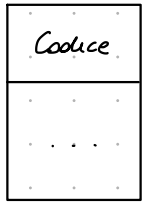
Generazione di codice

Analisi Lessicale — suddivido prog. in token
↓ lista di token
Analisi sintattica — analizza i token e produce l'albero
↓ AST
Analisi Semantica — scope e symbol table, Type Checking
↓ AST+
Generazione bytecode → Converte l'AST+ in codice oggetto

Quando vogliamo eseguire un programma:

- Inizialmente il SO ha il controllo
- Viene allocato spazio per il programma
- Il codice viene caricato in quello spazio
- L'OS salta all'entry point del programma



Assumiamo:

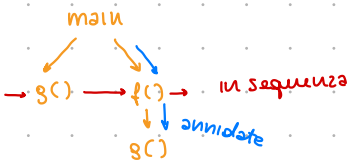
- esecuzione sequenziale
- dopo la chiamata di una procedura il controllo ritorna alla riga successiva

Scope e Lifetime

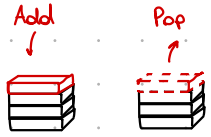
- Scope → concetto statico! (visibilità)
- Lifetime → concetto dinamico! → durata

Activation tree

Albero per rappresentare i lifetime delle funzioni

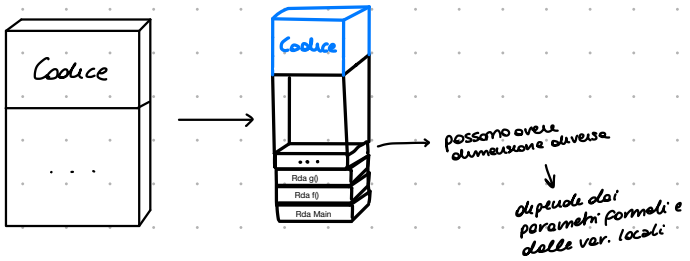


Essendoci funzioni annidate, la struttura dati più comoda è lo stack (pila lifo)



→ main
→ main g → g termina
→ main f
→ main f g

Le informazioni per gestire le chiamate di funzioni sono negli AR - Activation Record



- Quando f $\xrightarrow{\text{invoca}}$ g
  - ↳ f è sospesa fino alla fine di g

a g servono info. per sapere dove riprendere

Quindi sappiamo che un AR è composto da:

1. spazio per il valore di ritorno
2. parametri attuali
3. puntatore al precedente AR → chiamato Control Link
4. stato della macchina prima della chiamata
5. variabili locali
6. altri val. temporanei

in breve →

var. locali
indirizzo di ritorno
par. attuali
puntatore al prec. AR

- Il compilatore deve determinare a compile time il layout degli activation record e generare codice che accede correttamente ai campi degli AR

L'AR viene riempito sia dal chiamato che dal chiamante!

var. locali	} chiamato
indirizzo di ritorno	} chiamante
par. attuali	
puntatore al prec. AR	

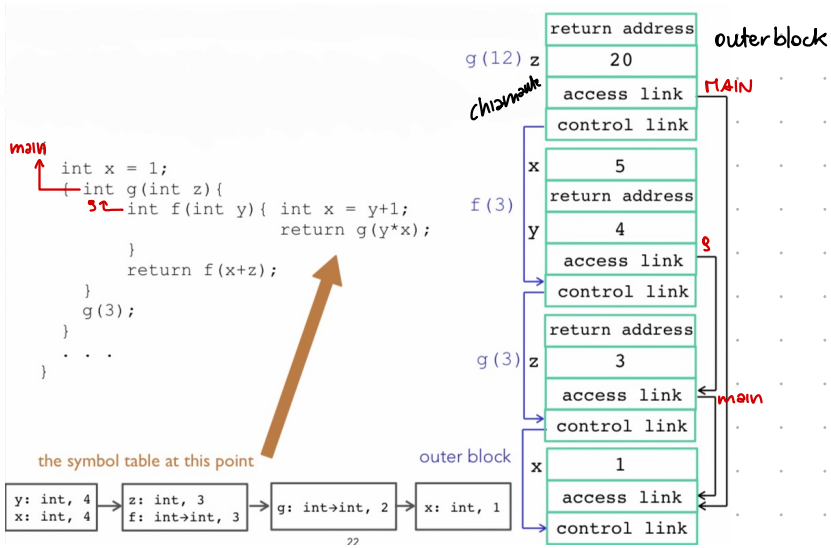
Variabili Globali

Le variabili globali devono fare riferimento allo stesso elemento in ogni punto del codice

