Fondamenti di Informatica

AA 2018/2019

Eng. Ph.D. Michela Paolucci

DISIT Lab http://www.disit.dinfo.unifi.it
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-2758515, fax: +39-055-2758570
michela.paolucci@unifi.it

DISIT Lab
http://www.disit.dinfo.unifi.it
Department of Information Engineering,
DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy

Libro di testo

- Enrico Vicario
- Fondamenti di programmazione. Linguaggio C, strutture dati, algoritmi elementari, C++
- Editore: Esculapio
- ISBN: 9788874884582
- NOTA: *Queste slide integrano e non sostituiscono quanto scritto sul libro di testo.*

Pagina del Corso

http://www.disit.org/drupal/?q=node/7020

Qui trovate:

- AVVISI
- Slide del corso
- Approfondimenti

Orario del Corso e Ricevimento

Insegnamento: FONDAMENTI DI INFORMATICA (A-L) (FonDiInf(A-L))

Ingegneria FIRENZE - A.A. 2018/2019 - 2° periodo

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
08:15						
09:15						
10:15						
11:15	(Auditorium A - C.D.M.)					
12:15	(Auditorium A - C.D.M.)					
14:00					(Auditorium A - C.D.M.)	
15:00					(Auditorium A - C.D.M.)	
16:00					(Auditorium A - C.D.M.)	
17:00						
18:00						

Docenti: PAOLUCCI MICHELA

Legenda: C.D.M.: Centro Didattico Morgagni

docente via email:

- michela.paolucci@unifi.it

Outline

- Definizione di un linguaggio
  - Sintassi di un linguaggio
  - Grammatica
  - Albero sintattico
  - Il metalinguaggio BNF
  - Semantica e Sintassi

Definizione di un linguaggio

- La definizione di un linguaggio consiste di:
  - Sintassi:
    - determina e classifica cosa può apparire in una espressione legale del linguaggio
    - rappresenta le regole che permettono di scrivere in maniera corretta le ‘frasi’ del linguaggio
  - Semantica:
    - Definisce il significato associato a ciascuna classe di espressioni legali
    - Specifica il significato delle frasi, distinguendo tra quelle che hanno un significato e quelle che, invece, non ne hanno.

Sintassi di un linguaggio (1)

- Un VOCABOLARIO è un insieme di simboli V
- L'UNIVERSO LINGUISTICO di un vocabolario è denotato con V* ed è costituito dall'insieme di tutte le sequenze finite di simboli in V, inclusa la parola nulla λ
- Un LINGUAGGIO L sul VOCABOLARIO V è un sottoinsieme di V*
- Esempio, supponiamo di voler definire un linguaggio sul vocabolario V={p,c}, allora si ha:
  - L = {λ, p, c, cp, cc, pc, pp, ccc, ccp, …} incluso in V*

Sintassi di un linguaggio (2)

- Il problema di definire la sintassi di un Linguaggio sta nel definire i limiti del sottoinsieme di V*

- Un linguaggio ha le seguenti caratteristiche:
  - E' composto da infinite espressioni diverse
  - Le espressioni devono essere definite attraverso un insieme di regole che ammetta una rappresentazione finita
  - L'approccio più comune per descrivere un linguaggio è quello di usare una GRAMMATICA

Grammatica (1)

- Una GRAMMATICA è una quadrupla G<V,N,S,P>, con:
  - V = Vocabolario, insieme i cui elementi sono detti simboli terminali
  - N = Insieme disgiunto da V, i cui elementi sono detti Categorie Sintattiche
  - S = Simbolo iniziale, è un elemento di N
  - P = Produzione, è una relazione di N su (N U V)*, ovvero P è un sottoinsieme del prodotto Cartesiano N x (N U V)*

- NOTA1: (N U V)*, ovvero l'Universo Linguistico di (N U V), è l'insieme di tutte le sequenze finite di elementi appartenenti a (N U V)
- NOTA2:Prodotto Cartesiano. Dati due insiemi A e B, si definisce Prodotto Cartesiano, L'insieme delle coppie ordinate: <a,b>, con a ∈ A e b ∈ B

Grammatica - Produzione

- Una Produzione, è una relazione di N su (N U V)*, ovvero P è un sottoinsieme del prodotto Cartesiano N x (N U V)*

- Ovvero P è un insieme di coppie <n, η>, dove n è un elemento di N e η è una sequenza di elementi che appartengono a N o a V:
    - $n \in N$
    - $\eta \in (N \cup V)$

Grammatica – Semantica (1)

- Per poter comprendere fino in fondo come una GRAMMATICA serva a definire un linguaggio, è necessario definire non solo la **sintassi** (descritta fino ad ora) ma anche la **semantica**
- La semantica serve per definire le regole attraverso le quali la quadrupla che rappresenta la grammatica G definisce un linguaggio Lg
- E' necessario definire le regole semantiche associate a ciascuna categoria sintattica N
- La regola attraverso cui una grammatica definisce un linguaggio è descritta grazie a tre definizioni:
  - Derivazione diretta
  - Derivazione indiretta
  - Linguaggio definito da una grammatica

Grammatica – Semantica (2)

- **Derivazione Diretta:**
  - Siano pre e post elementi di (N U V)*
  - Siano:
    - a_0 elemento di N, con N categoria sintattica
    - α_1 elemento di (N U V)*, ovvero una sequenza di elementi appartenenti o a N o a V
- Si dice che: pre α_1 post deriva direttamente da pre a_0 post, e si scrive:

  pre a_0 post $\xrightarrow{1}$ pre α_1 post SE $< a_0, \alpha_1 > \in P$

NOTE:

- (N U V)* è l'Universo Linguistico di (N U V), ovvero l'insieme di tutte le sequenze finite di elementi appartenenti a (N U V)
- P = è una relazione di N su (N U V)*, ovvero P è un sottoinsieme del prodotto Cartesiano N x (N U V)*
- Prodotto Cartesiano, L'insieme delle coppie ordinate: <a,b>, con a ∈ A e b ∈ B

Grammatica – Semantica (3)

- **Derivazione Indiretta:**
  - Siano:
    - $\alpha_0 \in (N \cup V)^*$
    - $\alpha_k \in (N \cup V)^*$
  - Si dice che α_k deriva da α_0, e si scrive:
    - $\alpha_0 \rightarrow \alpha_k$ (si legge "α_k deriva da α_0") SE esiste una sequenza di elementi $\alpha_k \in (N \cup V)^*$ tali che:
      - Per ogni $k \in [1,K]$ $\quad \alpha_{k-1} \xrightarrow{1} \alpha_k$
- Esempio:
  - K=3, si dice che $\alpha_0 \rightarrow \alpha_3$ SE e solo SE:
    - $\alpha_0 \xrightarrow{1} \alpha_1$; $\alpha_1 \xrightarrow{1} \alpha_2$; $\alpha_2 \xrightarrow{1} \alpha_3$

NOTA: $(N \cup V)^*$ è l'insieme di tutte le sequenze finite di elementi appartenenti a $(N \cup V)$

Grammatica – Esempio (1)

