Vincoli Sintattici Contestuali

(Semantica Statica)

Esempi:

- una variabile in uso deve prima essere dichiarata
- Compatibilità di tipo in un assegnamento

  x := e   "x" e l'espressione "e" devono avere lo stesso tipo
- il numero (e il tipo) dei parametri attuali di una chiamata di procedura deve essere uguale al numero (e al tipo) dei parametri formali della dichiarazione

Sono vincoli sintattici, ma non esprimibili per mezzo di grammatiche libere (o BNF), perché queste non sono in grado di descrivere vincoli che dipendono dal contesto (ad es: dalle dichiarazioni)

Soluzioni? Ci sono due alternative:

(1) Usare strumenti più potenti, come le grammatiche dipendenti dal contesto

Poco pratico: complessità del problema "w∈L(G)?" (con G contestuale) esponenziale nella lunghezza di w e nella taglia di G

(2) Usare controlli "ad hoc"
Nella costruzione di un compilatore, la fase di "Analisi Semantica" fa questi controlli

Sintassi o Semantica?

I vincoli sintattici contestuali "appartengono" alla sintassi.

Ma tradizionalmente, nel gergo dei LP, si intende:

- Sintassi: quello che si descrive in BNF (grammatiche libere)
- Semantica: tutto il resto ...

I vincoli contestuali sono dunque vincoli "semantici", detti di semantica statica, cioè vincoli che possono essere verificati ispezionando il codice del programma, senza mandarlo in esecuzione

Il compilatore delega questi controlli di semantica statica (o sintassi contestuale) alla 3ª fase, denominata

Analisi Semantica

Semantica Statica vs. Semantica Dinamica

Per semantica <u>statica</u> (o sintassi contestuale) si intende l'insieme di quei controlli che possono essere fatti sul testo del programma, senza eseguirlo. Ad esempio, <u>Type checking</u>

```
int A;
bool B;
A := B      ← errore di tipo
```

Per semantica <u>dinamica</u> si intende una rappresentazione formale dell'esecuzione del programma, la quale può mostrare errori "dinamici", cioè durante l'esecuzione

```
P=  int A, B;
    read (A);
    B:= 10/A;
      ⋮
```

Se l'utente fornisce in input il valore 0 per A, allora la divisione 10/A dà errore in esecuzione.

In questo caso, non possiamo staticamente sapere se l'errore si verificherà perché l'occorrere dell'errore dipende dall'input che l'utente fornirà quando il programma verrà eseguito.

Semantica Dinamica

- Fornire un modello matematico che descriva, indipendentemente dall'architettura su cui il programma viene eseguito, il "comportamento" del programma.
- L'implementazione del linguaggio (in cui è scritto il programma) deve rispettare questa "specifica" del linguaggio

Esempio

$P = \quad x := x + 1$

store
σ = insieme di associazioni tra nomi e valori

modello "grafo"

$$\langle x := x+1 , \sigma \rangle \longrightarrow \sigma \left[{}^{\sigma(x)+1}/_{x} \right]$$

valuto il comando utilizzando uno store σ

il risultato è uno store "aggiornato" in cui ad x è associato il valore $\sigma(x) + 1$

(Astratto e indipendente dall'architettura)

A chi serve la semantica dinamica

- **Al programmatore** : ANALISI DEL PROGRAMMA
  - deve sapere esattamente cosa debba fare il suo programma
  - deve poter dimostrare proprietà del suo programma (ad es: "termina sempre per ogni possibile input?")

- **Al progettista del linguaggio** :
  - strumento di specifica del linguaggio
  - deve poter dimostrare proprietà del linguaggio (ad es: "è Turing-completo?")

- **All'implementatore del linguaggio**
  - riferimento per dimostrare la correttezza dell'implementazione

Ricorda: un compilatore è corretto quando preserva la semantica (dinamica!), quindi per dimostrare che un compilatore è corretto serve avere una semantica per il ling. sorgente e per il ling. oggetto.

Come si fa a definire la semantica?