↳ non possiamo metterle in un AR solo

Serve un puntatore per accedere alle variabili dichiarate in scope esterni

↳ Quindi un puntatore che punta all'AR più "vicino", chiamato Access Link

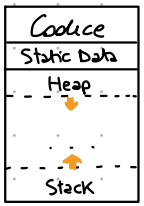


* Garbage Collector *

Quindi:

- La code Area contiene codice
- L'area statica contiene dati non indirizzi fissi (Es. var. globali)
- Lo stack contiene gli AR
- Lo heap contiene tutti gli altri dati

Siccome sia heap che stack crescono e non vogliamo collisioni, li mettiamo in zone opposte della memoria



Definiamo il nostro bytecode

- AO
- RA
- FP
- SP
- AL
- T1

- permette operazioni matematiche che usano registri per operandi e risultati

- add R1 R2 Sommo R1 R2 e metto val. sullo stack
- addi R1 n
- sub R1 R2
- subi R1 n
- storei R1 n n salvato in R1
- move R1 R2 il valore di R1 copiato in R2
- pushr R1 prendo il val. in R1 e copio sul top della pila
- popr R1 prendo il valore sul top della pila e salvo in R1

Per ogni espressione, vogliamo:

- calcolare il valore di e e salvarlo in AO (registro speciale chiamato accumulator)
- preservare lo SP e quindi il contenuto dello stack

Invariante principale!

Definiamo una funzione cgen (SymbolTable M, Node e) che genera il codice per e

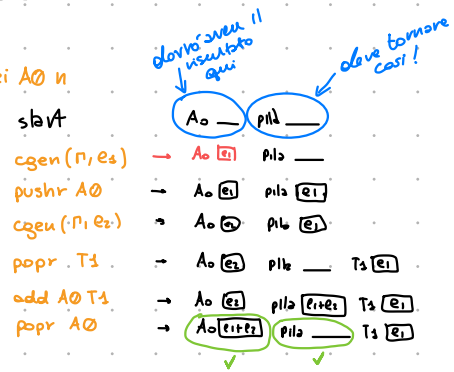
Vediamo alcune Code Gen:

- costante

  cgen (r, n) = storei A0 n

- espressione

  cgen (r, e1 + e2):



Ci serve un'operazione per il controllo di flusso:

- BEQ R1 R2 label — salto condizionato

  (Salta alla label se R1 = R2)

  → etichetta che ci serve per capire a che punto del codice saltare

- B label — salto incondizionato

con queste due operazioni possiamo generare il codice per l'if then else

Codice

```
cgen(r, if(e1 == e2) then e3 else e4) =
    label false_if = new label
    label true_if = new label
    label end_if = new label
    cgen(r, e1)
    pushr A0
    cgen(r, e2)
    popr T1
    beq A0 T1 true_if
    (false_if:)
    cgen(r, e4)
    b end_if
true_if:
    cgen(r, e3)
end_if:
```

dichiarazione delle label

senza questa alla prossima cgen perdiamo il valore di e1, essendo in A0

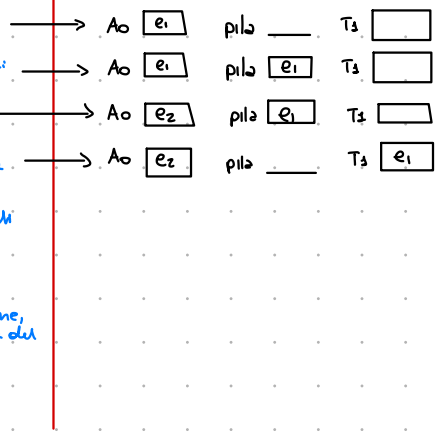
metto in T1 il valore di e1 che avevo lasciato sulla pila mi serve perché BEQ prende due registri

(in realtà non serve!)

Se sono uguali salta, altrimenti esegue la riga dopo (l'else)

salto incondizionato alla fine, altrimenti eseguirei il codice del then!

