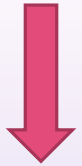
CRITERI GENERALI PER IL C.I. (WHILE)

```
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;
```



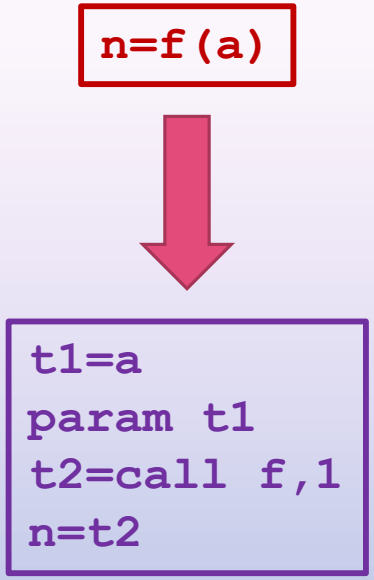
```
iffalse a<b goto L1
iffalse c<d goto L1
t=true
goto L2
L1: t=false
L2: x=t
```

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:
}
```

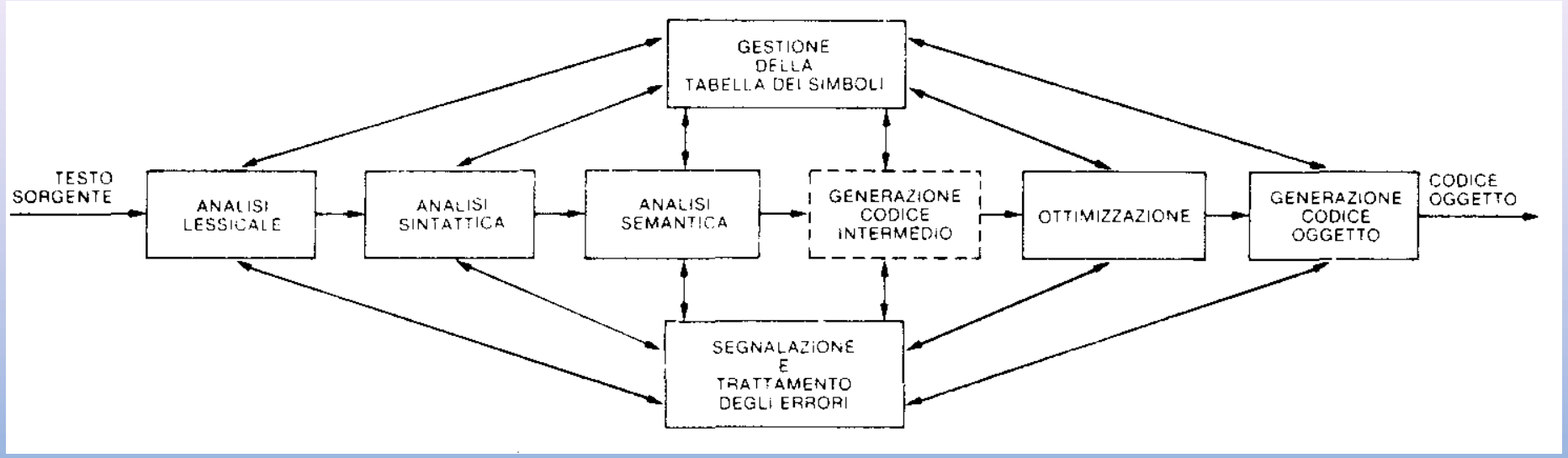
CRITERI GENERALI PER IL C.I. (CHIAMATA DI FUNZIONE)



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 assembly.

In questa fase è possibile prevedere eventuali azioni di ottimizzazione del codice.



GENERAZIONE DEL CODICE OGGETTO

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 assembly 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.

```
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 funzione).

```
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 tre 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)

Le scelte effettuate in queste tre fasi possono infuire sull'efficienza del codice target

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 pcode 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 una 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.

ALLOCAZIONE DEI REGISTRI

I registri sono gli elementi di memorizzazione più veloce, ma in genere non sono tanti. Si pongono quindi i seguenti problemi:

- 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 e Dipende dall'architettura

ORDINE DI VALUTAZIONE

La scelta di un particolare ordine con cui eseguire le istruzioni può far si che sia sufficiente l'uso di un numero inferiore di registri

La scelta del miglior ordine è un problema NP-completo

LINGUAGGIO TARGET

Se consideriamo 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

Comunemente, un 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 NEL C.I.

Molti generatori di codice partizionano le istruzioni del C.I in blocchi base che saranno sicuramente eseguiti in sequenza. Ciò agevola l'ottimizzazione del codice.

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
```

BLOCCHI BASE E DAG

Molte delle tecniche di ottimizzazione locale trasformano un blocco base in un nodo di un directed acyclic graph (DAG).

Una rappresentazione basata su DAG consente di effettuare varie trasformazioni nel codice mirate a migliorarne la qualità

- Eliminazione di sottoespressioni comuni
- Eliminazione del codice morto
- Riordinare istruzioni che non presentano dipendenze reciproche
- …

Ogni istruzione del blocco base viene tradotta in istruzioni nel codice target.

UN ESEMPIO

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`

In modo analogo si procede con le altre istruzioni

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.