- Si consideri il linguaggio definito dalle espressioni aritmetiche che combinano le variabili var di un insieme attraverso gli operatori +, -, *, / e le parentesi (,)
- Inoltre si noti che il linguaggio deve:
  - Includere l'espressione: a + b * (c – d)
  - Escludere l'espressione: a + b - * c
- SVOLGIMENTO:
- **Vocabolario**, (ovvero i simboli elementari che possono comparire in una espressione): V = var U {+, *, -, /, (,)}
- **Simbolo iniziale** (ovvero la categoria di cui il linguaggio definisce le espressioni legali): S = expr
- **Produzioni**, permettono di 'ridurre' il simbolo iniziale in simboli terminali passando attraverso un insieme di categorie sintattiche N

- N = Insieme disgiunto da V, i cui elementi sono detti Categorie Sintattiche
- S = Simbolo iniziale, è un elemento di N

Grammatica – Esempio (2)

- **Produzioni**, permettono di 'ridurre' il simbolo iniziale in simboli terminali passando attraverso un insieme di categorie sintattiche N

- Definiamo allora le seguenti produzioni per il caso in esame:
  - p1: expr → var , (var deriva da expr) “qualunque variabile var è una espressione legale”
  - p2: expr → (expr) , “una espressione legale tra parentesi è essa stessa una espressione legale”
  - p3: expr → expr **op** expr, “la combinazione di due espressioni legali attraverso un operatore **op** è essa stessa una espressione legale”
- NOTA: grazie a p3 abbiamo introdotto la categoria sintattica **op**, che rappresenta i simboli +, -, *, / e che implica la definizione di quattro ulteriori Produzioni

Grammatica – Esempio (3)

- **Produzioni**, [...]
  - p3: expr ⟶ expr **op** expr
- Grazie a p3 abbiamo introdotto la categoria sintattica **op**, che rappresenta i simboli +, -, *, / e che implica la definizione di quattro ulteriori Produzioni:
  - p4: op ⟶ +
  - P5: op ⟶ -
  - P6: op ⟶ *
  - P7: op ⟶ /
- Le Produzioni della Grammatica formano quindi il seguente insieme:
  - P = {expr⟶var; expr⟶(expr); expr⟶expr **op** expr; op⟶+; op⟶-; op⟶*;op⟶/, var ⟶ ... } //ci sono i puntini perché si sono omesse le regole sintattiche per formare gli identificativi legali per un elemento di var

Grammatica – Esempio (4)

- **Categorie Sintattiche**: N = {expr; op; var}
- **VERIFICA:** Verificare che l'espressione a+b*(c-d) è una espressione legale nella Grammatica appena definita
- Supponiamo: a,b,c,d simboli in var, allora l'espressione a+b*(c-d) si scrive:
  - var + var * (var-var)
- Applichiamo p1 (expr ⟶ var):
  - var + var * (var-var) ⟵ expr + expr*(expr - expr)
- Applichiamo p4, p5, p6 (operatori):
  - expr + expr * (expr – expr) ⟵ expr op expr op (expr op expr)
- Applichiamo p3 (expr ⟶ expr **op** expr) e p2 (expr ⟶ (expr)) :
  - expr op expr op (expr op expr) $\overset{p3}{\longleftarrow}$ expr op expr op (expr)

  $\overset{p2}{\longleftarrow}$ expr op expr op expr $\overset{p3}{\longleftarrow}$ expr op expr $\overset{p3}{\longleftarrow}$ expr
- Concatenando, si ottiene a+b*(c-d) ⟵ expr, ovvero "a+b*(c-d) è una espressione legale"

- p1: expr ⟶ var
- p2: expr ⟶ (expr)
- p3: expr ⟶ expr op expr

- p4: op ⟶ +
- P5: op ⟶ -
- P6: op ⟶ *
- P7: op ⟶ /

Grammatica – Esempio (5)

VERIFICA: Verificare che l'espressione a+b*-c NON è una espressione legale nella Grammatica appena definita

Supponiamo: a,b,c,d simboli in var, allora l'espressione a+b*-c si scrive:

- var+var*-var

Applichiamo p1 (expr ⟶ var):

- var+var*(var-var) ⟵ expr+expr*-expr

- p1: expr ⟶ var
- p2: expr ⟶ (expr)
- p3: expr ⟶ expr op expr

- p4: op ⟶ +
- P5: op ⟶ -
- P6: op ⟶ *
- P7: op ⟶ /

Non si può estendere oltre attraverso l'applicazione delle produzioni definite nella grammatica in esame

In questo caso si dice che "La riduzione NON si è chiusa" e quindi NON è vero che "a+b*-c deriva dal simbolo inziale expr", ovvero "a+b*-c NON è una espressione legale per la grammatica definita"

Albero sintattico (1)

- Un ALBERO SINTATTICO è il processo con il quale una sequenza di simboli nel vocabolario V viene ridotta al simbolo iniziale S della grammatica G di un linguaggio L_G
- Un ALBERO è un insieme di nodi sui quali è definita una relazione di successione nella quale:
  - Ciascun nodo ha un predecessore
  - Ciascun nodo ha un numero non limitato di successori
  - Un nodo speciale, detto radice, non ha alcun predecessore
- Applicando la definizione di Albero, all'ALBERO SINTATTICO si ha che, in questo caso:
  - Nodi: rappresentano i simboli elementari e le produzioni
  - Archi: rappresentano le relazioni di derivazione diretta tra i nodi in base alle produzioni della grammatica

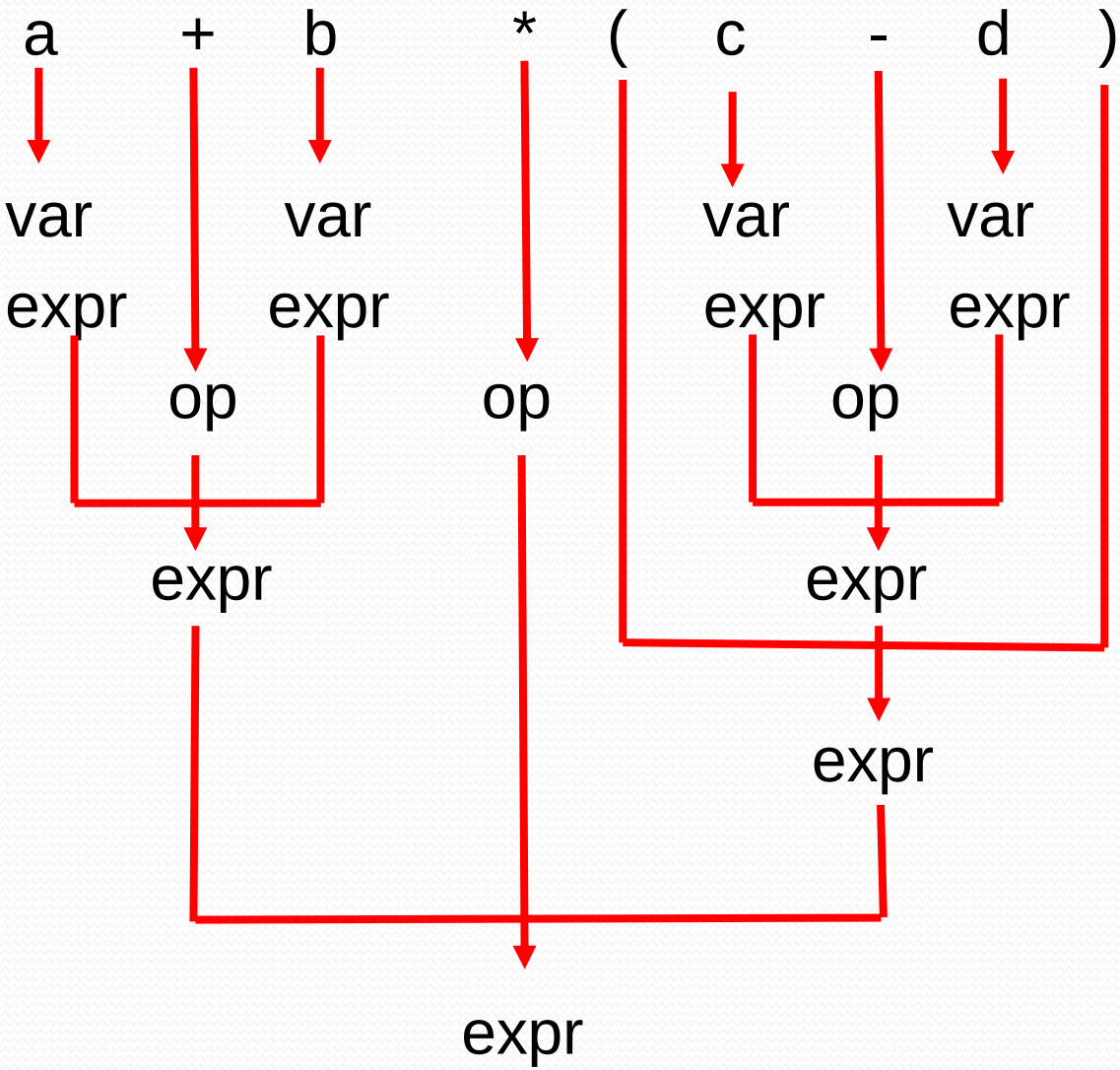
Albero sintattico (2)

