Università degli Studi di Ferrara

Dai Riconoscitori agli Interpreti

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2015/2016

Prof. MARCO GAVANELLI

Dipartimento di Ingegneria

DAI PURI RICONOSCITORI…

Finora abbiamo considerato *puri riconoscitori,* che

- **accettano in ingresso una stringa di caratteri**
- **riconoscono se essa appartiene al linguaggio**

La risposta è dunque del tipo *"sì o no":*

- **non ha importanza *come* si arriva a stabilire se la frase è lecita**
- **in particolare, *si può sempre sostituire una grammatica con una equivalente*, perché l'effetto finale è identico.**

Però, con un puro riconoscitore non si va molto in là..

… AGLI INTERPRETI

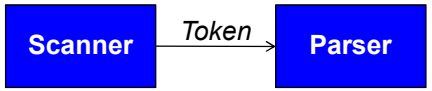
Un *interprete* è più di un puro riconoscitore

- **non solo riconosce se una stringa appartiene al linguaggio..**
- **..ma *esegue azioni* in base al *significato (semantica)* della frase**
  - **può svolgerle direttamente → *valutazione immediata***
  - **o costruire *strutture dati* per permettere di svolgerle *in un secondo tempo*.**

Conseguenze:

- **la sequenza di derivazione *diventa importante* perché contribuisce a definire il significato della frase**
- ***non* è detto di poter sostituire una grammatica con una equivalente perché *l'equivalenza è tale solo "ai morsetti"!***

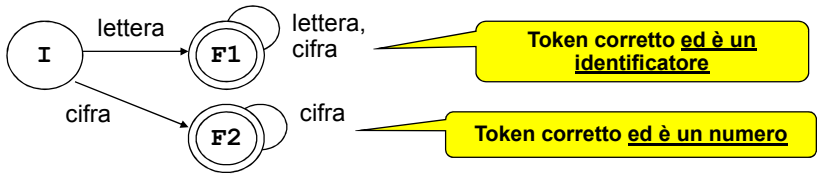
INTERPRETE: STRUTTURA



- **Un interprete è di solito strutturato su due componenti:**
  - **l'*analizzatore lessicale (scanner)***
  - **l'*analizzatore sintattico-semantico (parser)***
- **Spesso organizzati in un'architettura *client/server***
  - **lo scanner analizza le *parti regolari* del linguaggio, fornendo al parser *singole parole (token) già aggregate,*** evitandogli di doversi occupare dei dettagli relativi ai singoli caratteri
  - **il parser riceve dallo scanner le *singole parole (token) e le usa come elementi terminali del suo linguaggio per valutare la correttezza della loro sequenza:*** il lavoro già svolto dallo scanner gli permette di concentrarsi sulle parti context-free del linguaggio.
- Tipicamente il *client* chiede iterativamente al *server* i vari token che compongono la frase, via via che gli servono

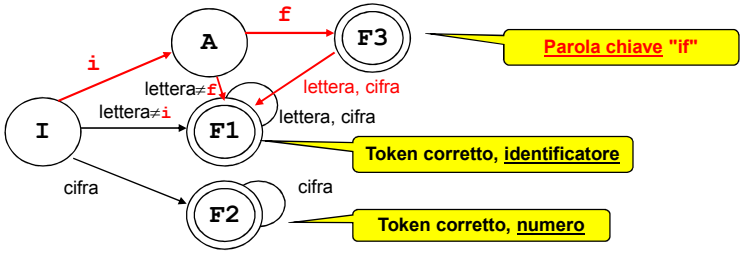
ANALISI LESSICALE

- **L'*analisi lessicale* consiste nella individuazione delle singole parole (*token*) che compongono una frase**
- **Ciò viene fatto raggruppando i singoli caratteri dell'input secondo le produzioni regolari associate alle diverse possibili *categorie lessicali* (*identificatore, numero, etc.*)**
- **L'analizzatore lessicale può *categorizzare i token mentre li analizza* semplicemente *osservando in quale stato finale del suo RSF* si viene a trovare:**