Stato della macchina

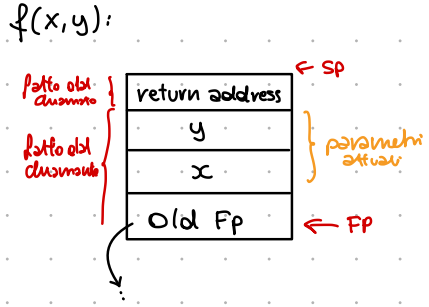


Il codice per le definizioni e le chiamate di funzione dipende dal layout dell'AR

- Il risultato è sempre in A0 → quindi non serve nell'AR
- Nell'AR ci sono i parametri attuali
- Al termine della funzione per l'invariante sappiamo che SP si troverà come si trovava prima della chiamata
- dobbiamo salvare il return address

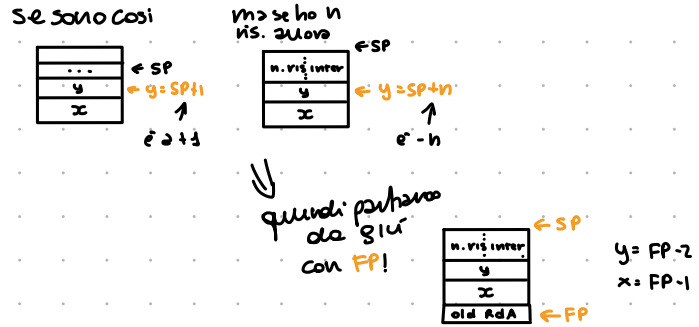
Dobbiamo quindi implementare una pila di AR

- Ogni AR dovrà avere il puntatore all'AR chiamante
- Il puntatore verrà salvato sul registro FP
- FP punterà alla posizione del primo parametro del chiamato (che è il valore del precedente AR)



- Non ci servono gli access link perché non abbiamo dichiarazioni annidate

- Se abbiamo dei risultati intermedi, questi fanno crescere lo stack e abbiamo problemi con l'offsett delle variabili a partire da SP!
  ↳ Soluzione: usiamo FP che punta alla prima var. →



Code Generation per la chiamata di funzione

bisogna preparare l'invocazione di funzione

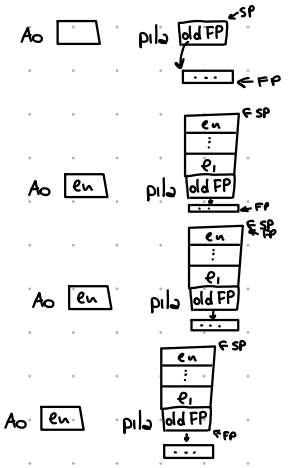
abbiamo una nuova op. JSUB label → salta alla label e salva in RA l'indirizzo della prossima istruzione

Codice

```
cgen(Γ, f(e1 ... en)) =
    pushr FP
    cgen(Γ, e1)
    pushr A0
      ⋮
    cgen(Γ, en)
    pushr A0

    move SP FP
    addi FP n+1
    Jsub lookup(Γ, f).label   ← salto alla procedura
```

Stato della macchina



Code Generation per la definizione di funzione

ci serve l'istruzione per jumpare al return address

- rsub RA

Codice

cgen (Γ, int f(int x,..., int xn) = e) =

lookup(Γ, f).label:

pushr RA

cgen (Γ, e)

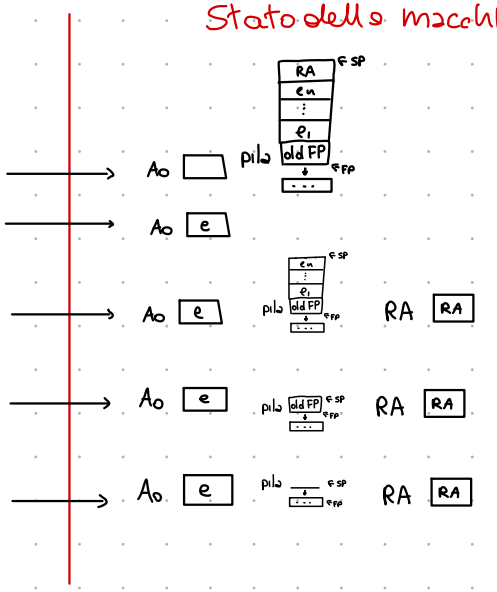
popr RA

addi SP SP n

popr FP

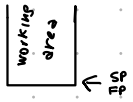
rsub RA → ho fixato tutto, Jumpa al return address

Stato della macchina



Code generation per programmi:

cgen (∅, D1; ...; Dn; E) = (funzioni)



storei SP max_val

storei FP max_val

cgen (Γ, e) → store ∅ ⊢ D1 .. Dn : Γ

halt

cgen (∅, D1; ...; Dn) → seq. di dich. di funz.

cgen (Γ, D; D') = cgen(Γ, D)
cgen(Γ', D')

espandiamo il linguaggio

↓

```
P    →  D ; P | E | S
D    →  T id(ARGS) = P
ARGS →  id, ARGS | id
E    →  int | id | if (E1 == E2) then E3 else E4
        | E1 + E2 | E1 - E2 | id(E1,...,En)
S    →  ( id = E ; | id(E1,...,En) ; )+
T    →  int | void
```