- Assegnata una sequenza di simboli nel vocabolario, SE sulla sequenza può essere costruito un albero che si chiude su una radice, ALLORA la sequenza è una ISTANZA della categoria sintattica associata alla radice

Albero sintattico – Esempi (1)

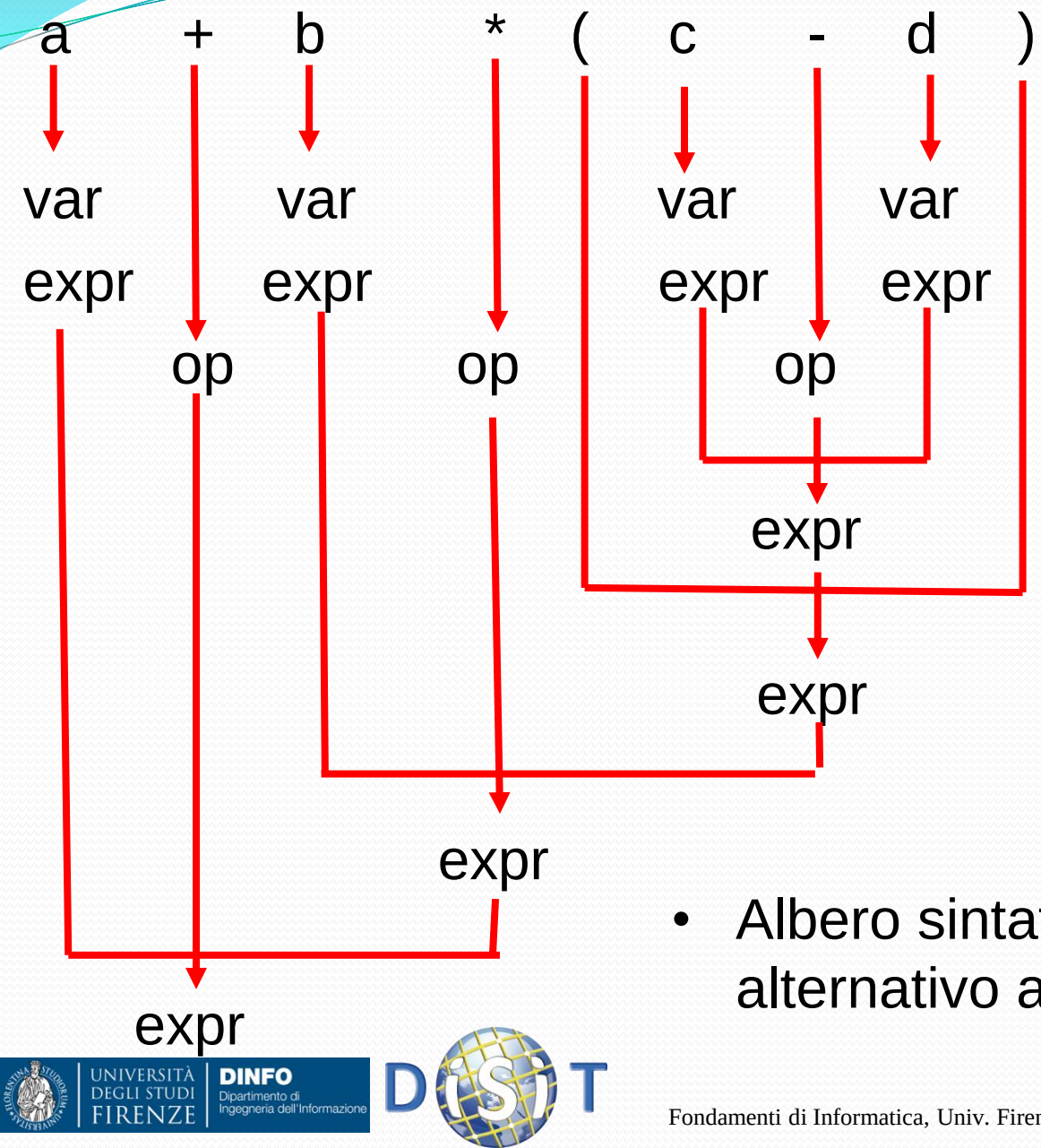
- a + b * (c-d)
  - Partendo da questa espressione è possibile trovare alberi sintattici diversi, questo dipende dall'ordine di applicazione delle produzioni
- a+b*-c
  - In qualsiasi ordine si decida di applicare le produzioni, questa espressione NON si chiude MAI

Albero sintattico – Esempi (2)



- p1: expr ⟶ var
- p2: expr ⟶ (expr)
- p3: expr ⟶expr op expr
- p4: op ⟶ +
- P5: op ⟶ -
- P6: op ⟶ *
- P7: op ⟶ /