PAROLE CHIAVE

- **Tuttavia, cablare ogni dettaglio nella struttura del RSF *non è una strategia vincente***
  - Un linguaggio comprende *le parole chiave*, *simboli speciali..*
  - Per tenerne conto, un RSF dovrebbe avere *regole ultra specifiche* capaci di "schivarle" una ad una → struttura RSF complicatissima!
- **Ad esempio, per escludere la parola chiave `if` dagli identificatori leciti, l'RSF precedente diventerebbe:**



TABELLE

- **Per evitare di cristallizzare nella struttura del RSF** le parole chiave, i simboli speciali, etc, conviene ***agire diversamente:***
  - Categorizzare ***in prima battuta*** le parole chiave come identificatori → grammatica e struttura RSF molto semplici
  - .. salvo poi **ri-categorizzarle correttamente *in seconda battuta*,** consultando opportune ***tabelle*** **che incapsulano la conoscenza di dettaglio del linguaggio**
    - ***tabella delle parole-chiave*** del linguaggio
    - ***tabella dei simboli speciali*** del linguaggio
  - In pratica, si sposta la decisione da un piano puramente sintattico a uno sintattico-semantico

Risultato: scanner modulare, estendibile, snello

ANALISI SINTATTICA TOP-DOWN

- **In presenza di grammatiche LL(1), *l'analisi top-down ricorsiva discendente* offre una tecnica semplice e diretta per costruire il riconoscitore.**
- **Negli esempi visti, ogni funzione restituiva un *boolean***
  - **erano puri riconoscitori → la risposta attesa era "sì o no"**
- **Per *passare da un puro riconoscitore a un interprete* occorre *propagare qualcosa di più* di un "sì o no"**
  - un **valore**, se l'obiettivo è la **valutazione immediata** (interprete) in un qualche dominio
  - un **albero**, se l'obiettivo è la **valutazione differita** (compilatore o interprete a più fasi)
    → la vera valutazione nel dominio di interesse avviene più avanti.

IL CASO DI STUDIO CLASSICO: ESPRESSIONI ARITMETICHE

- **Si supponga di voler riconoscere *espressioni aritmetiche* con le quattro operazioni +, −, *, /**

`3+4-5`	`3+4*5`	`9-4-1`	`9-4/2`

- **Un *puro riconoscitore* deve solo dire *se sono corrette***
  - ogni funzione restituisce un *boolean*
- **Un *interprete* deve anche dire *quanto valgono***
  - se il dominio sono gli *interi* (Z), il *risultato* può essere un *valore int*
  - se il dominio sono i *reali* (R), il *risultato* può essere un *valore double*
  - se l'obiettivo è invece la *valutazione differita*, il *risultato* può essere un *opportuno oggetto* adatto a rappresentare un *albero*

ANALISI DEL DOMINIO – SINTASSI

- **Le espressioni aritmetiche ci sono state insegnate fin dalle scuole elementari, tipicamente utilizzando una *notazione infissa* basata sui *quattro operatori* +, −, ×, :**

`3+4-5`	`3+4×5`	`9-4-1`	`9-4:2`

- **curiosamente sostituiti dagli informatici in +, −, *, /**

`3+4-5`	`3+4*5`	`9-4-1`	`9-4/2`

- **Ad essa si accompagnano spesso le *parentesi* per esprimere priorità e associatività "non standard"**

`3+4*5`	`(3+4)*5`	`9-4-1`	`9-(4-1)`

ANALISI DEL DOMINIO – SEMANTICA

NEL DOMINIO ARITMETICO USUALE:

- **i *valori numerici* si assumono espressi in *notazione posizionale su base dieci***
  - quindi, "15" è *quindici* e non *sei* o *quattro*
- **il *significato inteso* dei *quattro operatori* è quello di *somma, sottrazione, moltiplicazione, divisione***
- **si introducono le nozioni di *priorità* e *associatività***
  - ***priorità* fra operatori *diversi***
    gli operatori moltiplicativi sono di solito prioritari su quelli additivi
    → `3+4*5` denota *ventitré*, *non trentacinque*
  - ***associatività* fra operatori *equiprioritari***
    solitamente si associa a sinistra
    → `9-4-1` denota *quattro (9-4)-1*, *non sei 9-(4-1)*