Due tecniche principali:

- **Operazionale**: (macchina astratta a stati e transizioni)
  - si costruisce una specie di automa che, passo a passo, mostra l'effetto della esecuzione delle varie istruzioni
  - Enfasi su come si calcola

- **Denotazionale**: si associa ad ogni programma sequenziale una funzione da input ad output (incluse strutture ausiliarie dette ambiente e memoria)

  Enfasi su cosa si calcola

  (vengono nascosti i passi intermedi del calcolo)

Vedremo l'uso della semantica operazionale per un semplice linguaggio imperativo

Altri aspetti della descrizione di un linguaggio

PRAGMATICA: insieme di "regole"/consigli sul modo in cui è meglio usare le istruzioni a disposizione
("buone maniere" per ling. naturali, "stile di programmazione" per ling. artificiali)

Esempi:

- evitare le istruzioni di salto (GOTO) quando possibile (altrimenti i programmi diventano presto incomprensibili)
- usare le variabili di controllo del for solo a quello scopo (per evitare side-effect pericolosi)
- scelta della modalità più appropriata di passaggio di parametri ad una funzione (ad es: mai usare il passaggio per valore per strutture dati grandissime - spreco di memoria)
- scelta tra iterazione determinata (FOR - da preferirsi, quando possibile) e indeterminata (while/repeat)

Infine... IMPLEMENTAZIONE

ovvero scrivere un compilatore (o un interprete) per una macchina ospite già realizzata, costruendo così una macchina astratta per il linguaggio

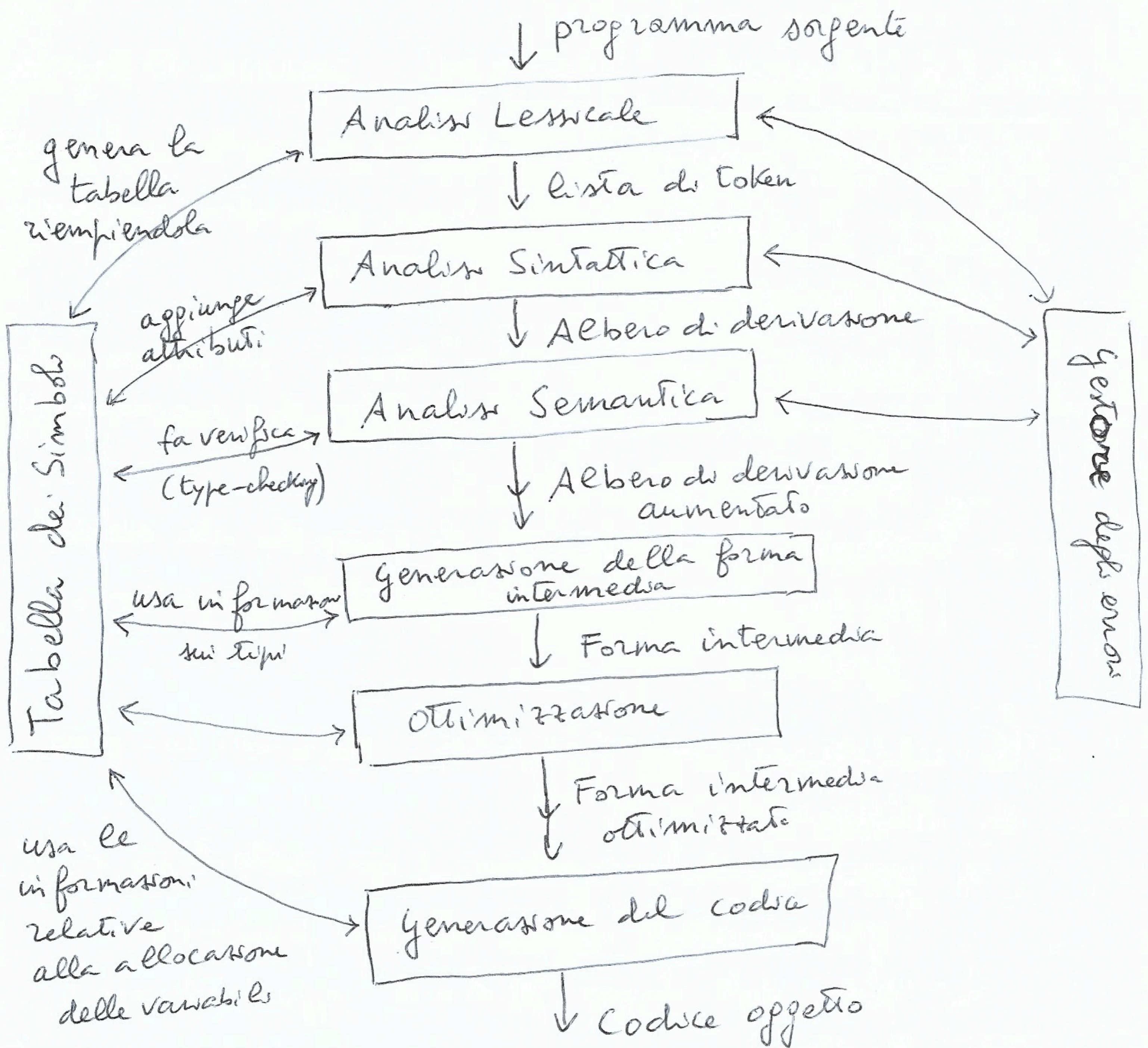
- correttezza dell'implementazione?