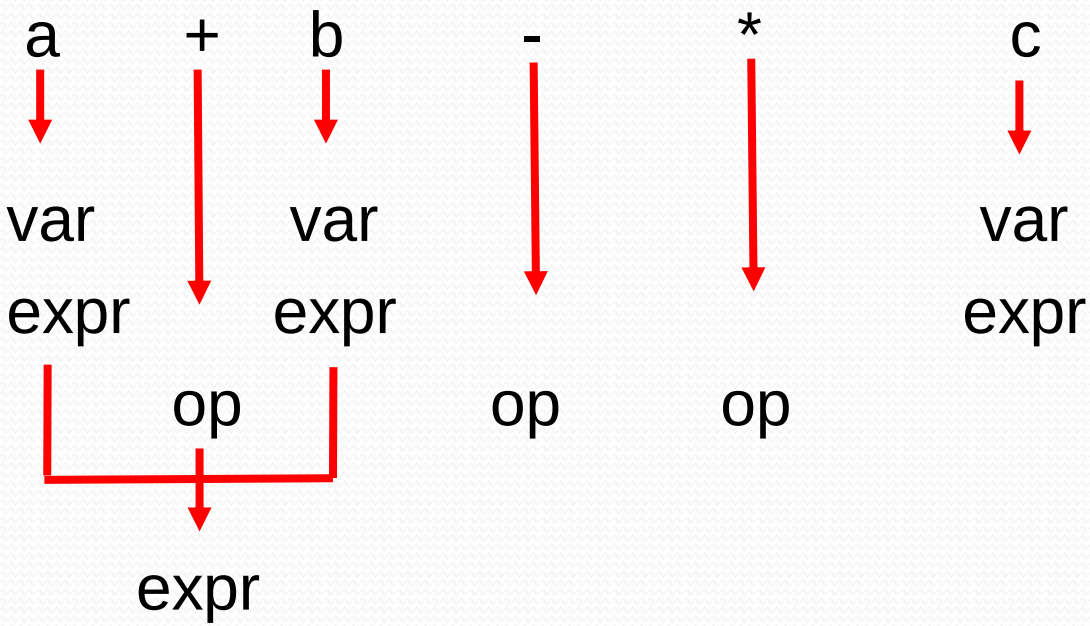
Albero sintattico – Esempi (3)



- p1: expr ⟶ var
- p2: expr ⟶ (expr)
- p3: expr ⟶expr op expr
- p4: op ⟶ +
- P5: op ⟶ -
- P6: op ⟶ *
- P7: op ⟶ /

- Albero sintattico alternativo al precedente

Albero sintattico – Esempi (3)



- p1: expr ⟶ var
- p2: expr ⟶ (expr)
- p3: expr ⟶expr op expr
- p4: op ⟶ +
- P5: op ⟶ -
- P6: op ⟶ *
- P7: op ⟶ /

- L'espressione a+b-*c NON è una forma legale della categoria sintattica expr NON essendo possibile costruire un albero che abbia expr come radice e a+b-*c come sequenza delle foglie

Il Metalinguaggio BNF

- Il BNF, Backus Naur Form, è un formalismo che semplifica la rappresentazione e la comprensione delle produzioni rispetto alla notazione elementare $\eta \rightarrow n$
- Il BNF è un linguaggio che serve per rappresentare altri linguaggi, ovvero è un metalinguaggio
- Esistono numerose estensioni e rappresentazioni del BNF, che vengono definite come extended BNF, EBNF

EBNF, Forma di Backus-Naur Estesa (1)

- Un programma è costruito come una sequenza di simboli o caratteri che formano il ‘testo’
- Come per i linguaggi naturali, ogni linguaggio di programmazione ha associato un insieme di regole rigidamente definito che descrive le modalità di costruzione di un programma valido per il linguaggio considerato
- Queste regole servono per:
  - Garantire al programmatore la correttezza e l’effetto del suo programma
  - Permettere di far comprendere il programma al calcolatore e a chiunque lo legga

EBNF, Forma di Backus-Naur Estesa (2)

- Le regole del linguaggio sono costituite da due parti:
  - SINTASSI:
    - insieme di regole che definiscono le relazioni possibili tra i singoli costrutti del linguaggio.
    - Le regole sintattiche definiscono il modo in cui le ‘parole’ (o vocabolario) del linguaggio si possono combinare per formare le ‘frasi’
    - La sintassi dei linguaggi di programmazione è la forma in cui devono essere scritti i programmi
    - ESEMPIO:
      - Lingua italiana: soggetto, verbo e complemento sono legati in base a specifiche regole di sintassi
  - SEMANTICA: …

EBNF, Forma di Backus-Naur Estesa (2)

- Le regole del linguaggio sono costituite da due parti:
  - SINTASSI:
    - […]
  - SEMANTICA:
    - E' il significato che viene dato alle sequenze e combinazioni di costrutti offerti da un linguaggio
    - Nel caso di linguaggi di programmazione, è il significato di un programma scritto in quel linguaggio
    - Di solito le regole semantiche sono stabilite in modo meno formale rispetto alle regole semantiche

EBNF, Esempi (1)

- Prendiamo in esame le due frasi:
  - Il gatto è verde
  - Il verde è gatto
- La prima frase è sintatticamente giusta MA semanticamente sbagliata
- La seconda frase è sbagliata sia dal punto vista semantico che da quello sintattico

- NOTA: E' da tenere presente sempre il contesto in cui si opera. Se ad esempio la prima frase fosse scritta in un romanzo di fantascienza, allora potrebbe anche risultare semanticamente corretta

Il metalinguaggio EBNF (1)

- In informatica si usano strumenti formali per descrivere la sintassi di un linguaggio: questi strumenti sono detti ‘metalinguaggi’
- Un metalinguaggio è un linguaggio che serve per descrivere altri linguaggio
- In particolare il metalinguaggio che serve per descrivere il linguaggio C è il BNF (Backus-Naur Form)
- Il BNF assume la notazione EBNF (Extended Backus-Naur Form) nella sua versione più aggiornata
- In generale nell’EBNF i costrutti sintattici sono definiti da termini, alcuni dei quali sono racchiusi tra parentesi angolari <>

EBFN e linguaggio C

EBNF e Tipi nel linguaggio C (1)

- Vediamo come i tipi in C si possono rappresentare in C facendo uso del metalinguaggio EBNF
- In un linguaggio di programmazione un tipo è caratterizzato dall'insieme dei valori rappresentati dalle operazioni che possono essere applicate su di essi
- Il tipo è caratterizzato anche dal modo con cui i valori sono codificati e dagli algoritmi che sono usati nella realizzazione delle operazioni
- Il C include 5 tipi elementari:
  - char, int, float, double, void
- Al tipo int possono essere applicati i modificatori:
  - short, long e unsigned

EBNF e Tipi nel linguaggio C (2)

- Il C include 5 tipi elementari:
  - char, int, float, double, void
- Al tipo int possono essere applicati i modificatori:
  - short, long e unsigned

<type> ::= char **|** [unsigned] [short **|** long] int **|** foat **|** double **|** void

Simbolo	Significato
<nome_label>	Definisce un simbolo non-terminale
::=	Assegnazione
|	OR logica
{}	0 o più elementi
[]	0 o 1 elementi

- In C sono ammessi come tipi validi: char, unsigned long int, unsigned short int, long int, short int, unsigned int, int, float, double, void

Esempi di tipi in C (1)

- Data la:

<type> :: = char | [unsigned] [short | long] int | foat | double | void

- Sono dichiarazioni ammissibili:
  - int a;
  - short b;
  - unsigned long int;
  - …
- Sono dichiarazioni NON ammissibili:
  - short float x;
  - unsigned char c;
  - long double t;
  - boolean b;
  - …

Esempi di tipi in C (2)

Main.c*

Esempi_base1 | (Ambito globale)

```
1   #include <stdbool.h>
2   #include <stdio.h>
3   #include <string.h>
4   #include <stdlib.h>
5   #define N 10
6   main() {
7       short char g;
8       unsigned int z;
9
10
```

99 %

Elenco errori

Intera soluzione | 2 Errori | 0 di 4 Avvisi | 0 Messaggi

	Codice	Descrizione
abc		combinazione di identificatori di tipo non valida
⊗	C2632	'short' non può essere seguito da 'char'

Tipi, variabili e costanti: char

<type> :: = char | [unsigned] [short | long] int | foat | double | void

- char: rappresenta i caratteri della tabella ASCII (numeri interi senza segno)