UNA GRAMMATICA PER LE ESPRESSIONI

Consideriamo il linguaggio **L(G)** relativo alla seguente ***grammatica per espressioni aritmetiche:***

VN = { EXP }

VT = { +, *, −, /, num }

S = EXP

P = {

```
EXP ::= EXP + EXP    // plusexp
EXP ::= EXP - EXP    // minusexp
EXP ::= EXP * EXP    // timesexp
EXP ::= EXP / EXP    // divexp
EXP ::= num          // numexp
```

}

num denota la notazione in base 10 di un numero intero senza segno, che si suppone nota (cfr. tipo 3)

Esempi di frasi lecite

Espressione	*Valore*
`5 + 3`	`8`
`2 + 3 * 4`	`14`
`5 - 3 - 1`	`1`

Esempi di frasi illecite

Espressione
`5 + 3 +`
`(2 + 3) * 4`
`-5`

UNA GRAMMATICA PER LE ESPRESSIONI

Consideriamo il linguaggio **L(G)** relativo alla seguente ***grammatica per espressioni aritmetiche:***

VN = { EXP }

VT = { +, *, –, /, num }

S = EXP

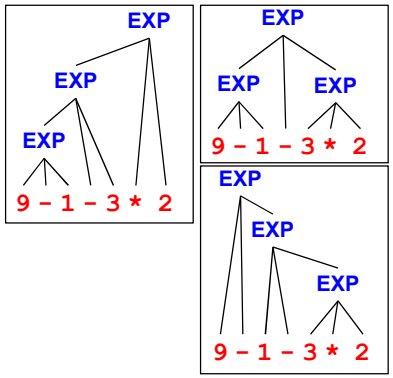
P = {

EXP	::=	EXP + EXP	// *plusexp*
EXP	::=	EXP – EXP	// *minusexp*
EXP	::=	EXP * EXP	// *timesexp*
EXP	::=	EXP / EXP	// *divexp*
EXP	::=	num	// *numexp*

}

num denota la notazione in base 10 di un numero intero senza segno, che si suppone nota (cfr. tipo 3)

È una grammatica *ambigua*



- Le descrizioni ambigue vanno *evitate* ogni volta che è possibile
- **Meglio sarebbe *cablare nella struttura della grammatica* le informazioni cruciali!**

UNA GRAMMATICA PER LE ESPRESSIONI

Consideriamo il linguaggio **L(G)** relativo alla seguente ***grammatica per espressioni aritmetiche:***

VN = { EXP }

VT = { +, *, –, /, num }

S = EXP

P = {

EXP	::=	EXP + EXP	// *plusexp*
EXP	::=	EXP – EXP	// *minusexp*
EXP	::=	EXP * EXP	// *timesexp*
EXP	::=	EXP / EXP	// *divexp*
EXP	::=	num	// *numexp*

}

num denota la notazione in base 10 di un numero intero senza segno, che si suppone nota (cfr. tipo 3)

Semantica informale:

- **ogni *Exp* è un'espressione**
- **se *Exp* è *num*, l'espressione denota un intero e il valore dell'espressione coincide con quello del numero.**
- **se invece *Exp* è *EXP op EXP* l'espressione denota il valore ottenuto *applicando* l'operatore ai valori denotati dalle due espressioni e_1,e_2. *Si conviene che gli operatori *,/ abbiano priorità maggiore rispetto agli operatori +,–.***
  Operatori equiprioritari sono valutati da sinistra a destra.

UNA GRAMMATICA «A STRATI»

- **Si può *dare una struttura gerarchica* alle espressioni, così da *esprimere intrinsecamente le priorità* degli operatori e *la loro associatività.***

VN = { EXP, TERM, FACTOR }

VT = { +, *. –, /, (,), num }

P = {

EXP	::=	TERM	1
EXP	::=	EXP + TERM	
EXP	::=	EXP – TERM	
TERM	::=	FACTOR	2
TERM	::=	TERM * FACTOR	
TERM	::=	TERM / FACTOR	
FACTOR	::=	num	3
FACTOR	::=	(EXP)	