  Bisogna dimostrare che preserva la semantica, ovvero che il programma sorgente e quello oggetto calcolano la stessa funzione (per i linguaggi sequenziali)

- costo dell'implementazione?

  - è il compilatore in grado di produrre codice efficiente?
  - certi costrutti linguistici, scelti dal progettista del linguaggio, sono troppo onerosi in pratica?

Struttura di un Compilatore



Ogni errore rilevato (nelle prime 3 fasi) non blocca il compilatore, ma genera un opportuno messaggio d'errore

Fasi Principali della Compilazione

Analisi Lessicale (Scanner): spezza il programma sorgente nei componenti sintattici primitivi, chiamati tokens (identificatori, numeri, operatori, parentesi, parole riservate, ...)

- controlla solo che il "lessico" sia ammissibile (operatori inesistenti, identificatori non ammessi, simboli non previsti, ..)
- riempie parzialmente la tabella dei simboli (per gli identificatori di variabili, procedure, funzioni...)

Per realizzare uno Scanner, avremo bisogno di studiare:

- grammatiche regolari (un sotto-tipo di grammatiche libere le cui produzioni sono del tipo $A \to aB$ oppure $A \to a$)
- espressioni regolari (un formalismo usato per descrivere i linguaggi generati da grammatiche regolari)
- automi a stati finiti (NFA – DFA): uno strumento che permette di riconoscere i linguaggi "regolari"

Analisi Sintattica (Parser)

A partire dalla lista di tokens, generata dallo scanner, il parser produce l'albero di derivazione del programma, riconoscendo se le frasi sono sintatticamente corrette.

Ad esempio, controlla che

- le parentesi usate in espressioni aritmetiche siano bilanciate

  (a (+ b)))   2 errori! Quali?

- che i comandi siano composti secondo le regole grammaticali

  if x = 5 then then x := 3   errore!

Per realizzare un Parser, avremo bisogno di:

- grammatiche libere da contesto
- automi a pila (pushdown automata - PDA) soprattutto nella loro versione deterministica (DPDA)

Analisi Semantica

Esegue dei controlli di semantica statica (ovvero sintattici contestuali) per rilevare eventuali errori "semantici"

- arricchisce l'albero di derivazione generato dal Parser con informazioni sui tipi
- verifica i tipi negli assegnamenti, parametri attuali vs. formali, dichiarazione e uso di variabili, ...
- genera eventualmente opportuni msg d'errore.

Generazione della Forma Intermedia

Genera codice scritto in un ling. intermedio, indipendente dall'architettura, facilmente traducibile nel linguaggio macchina di varie macchine diverse.

- easy to produce — easy to translate

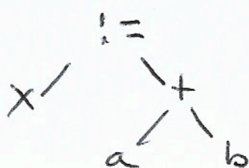
- utilizza operazioni molto semplici, tipicamente "three-address code"

  del tipo $x := y + z$

- nel generare codice intermedio, si segue la struttura dell'albero sintattico, ricavato dall'albero di derivazione

$X := a + b$ $\qquad$ temp1 := a + b

$\qquad\qquad\qquad$ x := temp1



Ottimizzazione

Si effettuano ottimizzazione sul codice intermedio per renderlo più efficiente:

- rimozione di codice inutile (dead code)
- espansione in linea di chiamate di funzioni
- fattorizzazione di sottoespressioni
- mettere fuori dai cicli sottoespressioni che non variano

Alla fine si ottiene un codice intermedio ottimizzato.

Generazione del codice

Viene generato codice per una specifica architettura (include anche l'assegnazione dei registri e ottimizzazioni specifiche per quell'architettura)

- quali variabili conviene tenere nei registri del processore?

