che cos'è un compilatore?

- è una traduzione da un linguaggio, detto *sorgente*, in un altro, detto *oggetto*
  - **di solito**, il linguaggio sorgente è un linguaggio ad alto livello, il linguaggio oggetto è un linguaggio macchina
  - **eccezione**: traduzione da codice macchina a codice macchina, manipolazioni di files XML
- i compilatori inoltre controllano che

  i programmi siano **sintatticamente corretti**

  i programmi non contengano **errori "semantici"** quali codice morto, dichiarazioni di variabili non utilizzate (cf. *warnings*)

esempi:

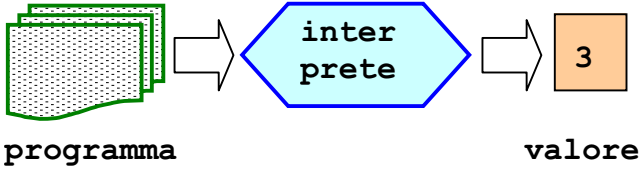
compilazioni di espressioni regolari in emacs, nei motori di ricerca, etc.
compilazione dal tex al postscript o all'HTML
compilazioni di linguaggi di programmazione

compilazione di linguaggi di programmazione

traduzione di programmi in un qualche linguaggio di programmazione in una sequenza di istruzioni direttamente eseguibile da un calcolatore (in linguaggio macchina)

compilatori ed interpreti

un compilatore trasforma programmi in programmi



un interprete valuta programmi

la compilazione è un problema "difficile": principio per semplificarla

dividere il problema in una sequenza di trasformazioni semplici

introdurre rappresentazioni intermedie di codici in punti importanti della sequenza

osservazione: in acuni casi il problema è semplice, e non richiede alcuna catena di trasformazioni:

la compilazione di espressioni regolari in automi è pressochè diretta

il problema (attraverso un semplice esempio)

```
x := a * 2 + b * (x * 3)
```

che cosa è sintatticamente la frase di sopra?

è un comando valido?

come si determina il significato?

- divisione in parole
- conversione di parole in comandi
- interpretazione del significato dei comandi

} **trasformazioni successive**

divisione in parole: Analisi Lessicale

```
INPUT:   x := a *   2 + b * (x * 3)
OUTPUT:  id<x> assign id<a> times int<2> plus id<b> times lpar
         id<x> times int<3> rpar
```

fasi:

raggruppare i caratteri della stringa di input in token

eliminare caratteri superflui (blanks, new-lines, commenti, etc.)

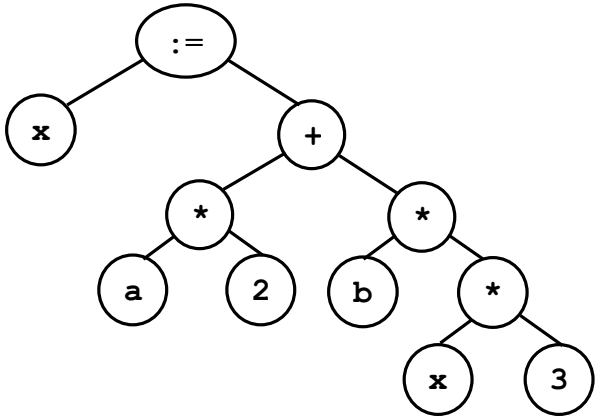
usare le espressioni regolari per la specifica e gli DFA per l'implementazione

flex

conversione delle parole in comandi: Analisi Sintattica

```
INPUT:   id<x> assign id<a> times int<2> plus id<b> times lpar
         id<x> times int<3> rpar
```

OUTPUT: alberi sintattici



fasi:

raggruppare i token della stringa di input in comandi

eliminare token superflui (parentesi, etc.)

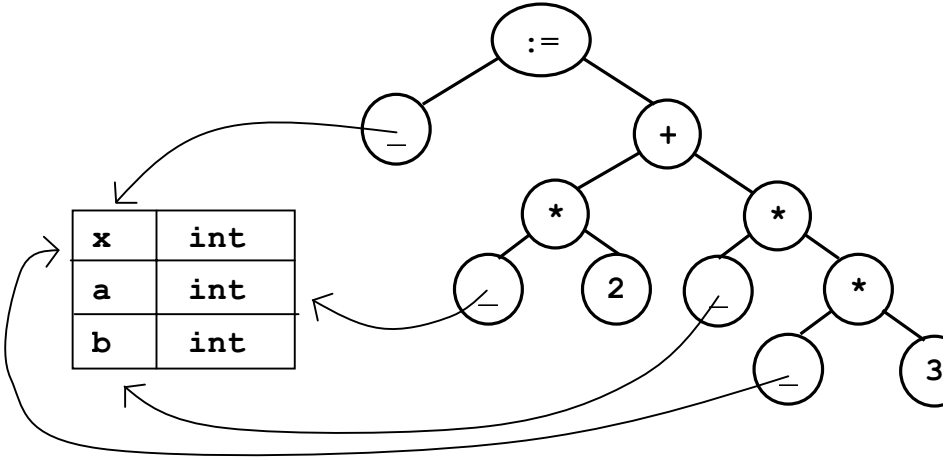
usare i linguaggi context-free per la specifica e i push-down automata per l'implementazione

bison

significato dei comandi: Analisi Semantica

INPUT: **alberi sintattici**

OUTPUT: **alberi sintattici con tabella dei simboli**



fasi:

- connette le def di variabili al loro uso
- verifica che ogni espressione ha il tipo corretto
- controlla se è legale leggere/scrivere **x**
- usa grammatiche con attributi, symbol tables, etc.

traduzione: Generazione di codice intermedio

INPUT: **alberi sintattici con tabella dei simboli**

OUTPUT: **rappresentazione intermedia più adatta per generare codice macchina (bytecode)**

fasi:

definizione della sintassi astratta intermedia

diversi tipi di rappresentazioni intermedie (gerarchica, lineare, ad albero, a triple)

generazione di codice macchina concreto: Selezione delle Istruzioni

INPUT: rappresentazione intermedia

OUTPUT: codice macchina targeted

fasi:

- implementazione della rappresentazione intermedia in un dato linguaggio assembler
- quali istruzioni utilizzare? [esempio: `mul` oppure `shift-left`?]
- quali modalità per accedere agli operandi? [indirizzamenti, costanti]
- tipi di salti
- utilizzare grammatiche ad alberi e programmazione dinamica

```
r1    load    M[fp+x]
r2    loadi   3
r3    mul     r1, r2
r4    load    M[fp+b]
r5    mul     r3, r4
r6    load    M[fp+a]
r7    sl      r6, 1
r8    add     r6, r5
      store M[fp+x]    r8
```