- solo consentite dichiarazioni annidate
- c'è l'assegnamento

- $x = e$

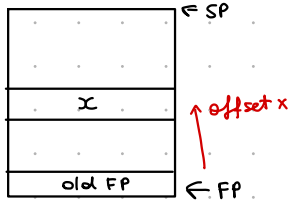
nuova istruzione

load R1 offset(R2)

↳ salvo il valore di R1 all'indirizzo R2+offset

↓ nel pratico si usa così

load A0 lookup(Γ, x).offset(FP)



Aggiorno il val della mia variabile x, posto nell'offset indicato nella symbol table.

(Nota: stiamo assumendo che x sia nel nostro AR)

Fino a
Slide 59
il restante è puramente teorico
e NON DOVREBBERO esserci esercizi

Extra:

Codice AND lazy
(se sto valutando e1 && e2 e e1 = false mi fermo)

```
cgen (Γ, e and e')
    exit = new label
    cgen(Γ, e)
    pushr T0 0
    Beq A0 T0 exit
    [ cgen(Γ, e') ← Verifica solo e'
exit:
```

Codice While

```
CGen (Γ, while (e1) { e2 }) =
start:
    cgen(Γ, e1)          new label: false
    pushr T0 0           new label: start
    BEQ A0 T0 false
    [ push A0
    [ cgen (Γ, e2)
      Branch START
false:  ↑ incondiz.
```

Codice For (Python)

```
Cgen (Γ, FOR I in range(E, E') : S
loop = new label
exit = new label
    cgen (Γ, e)        I = A0
    LOAD A0 → Γ(i).offset(FP)
    Cgen (Γ, e')
    Push A0
loop
    STORE T1 ← Γ(I).offset(FP)
    BGTE  T1  A0  EXIT
    Cgen (Γ, S)        ← Controllo
    Cgen (Γ, i+1)
    LOAD A0 → Γ(i).offset(FP)
    POPRA0
    PUSH A0   } incremento il pil
    BLOOP
EXIT:
    Pop
```

Codice Do While

```
codegen (Γ, do S while E)
    loop = new label
loop codegen(Γ, S)
    codegen (Γ, e)
    pushr T0 1
    BEQ A0 T0 loop
```

Esempio di gen. di codice da esame (19/12/19)

Esercizio 3 (8 punti)

1. Definire la funzione `code_gen` per il termine `do S while E` che esegue S, quindi controlla E e se essa è vera riesegue S, altrimenti l'esecuzione termina.
2. Come verifica, si generi il codice di

```
do do ( x:= x+1 ; y:= y+x ) while (x>y) while (y<x+z)
```

dove le variabili `x`, `y` e `z` si trovano ad offset +4 e +8 e +12 del frame pointer FP.

Avendo il do while come definito sopra:

codegen(Γ, do do (x := x+1 ; y = y+x) while(x>y) while (y < x + z))

loop:

codegen (Γ, do (x := x+1 ; y = y+x) while(x>y))

codegen(Γ, y < x + z)

pushir T0 1

BEQ A0 T0 loop

codegen(Γ, y)

push A0

codegen (Γ, x+z)

pop T0

LT T0 A0 → metto in A0 (y x+z) 1 o 0

codegen(Γ,x)

push A0

codegen (Γ, z) → store A0 Γ(z).offset(Γ)

pop T0

ADD A0 T0

pop A0

codegen (Γ, do (x := x+1 ; y = y+x) while(x>y):

loop:

codegen(Γ, x = x+1 ; y = y + x)

codegen (Γ, x > y)

pushir T0 1

BEQ A0 T0 loop

codegen(Γ, x) → ...store

push A0

codegen (Γ,y) → ...store.

pop T0

GT T0 A0 → metto in A0 1 o 0

codegen (Γ, x = x+1 ;)

codegen (Γ; y = y+x;)

cgen (Γ, x+1):

store A0 Γ(x).offset(4)

push A0

cgen(Γ, 1) → A0 1

pop T0

add A0 T0

pop A0

LOAD A0 Γ(x).offset(4)

cgen (Γ, y+x):

cgen(Γ,y) → store A0 Γ(y).offset(8)

push A0

cgen(Γ, x) → store A0 Γ(x).offset(4)

pop T0

add A0 T0

pop A0

LOAD A0 Γ(y).offset(8)