}

APPROCCIO A STRATI: IDEA

- **Ogni strato considera *terminali* gli elementi linguistici *definiti in altri strati***
  - **EXP considera terminali +, – e TERM**
  - **TERM considera terminali *, / e FACTOR**
  - **FACTOR considera terminali num, (,) e EXP**

EXP	::=	TERM	1
EXP	::=	EXP + TERM	
EXP	::=	EXP – TERM	
TERM	::=	FACTOR	2
TERM	::=	TERM * FACTOR	
TERM	::=	TERM / FACTOR	
FACTOR	::=	num	3
FACTOR	::=	(EXP)	

STRATI e SOTTO-LINGUAGGI

- **Ogni strato definisce quindi *un suo sotto-linguaggio* che usa quei «terminali»**
  - L(EXP) = TERM ± TERM ± TERM …
  - L(TERM) = FACTOR */ FACTOR */ FACTOR …
  - L(FACTOR) = `num` | (EXP)

EXP	::=	TERM	1
EXP	::=	EXP + TERM	
EXP	::=	EXP – TERM	
TERM	::=	FACTOR	2
TERM	::=	TERM * FACTOR	
TERM	::=	TERM / FACTOR	
FACTOR	::=	num	3
FACTOR	::=	(EXP)	

Il sotto-linguaggio di ***num*** (regolare) è **indipendente dal resto:** potrebbe adottare una sintassi binaria, romana…

NON-AMBIGUITÀ e RESPONSABILITÀ

- **La grammatica *non è più ambigua* perché ogni strato può produrre *solo certi operatori***
  - somme e sottrazioni sono di competenza del livello 1
  - moltiplicazioni e divisioni sono di competenza del livello 2
  - singoli valori e sotto-espressioni competono al livello 3

EXP	::=	TERM	1
EXP	::=	EXP + TERM	
EXP	::=	EXP – TERM	
TERM	::=	FACTOR	2
TERM	::=	TERM * FACTOR	
TERM	::=	TERM / FACTOR	
FACTOR	::=	num	3
FACTOR	::=	(EXP)	

COMPITI DEI VARI STRATI

- **Gli strati superiori «aggregano» entità prodotte in strati inferiori**
  - somme e sottrazioni *aggregano termini*
  - moltiplicazioni e divisioni *aggregano fattori*
  - i fattori sono *entità atomiche* (semplici o composte)

EXP	::=	TERM	1
EXP	::=	EXP + TERM	
EXP	::=	EXP – TERM	
TERM	::=	FACTOR	2
TERM	::=	TERM * FACTOR	
TERM	::=	TERM / FACTOR	
FACTOR	::=	num	3
FACTOR	::=	(EXP)	

PRIORITÀ DEGLI OPERATORI

- **La stratificazione *induce priorità fra gli operatori:* le entità di strati bassi vanno sintetizzate *per prime***
  - MAX priorità: procurarsi i fattori
  - MED priorità: aggregare i fattori in termini
  - MIN priorità: aggregare i termini in espressioni

EXP	::=	TERM	Priorità MIN
EXP	::=	EXP + TERM	
EXP	::=	EXP – TERM	
TERM	::=	FACTOR	Priorità MED
TERM	::=	TERM * FACTOR	
TERM	::=	TERM / FACTOR	
FACTOR	::=	num	Priorità MAX
FACTOR	::=	(EXP)	

ASSOCIATIVITÀ DEGLI OPERATORI

- **Entro ogni strato, la ricorsione (se presente) stabilisce *come si aggregano entità di pari livello***
  - **Ricorsione SINISTRA = associatività operatori A SINISTRA**
  - **Ricorsione DESTRA = associatività operatori A DESTRA**
  - **Nessuna ricorsione = operatori NON ASSOCIATIVI**

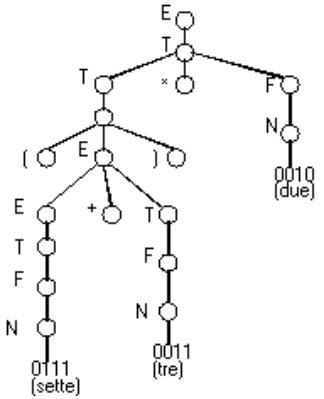
EXP ::= TERM EXP ::= EXP + TERM EXP ::= EXP – TERM	Associatività SINISTRA
TERM ::= FACTOR TERM ::= TERM * FACTOR TERM ::= TERM / FACTOR	Associatività SINISTRA
FACTOR ::= num FACTOR ::= (EXP)	(N/A)