Tabella dei simboli

Memorizza le informazioni sui nomi presenti nel programma (identificatori di variabili, funzioni, procedure, ...)

- ad es. per le matrici, mette come attributo la dimensione e il tipo dei suoi elementi

Un esempio istruttivo

Prendiamo un piccolo frammento di un programma PASCAL

var position, initial, rate : real

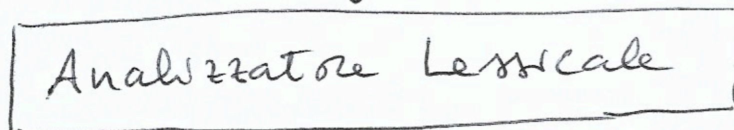
position := initial + rate * 60

e vediamo come viene trasformata l'istruzione

position := initial + rate * 60

nelle varie fasi del compilatore

position := initial + rate * 60



<id , 1> <:=> <id , 2> <op,+> <id , 3> <op,*> <const,60>

più brevemente

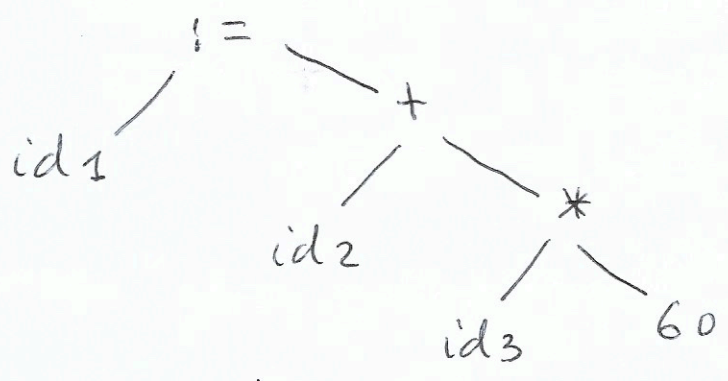
$$id_1 := id_2 + id_3 * 60$$

dove 1, 2, e 3 sono indirizzi nella Tabella dei Simboli

		Attributo
1	position	real
2	initial	real
3	rate	real
	⋮	

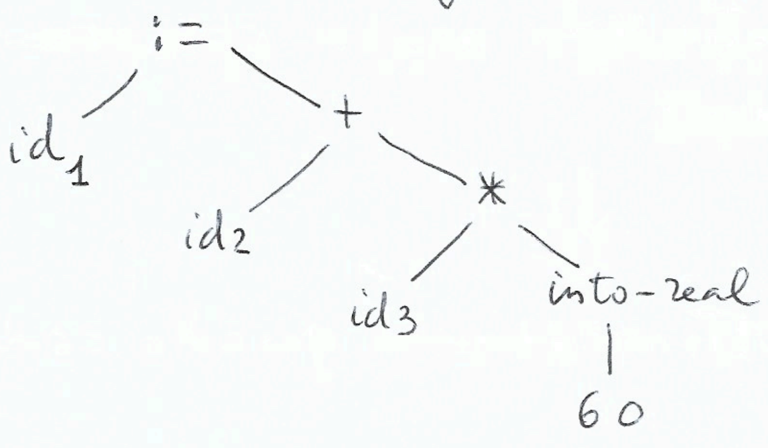
$id_1 := id_2 + id_3 * 60$

Analizzatore Sintattico



Albero Sintattico

Analizzatore Semantico



Poiché tutti gli identificatori sono di tipo "real", allora tutte le operazioni saranno su "real" ed anche 60 deve essere convertito in "real"

↓

Generazione del codice intermedio

↓

```
temp1 := into-real(60)
temp2 := id3 * temp1
temp3 := id2 * temp2
id1 := temp3
```

- usiamo "three-address code" come linguaggio intermedio
- la generazione del codice segue l'albero sintattico bottom-up
- un temporaneo viene generato per memorizzare il risultato di ogni operazione
- ci sono istruzioni senza operazioni "vere" (l'ultima)

Ottimizziamo questo codice!

↓

Ottimizzazione

↓

$temp_1 := id_3 * 60.0$

$id_1 := id_2 * Temp_1$

conversione fatta dal compilatore (non necessaria la funzione int-to-real)

↓

Generazione del Codice

↓

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Codice Assembler (o codice macchina rilocabile)

la "F" sta per "floating-point arithmetics"

deve essere trattata come costante (e non come indirizzo di memoria)