Tabella dei codici ANSI-ASCII

	032	@	064	`	096	€	0128		0160	À	0192	à	0224
!	033	A	065	a	097		0129	¡	0161	Á	0193	á	0225
"	034	B	066	b	098	‚	0130	¢	0162	Â	0194	â	0226
#	035	C	067	c	099	ƒ	0131	£	0163	Ã	0195	ã	0227
$	036	D	068	d	0100	„	0132	¤	0164	Ä	0196	ä	0228
%	037	E	069	e	0101	…	0133	¥	0165	Å	0197	å	0229
&	038	F	070	f	0102	†	0134	¦	0166	Æ	0198	æ	0230
'	039	G	071	g	0103	‡	0135	§	0167	Ç	0199	ç	0231
(	040	H	072	h	0104	ˆ	0136	¨	0168	È	0200	è	0232
)	041	I	073	i	0105	‰	0137	©	0169	É	0201	é	0233
*	042	J	074	j	0106	Š	0138	ª	0170	Ê	0202	ê	0234
+	043	K	075	k	0107	‹	0139	«	0171	Ë	0203	ë	0235
,	044	L	076	l	0108	Œ	0140	¬	0172	Ì	0204	ì	0236
-	045	M	077	m	0109		0141		0173	Í	0205	í	0237
.	046	N	078	n	0110	Ž	0142	®	0174	Î	0206	î	0238
/	047	O	079	o	0111		0143	¯	0175	Ï	0207	ï	0239
0	048	P	080	p	0112		0144	°	0176	Ð	0208	ð	0240
1	049	Q	081	q	0113	‘	0145	±	0177	Ñ	0209	ñ	0241
2	050	R	082	r	0114	’	0146	²	0178	Ò	0210	ò	0242
3	051	S	083	s	0115	“	0147	³	0179	Ó	0211	ó	0243
4	052	T	084	t	0116	”	0148	´	0180	Ô	0212	ô	0244
5	053	U	085	u	0117	•	0149	µ	0181	Õ	0213	õ	0245
6	054	V	086	v	0118	–	0150	¶	0182	Ö	0214	ö	0246
7	055	W	087	w	0119	—	0151	·	0183	×	0215	÷	0247
8	056	X	088	x	0120	˜	0152	¸	0184	Ø	0216	ø	0248
9	057	Y	089	y	0121	™	0153	¹	0185	Ù	0217	ù	0249
:	058	Z	090	z	0122	š	0154	º	0186	Ú	0218	ú	0250
;	059	[	091	{	0123	›	0155	»	0187	Û	0219	û	0251
<	060	\	092			0124	œ	0156	¼	0188	Ü	0220	ü	0252
=	061	]	093	}	0125		0157	½	0189	Ý	0221	ý	0253
>	062	^	094	~	0126	ž	0158	¾	0190	Þ	0222	þ	0254
?	063	_	095		0127	Ÿ	0159	¿	0191	ß	0223	ÿ	0255

- *Esempi:*
  - *il carattere 0 è codificato dal numero 48*
- *Le parentesi graffe sono rappresentate dai numeri 123 e 124 …*
- *Il numero 0 NON codifica alcun carattere, mentre invece viene usato come segno di terminazione nella codifica delle stringhe*

Tipi, variabili e costanti: int (1)

<type> :: = char | [unsigned] [short | long] int | foat | double | void

- int: rappresenta i numeri interi con segno
- Il numero di byte usati nella rappresentazione e quindi la dinamica dei valori può variare a seconda della architettura della macchina su cui viene compilato un programma (1 byte = 8 bit)
- Tipicamente sono usati 4 bytes

Tipi, variabili e costanti: int (2)

<type> :: = char | [unsigned] [short | long] int | float | double | void

- Al tipo intero possono essere applicati i 'modificatori' **long**, **short** e **unsigned**:
  - long: aumenta il numero di bytes usati nella codifica, espandendo la dinamica dei valori rappresentati
  - short: riduce il numero di bytes della codifica
    - NOTA: long e short SONO ALTERNATIVI
  - unsigned: codifica che non rappresenta il segno e raddoppia quindi la dinamica dei valori positivi
- L'effetto dei modificatori short e long dipende in realtà dalla architetture degli elaboratori. Tipicamente si ha:
  - short: 1byte (può essere minore o uguale alla dimensione di un int)
  - long: 4 bytes (può essere maggiore o uguale alla dimensione di un int)

Tipi, variabili e costanti: float, double e void

- float: rappresenta valori razionali in floating point

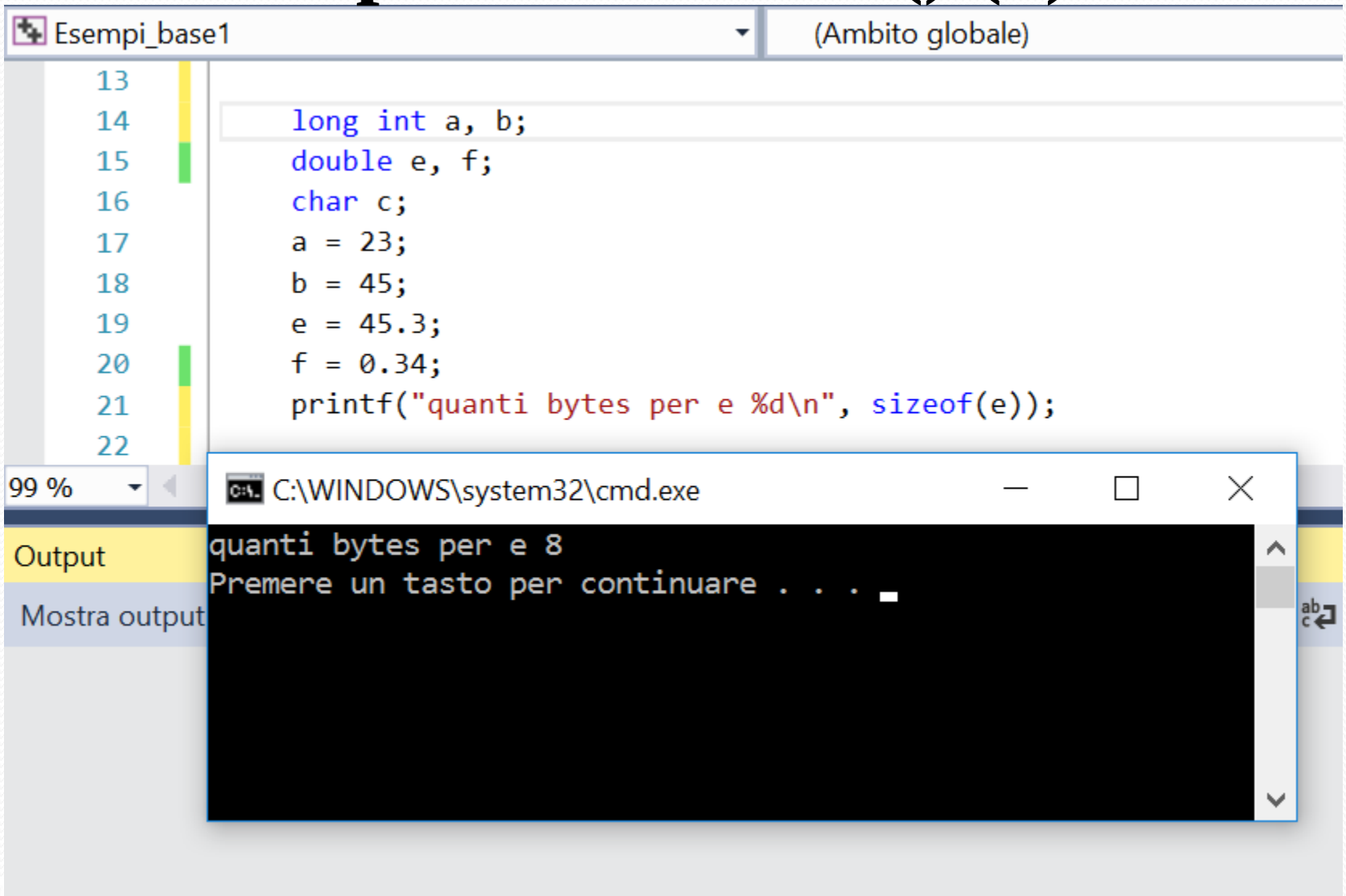
- double: rappresenta valori razionali in floating point in doppia precisione