ESEMPI
(con numeri in notazione binaria)

Frase:

`(0111 + 0011) * 0010`

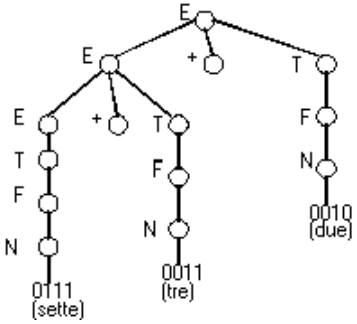
Albero di derivazione:



Frase:

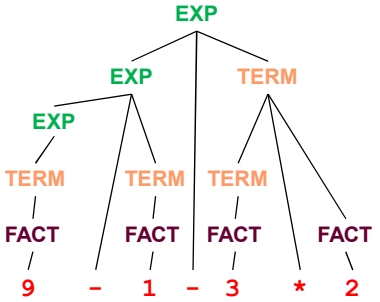
`0111 + 0011 + 0010`

Albero di derivazione:



ANALISI TOP-DOWN..?

EXP	::=	TERM
EXP	::=	EXP + TERM
EXP	::=	EXP – TERM
TERM	::=	FACTOR
TERM	::=	TERM * FACTOR
TERM	::=	TERM / FACTOR
FACTOR	::=	num
FACTOR	::=	(EXP)



- **La grammatica a lato è *ricorsiva a sinistra* per inglobare nella sua struttura l'associatività desiderata per gli operatori.**
- **PECCATO che la ricorsione sinistra sia *incompatibile con l'analisi ricorsiva discendente,* ossia con la principale tecnica di costruzione di parser!**
  AARGHH!!
- ***Come fare?***
  - **se eliminiamo la ricorsione sinistra, cambia l'associatività degli operatori!**
  - **è un problema culturale!**

VARIANTE 1 – ASSOCIATIVA A DESTRA

Se si riscrivessero le regole come sotto, in forma *ricorsiva a destra:*

VN = { EXP, TERM, FACTOR }
VT = { +, *. –, /, (,), num }

P = {

EXP	::=	TERM
EXP	::=	TERM + EXP
EXP	::=	TERM – EXP
TERM	::=	FACTOR
TERM	::=	FACTOR * TERM
TERM	::=	FACTOR / TERM
FACTOR	::=	num
FACTOR	::=	(EXP)

}

L'ordine di *priorità* non cambierebbe, ma l'associatività fra operatori equiprioritari sarebbe ora a destra, dando luogo a un'aritmetica *alquanto inusuale!*

Ad esempio, la frase:

`0111 – 0011 – 0010`

verrebbe derivata come

sette - (tre-due)

anziché come

(sette - tre) – due

Una semantica *basata sull'albero di derivazione* risulta *diversa!*

GRAMMATICA ASSOCIATIVA A DESTRA
ANALISI TOP-DOWN

VN = { EXP, TERM, FACTOR }
VT = { +, *. –, /, (,), num }

P = {
EXP ::= TERM
EXP ::= TERM + EXP
EXP ::= TERM – EXP
TERM ::= FACTOR
TERM ::= FACTOR * TERM
TERM ::= FACTOR / TERM
FACTOR ::= num
FACTOR ::= (EXP)
}

- La grammatica "marziana" a lato, *ricorsiva a destra,* è compatibile con l'analisi ricorsiva discendente..
- *..ma A CARO PREZZO: tutte le associatività sono rovesciate!*
- Si può davvero convincere un'intera cultura a cambiare approccio solo perché farebbe comodo a noi..?

VARIANTE 2 – NON ASSOCIATIVA

Pur mantenendo lo stesso ordine di *priorità* fra gli operatori, si potrebbe anche *fare del tutto a meno* dell'associatività.