- void: è il tipo nullo.
  - Non esistono variabili di tipo void
  - Il tipo void serve per specificare un tipo dove questo non ha nessun particolare effetto ma permette di mantenere regolarità nel linguaggio
  - Il ruolo del tipo void diventerà chiaro quando si tratteranno puntatori e funzioni

Operatore sizeof() (1)

- sizeof() non è un operatore intrinseco al compilatore e non fa parte di alcuna libreria
- Restituisce sempre un valore di tipo int, indipendentemente dal tipo di dato o di variabile specificato tra le parentesi.
- … supponiamo di avere effettuato una serie di dichiarazioni all'interno del nostro programma, se vogliamo conoscere il numero di bytes che sono stati riservati in memoria per ogni variabile, basta fare uso di sizeof(nome_variabile)
- ESEMPIO:
  - printf("quanti bytes per e %d", sizeof(e));
  - Stamperà in console il numero di byte riservati in memoria per la variabile e (ad esempio, 4, 2, 8, etc..)

Operatore sizeof() (2)



EBNF e Variabili nel linguaggio C (1)

- Una variabile è una locazione di memoria che contiene un valore di un tipo
- Il valore può essere modificato nel corso della computazione MENTRE il tipo è INVARIANTE
- Una variabile è associata ad un nome
- Poter fare riferimento ad una variabile tramite il nome è uno degli aspetti caratterizzanti di un linguaggio simbolico
- Per poter fare riferimento ad una variabile attraverso il suo nome, questo DEVE essere specificato in una DICHIARAZIONE

EBNF e Variabili nel linguaggio C (2)

- Una DICHIARAZIONE ha sempre il seguente formato (è rappresentata dalle seguenti produzioni):

<declaration> :: = <type_decl><decl>;

Con:

<type_decl> :: = <type>|…

<type> :: = char | [unsigned] [short | long] int | foat | double | void |…

<decl> :: = <identifier> |…

- Il termine <identifier> è il nome della variabile. La sintassi informale sarà quindi la seguente:
  - È una sequenza di caratteri alfabetici e numerici
  - È possibile l'uso dell'underscore _
  - Il primo carattere NON può essere un numero
  - Può avere una lunghezza arbitraria ma il compilatore considera solo i primi 32 caratteri

EBNF e Variabili nel linguaggio C (3)

<declaration> :: = <type_decl><decl>;

con:

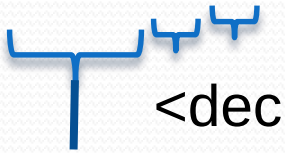
<type_decl> :: = <type>|…

<type> :: = char | [unsigned] [short | long] int | foat | double | void

<decl> :: = <identifier> |…

<declaration>

int a ;



il **;** determina la fine della dichiarazione

<decl> // è il nome della variabile

<type_decl> :: = <type> //è il tipo

EBNF e Variabili nel linguaggio C (4)

- Fino ad ora abbiamo visto le regole **sintattiche** tramite EBNF per le DICHIARAZIONI
- La **semantica** di una dichiarazione <declaration> consiste nel:
  - Associare un nome e un tipo ad una entità referenziabile nel programma
- Questo concetto si applica a diversi tipi di entità: variabile, array e funzione (che vedremo)
- Quando si effettua una dichiarazione, il compilatore provvede ad allocare in memoria lo spazio in cui la variabile viene mantenuta

EBNF e Variabili nel linguaggio C (5)

- Una variabile dichiarata può essere referenziata attraverso il suo identificatore <identifier>

<var> ::= <identifier> | …

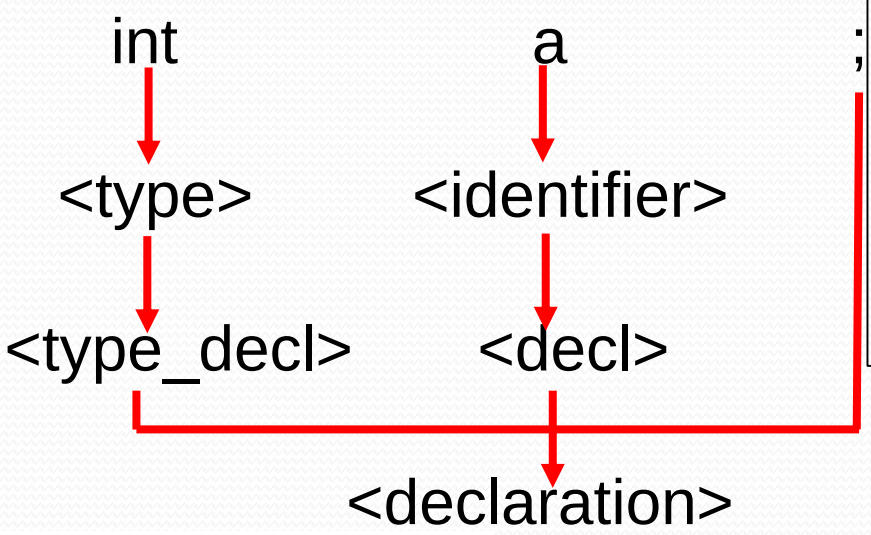
Variabili e alberi sintattici (1)

- Consideriamo le due dichiarazioni:

int a;

unsigned long int b;

- Per verificare la legalità delle due dichiarazioni è possibile utilizzare gli alberi sintattici



*<declaration> :: = <type_decl><decl>**;***
<type_decl> :: = <type>|…
<type> :: = char | [unsigned] [short | long] int | foat | double | void
<decl> :: = <identifier> |…