VN = { EXP, TERM, FACTOR }
VT = { +, *. –, /, (,), num }

P = {
EXP ::= TERM
EXP ::= TERM + TERM
EXP ::= TERM – TERM
TERM ::= FACTOR
TERM ::= FACTOR * FACTOR
TERM ::= FACTOR / FACTOR
FACTOR ::= num
FACTOR ::= (EXP)
}

Le regole *non avrebbero né ricorsione sinistra né destra,* e sarebbe sempre necessario usare le parentesi, anche quando di solito non le mettiamo.

Ad esempio, si dovrà scrivere:

```
(0111 + 0011) + 0010
```

in quanto questa sarebbe illecita:

```
0111 + 0011 + 0010
```

GRAMMATICA NON ASSOCIATIVA
ANALISI TOP-DOWN

VN = { EXP, TERM, FACTOR }
VT = { +, *. –, /, (,), num }

P = {
EXP ::= TERM
EXP ::= TERM + TERM
EXP ::= TERM – TERM
TERM ::= FACTOR
TERM ::= FACTOR * FACTOR
TERM ::= FACTOR / FACTOR
FACTOR ::= num
FACTOR ::= (EXP)
}

- La nuova grammatica a lato, *non ricorsiva né a sinistra né a destra,* è compatibile con l'analisi ricorsiva discendente.
- Il prezzo è più accettabile: l'unico obbligo è dover usare sempre le parentesi.
- MA.. si può davvero convincere un'intera cultura ad adottare una sintassi "verbosa" solo perché farebbe comodo a noi..?

Dovremo trovare un punto d'incontro più accettabile

Dalla grammatica al parser

INTERPRETAZIONE e ANALISI TOP-DOWN

- Il *riconoscitore* è un PDA, perché *la grammatica contiene self-embedding* (`FACTOR ::= (EXP)`)
- PROBLEMA: la sintassi concreta delle espressioni include produzioni *ricorsive a sinistra*

```
EXP    ::=  EXP + TERM        // plusexp
EXP    ::=  EXP - TERM        // minusexp
TERM   ::=  TERM * FACTOR     // timesexp
TERM   ::=  TERM / FACTOR     // divexp
```

- Così com'è, la grammatica non è LL(1)
  → l'analisi ricorsiva discendente *non* è *applicabile*

TRASFORMAZIONE DELLA GRAMMATICA

OBIETTIVO: ***riuscire a rendere la grammatica LL(1)***

- **Primo passo**: riscrittura in forma Extended BNF

```
EXP    ::=  TERM                    // caso base
EXP    ::=  EXP ( + | - ) TERM      // plusexp e minusexp
TERM   ::=  FACTOR                  // caso base
TERM   ::=  TERM ( * | / ) FACTOR   // timesexp e divexp
```

da cui

```
EXP    ::=    TERM  | EXP ( + | - ) TERM
TERM   ::=  FACTOR  | TERM ( * | / ) FACTOR
```

Ma queste sono ***produzioni regolari a sinistra***, della forma $X \rightarrow p \mid X\alpha$, che generano il linguaggio $L = p\{\alpha\}$
ossia ***p seguito da un qualsiasi numero di*** α

TRASFORMAZIONE DELLA GRAMMATICA

- **Secondo passo**: applicazione della trasformazione dalla forma $X \rightarrow p \mid X\alpha$ alla forma $p\{\alpha\}$

```
EXP    ::=  TERM { ( + | - ) TERM }
TERM   ::=  FACTOR { ( * | / ) FACTOR }
```

- La grammatica così ottenuta è LL(1)
  - EXP ha ora un'unica produzione, che inizia sempre con TERM
  - dopo il primo TERM, o non c'è niente (stringa vuota) oppure c'è uno dei due simboli + o -, nel qual caso segue un altro TERM
  - dopo il secondo TERM, idem… e così via
  - TERM ha un'unica produzione, che inizia per FACTOR
  - dopo il primo FACTOR, o non c'è niente (stringa vuota) o c'è uno dei due simboli * o /, nel qual caso segue un altro FACTOR
  - dopo il secondo FACTOR, idem… e così via