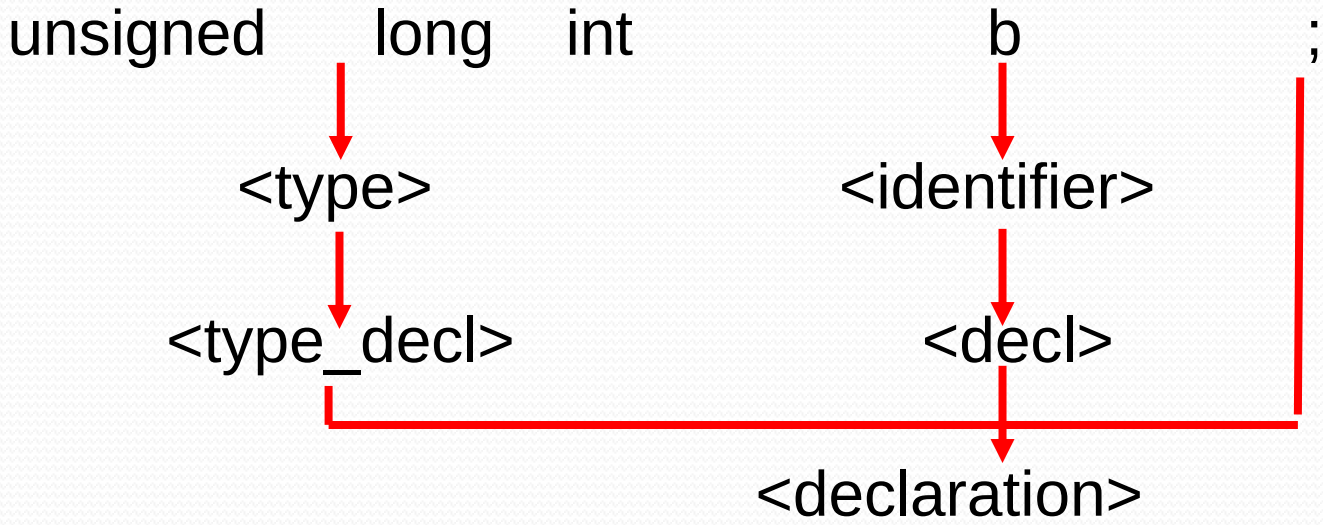
Variabili e alberi sintattici (2)

- Sapendo che:

<declaration> :: = <type_decl><decl>;
<type_decl> :: = <type>|…
<type> :: = char | [unsigned] [short | long] int | foat | double | void
<decl> :: = <identifier> |…

- Verifichiamo la correttezza della dichiarazione:

unsigned long int b;



EBNF e Costanti

- Le costanti NON possono essere modificate nel corso dell'elaborazione
- Le costanti hanno un tipo
- Il tipo associato alla costante può essere:
  - Dichiarato esplicitamente
  - Derivato dal compilatore in base al formato in cui una costante è espressa (definita) nel programma
- Una costante non è memorizzata in una locazione di memoria ma è parte del codice

EBNF e Operatori (1)

- Le espressioni combinano riferimenti a variabili e costanti attraverso gli operatori

- E' possibile raggruppare gli operatori in base alle seguenti categorie semantiche:
  - Aritmetici
  - Logici
  - Relazionali
  - Sui bit
  - di Assegnamento
  - di Composizione

EBNF e Operatori (2)

- Una espressione può essere composta dai termini elementari costanti e variabili, in EBNF:

<expr> ::= <const> | <var> | …

- Sulle variabili di tipo intero possono essere applicati gli operatori di incremento ++ e decremento --, sia in forma prefissa che suffissa, quindi si ha:

<expr> ::= <const>|<var>|++<var>|--<var>|<var>++|<var>--
| …

EBNF e Operatori (3)

- Esistono poi una serie di operatori attraverso i quali la combinazione di espressioni produce un'altra espressione:
- Operatori Aritmetici: +, -, *, /, %
- Operatori Relazionali: <=, <, =, >, >=, !=, ==
- Operatori Logici: && (AND), || (OR), ! (NOT)
- Operatori di manipolazione dei bit: &, |, ~, …

- NOTA:
  - Operatori UNARI: ! (NOT) e ~ (not bit a bit)
  - Operatori BINARI; tutti gli altri

EBNF e Operatori (4)

- Gli Operatori si dividono in due categorie: Unari o Binari, quindi in EBNF si ha:

<expr> ::= <const>|<var>|++<var>|--<var>|<var>++|<var>--
|<expr_1><op_2><expr_2>| <op_1><expr_1>
|…

- con:

<op_2> ::= +, -, *, /, %, <=, <, =, >, >=, !=, ==, &&, ||, &, |, …
<op_1> ::= !, ~

EBNF e Operatori (5)

- L'operazione di assegnamento è essa stessa una espressione in C, quindi in EBNF si estende ancora la definizione di espressione e si ha:

<expr> ::= <const>|<var>|++<var>|--<var>|<var>++|<var>--
| <expr_1><op_2><expr_2>|
<op_1><expr_1>|<var>=<expr_1>|<var><op_2><expr_2>|…

Esempi:

<var>=<expr_1> //operatori di assegnazione
a = b+1;
c = 6.92;

<var><op_2><expr_2> //operatori di assegnazione composti
a += b + c; // equivale → a = a + b + c;
a -= 1; // Equivale a scrivere: a = a - 1;

Operatore Ternario (1)

- Ricordiamo l'istruzione if… else

```
…
if(a>b){                    // (a>b) → guardia
        b =10;              // istruzioni che si trovano dentro le
        a = b + 125;        // graffe → corpo 2
}
else{                       //si esegue il corpo 2 SE a<=b
        b=5;
}
```

- Può essere rappresentata anche attraverso l'operatore ternario:

<guardia> ? <corpo1> : <corpo2>

Ovvero: "SE la guardia è verificata, viene eseguito il corpo1, ALTRIMENTI viene eseguito il corpo2"

Operatore Ternario (2)

```
…
if (a > b) //guardia
        max = a; //corpo1
else
        max = b; //corpo2
```

- Usando l'operatore Ternario si ha:

```
max = (a > b) ? a : b;
//max = (guardia) ? corpo1 : corpo2;
```

- le parentesi servono x comprendere meglio il senso e racchiudono la guardia, ovvero: "SE la guardia è verificata, viene eseguito il corpo1, ALTRIMENTI viene eseguito il corpo2"

EBNF e Operatore Ternario (3)

- Sapendo che la sintassi per l'operatore Ternario è:

<guardia> ? <corpo1> : <corpo2>

- Appare chiaro che in EBNF si ha:

<expr> ::= <expr_1>?<expr_2>: <expr_3>

Conversione tra tipi: casting implicito ed esplicito (1)

- Le variabili e le costanti appartenenti a tipi diversi possono in vari casi essere convertite da un tipo all'altro senza perdita di informazione
- Questa possibilità dipende da:
  - Tipi di dati convolti
  - Valore del dato
  - Rappresentazione che viene usata dallo specifico linguaggio e/o compilatore
- Le conversioni tra tipi si possono classificare in:
  - implicite
  - esplicite

Conversione tra tipi: casting implicito ed esplicito (2)

- Le conversioni tra tipi si dicono **implicite** quando il compilatore non richiede nessun intervento da parte del programmatore per specificare il tipo di conversione

- Le conversioni tra tipi si dicono **esplicite** quando è necessario specificare in dettaglio la conversione che deve essere effettuata

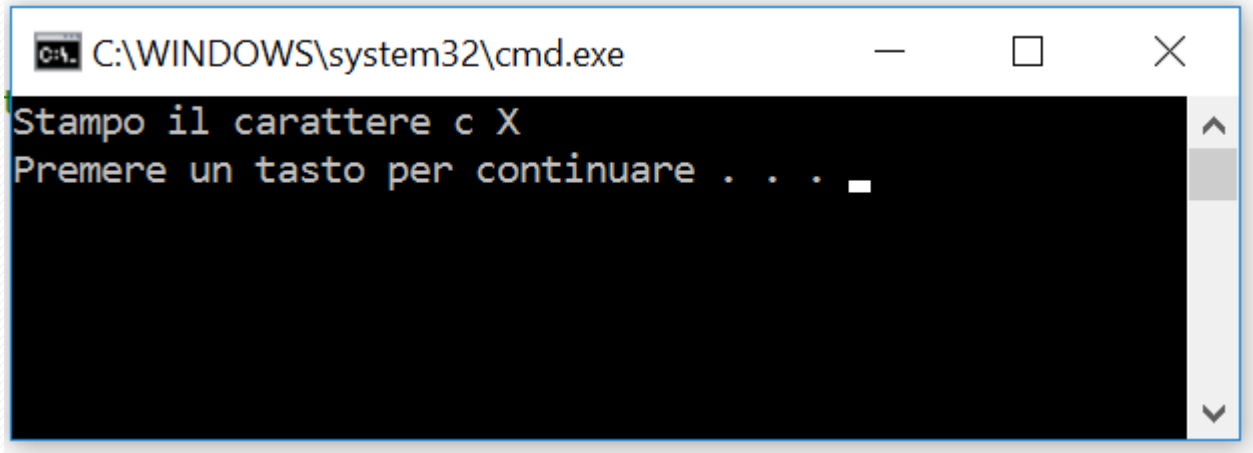
- Esempio di conversione implicita (che vengono realizzate in automatico da compilatore perché NON si ha perdita di informazione):
  - Conversione da numero carattere a intero

Cast implicito esempi (1)

```
#include <stdio.h>
main() {
        //cast implicito
        int a;
        char c;
        c = 88;
        a = c;
        printf("Stampo il carattere c %c\n", c);
        printf("Stampo il numero intero
        equivalente al carattere (tabella ASCII) %d\n", a);
}
```

@	064
A	065
B	066
C	067
D	068
E	069
F	070
G	071
H	072
I	073
J	074
K	075
L	076
M	077
N	078
O	079
P	080
Q	081
R	082
S	083
T	084
U	085
V	086
W	087
X	088
Y	089
Z	090

Cast implicito esempi (2)



…

```
int a;
char c;
a = 88;
c = a;
printf("Stampo il carattere c %c\n", c);
```

…

@	064
A	065
B	066
C	067
D	068
E	069
F	070
G	071
H	072
I	073
J	074
K	075
L	076
M	077
N	078
O	079
P	080
Q	081
R	082
S	083
T	084
U	085
V	086
W	087
X	088
Y	089
Z	090

Conversione tra tipi: casting implicito ed esplicito (3)

- Le conversioni esplicite si effettuano principalmente in due modi, per mezzo di:
  - Operatore di cast composto dal tipo della variabile verso la quale si vuole effettuare la conversione racchiuso tra parentesi tonde
    - … c = (char)a; //conversione verso u carattere
  - Uso di specifiche funzioni di libreria

Esempi di conversioni implicite ed esplicite

```
…
int a, b;
float e, f;
char c;
a = 23;
b = 45;
e = 45.3;
f = 0.34;
…. //esempi
a = e;          //errore, conversione da float a int, possibile perdita dati
a = (int)e;     //conversione esplicita
a = e*f;        //errore, conversione da float a int, possibile perdita dati
a = (int)e*f;         //conversione esplicita
f = a*45.3;           //conversione implicita a float
f = a * (int)45.3;  //conversione esplicita a implicita a float
c = a;                //conversione da int a char
```

NOTA se si commentano le righe lasciandone una per volta attiva e si attiva il debug -> si vedono gli warning nell'elenco degli errori

Esempi di conversioni implicite ed esplicite (2)

```
…
int a;
e = 45.3;
f = 0.34;
a = f*e;        //errore, conversione da float a int, possibile perdita dati
a = (int)f*e;        //conversione esplicita
```

- Queste assegnazioni generano risultati differenti
- Questo accade perché esistono delle **priorità tra gli operatori**, in particolare l'operatore di cast ha priorità maggiore rispetto a quello di prodotto

Caso1: a = f*e; RESTITUISCE 15, ovvero prima esegue la moltiplicazione tra float (0.34*45.3=15.402) e poi converte a int

Caso2: a = (int)f*e; RESTITUISCE 0, perché prima converte f a intero (f=0) e dopo effettua la moltiplicazione

Precedenza fra operatori

Livello di precedenza	Operatori	Descrizione
0	`::`	visibilità (class name:: member)
1	`() [] -> .`	Precedenza, subscritto di matrice/array, membro di struttura da puntatore (metodo da puntatore), membro di struttura da struttura (metodo da oggetto). (da sinistra a destra)
2	`! ~ ++ -- - + (type) * & sizeof() delete new`	Not logico, complemento a uno, incremento, decremento, cambio di segno, segno positivo, casting, indirezione (variabile puntata da), indirizzo di, dimensione di, delete, new. (da destra a sinistra)
3	`* / %`	Prodotto, divisione, modulo. (da sinistra a destra)
4	`+ -`	Somma (più), differenza (meno). (da sinistra a destra)
5	`<< >>`	Shift a sinistra sui bit, shift a destra sui bit. (da sinistra a destra) (utilizzati anche per lo stream di ingresso e uscita)
6	`< <= > >=`	Minore?, minore uguale?, maggiore?, maggiore uguale? (da sinistra a destra)
7	`== !=`	Uguale?, diverso? (da sinistra a destra)
8	`&`	And sui bit. (da sinistra a destra)
9	`^`	Xor (or esclusivo) sui bit. (da sinistra a destra)
10	`|`	Or sui bit. (da sinistra a destra)
11	`&&`	And logico. (da sinistra a destra)
12	`||`	Or logico. (da sinistra a destra)
13	`?:`	espressione condizionale. (da destra a sinistra)
14	`= += -= *= /= %= &= ^= |= <<= >>=`	Assegnazioni con operazione. (da destra a sinistra)
15	`,`	Virgola come separatore di istruzioni. (da sinistra a destra)

EBNF e Operatore di cast

- Sapendo che la sintassi per l'operatore di cast (cast esplicito) è: (tipo_variabile)variabile

- Appare chiaro che in EBNF si ha:
  - <expr> ::= (<type_decl>)<expr> …
  - con:
    - <type_decl> :: = <type>|…
    - <type> :: = char | [unsigned] [short | long] int | foat | double | void

EBNF e Operatori (6)

- L'operatore Ternario insieme alle parentesi (e alla virgola che vedremo in futuro) servono per: raccogliere, sequenzializzare (concatenare) le espressioni
- L'operatore di cast serve per controllare il tipo con cui è restituito il risultato di una espressione
- In EBNF si estende nuovamente la definizione di espressione e si ha:

<expr> ::= <const>|<var>|++<var>|--<var>|<var>++|<var>--
|<expr_1><op_2><expr_2>|<op_1><expr_1>
|<var>=<expr_1>|<var><op_2><expr_2>
|<expr_1>?<expr_2>:<expr_3> | (<expr>)
| (<type_decl>)<expr> |…

Link utili

- Testo di riferimento:
  - Fondamenti di programmazione. Linguaggio C, strutture dati e algoritmi elementari, C++
  - Enrico Vicario, Società editrice ESCULAPIO
- http://www.disit.dinfo.unifi.it

Fondamenti di Informatica

Eng. Ph.D. Michela Paolucci

DISIT Lab *http://www.disit.dinfo.unifi.it/*
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-2758515, fax: +39-055-2758570
michela.paolucci@unifi.it