TRASFORMAZIONE DELLA GRAMMATICA

- Si può esplicitare quanto asserito scrivendo:

```
EXP          ::= TERM AFTERTERM
AFTERTERM    ::= ε | +EXP | -EXP
TERM         ::= FACTOR AFTERFACTOR
AFTERFACTOR  ::= ε | *TERM | /TERM
```

- Questa grammatica è chiaramente LL(1)…
  …ma è anche *ricorsiva a destra !*
  - Non va dunque utilizzata "operativamente" per derivare la frase (e costruire l'albero sintattico), perché l'associatività degli operatori risulterebbe culturalmente «sbagliata»
  - ma solo per *verificare che la grammatica precedente sia LL(1)*
- Il riconoscitore sarà implementato seguendo la grammatica EBNF precedente, che *esclude la ricorsione.*

DALLA GRAMMATICA EBNF AL RICONOSCITORE

- Perché la grammatica EBNF esclude la ricorsione?

```
EXP   ::= TERM { ( + | - ) TERM }
TERM  ::= FACTOR { ( * | / ) FACTOR }
```

- Perché *evidenzia l'essenza: siamo in presenza di una stringa ripetuta zero o più volte,* ossia di una *iterazione*
  - un processo computazionale iterativo è codificabile in un linguaggio di programmazione *senza far uso della ricorsione*
  - a ogni ciclo, se, dopo TERM o FACTOR, c'è la stringa vuota l'iterazione ha termine;
    altrimenti si prosegue con una ben precisa produzione, quella richiesta dal simbolo terminale (+,- o rispett. *,/) incontrato.

ANALISI TOP-DOWN: APPLICAZIONE PRATICA

- Analisi ricorsiva discendente (top down)
  - una procedura o funzione per ogni simbolo non terminale
  - invocazione ricorsiva solo per il caso con self-embedding (realizza il PDA riconoscitore per grammatica di tipo 2)
- Ogni funzione restituisce:
  - un *boolean*, nel caso di *puri riconoscitori*
  - un *opportuno valore/oggetto*, nel caso di *parser completi* che effettuino anche una valutazione (o meta-valutazione)
- Ogni funzione termina:
  - o quando *la stringa di input finisce*
  - o quando incontra un *simbolo non appartenente al suo sotto-linguaggio di pertinenza*

SCHEMA DI PARSER

Ogni funzione *analizza il sotto-linguaggio di pertinenza*

- `parseExp` analizza L(EXP),
  per il quale +, - e TERM sono l'alfabeto terminale
- `parseTerm` analizza L(TERM),
  per il quale *, / e FACTOR sono l'alfabeto terminale
- `parseFactor` analizza L(FACTOR),
  il cui alfabeto terminale è costituito da (,) ed EXP

PRIMO ESEMPIO: *puro riconoscitore*

- ogni funzione restituisce un *boolean*

SCHEMA DI PARSER (1/3)

Cerca una sequenza di TERMINI.
Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

```
boolean parseExp()
{ boolean nextTerm,termSeq = parseTerm();
  while (currentToken ≠ '\0')
  { if (currentToken == '+')       // caso Term + Term …
    { currentToken = nextToken();
      nextTerm = parseTerm();
      termSeq = termSeq && nextTerm;
    }
    else if (currentToken == '-') // caso Term - Term …
    { currentToken = nextToken();
      nextTerm = parseTerm();
      termSeq = termSeq && nextTerm;
    }
    else return termSeq;          // caso Exp ::= Term
  }
  return termSeq;                 // input terminato
}
```

SCHEMA DI PARSER (2/3)

Cerca una sequenza di FATTORI.
Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

```
boolean parseTerm()
{ boolean nextFactor,factSeq = parseFactor();
  while (currentToken ≠ '\0')
  { if (currentToken == '*') // caso Factor * Factor …
    { currentToken = nextToken();
      nextFactor = parseFactor();
      factSeq = factSeq && nextFactor;
    }
    else if (currentToken == '/') // caso Factor / Factor …
    { currentToken = nextToken();
      nextFactor = parseFactor();
      factSeq = factSeq && nextFactor;
    }
    else return factSeq;          // caso Term::= Factor
  }
  return factSeq;                 // input terminato
}
```

SCHEMA DI PARSER (3/3)

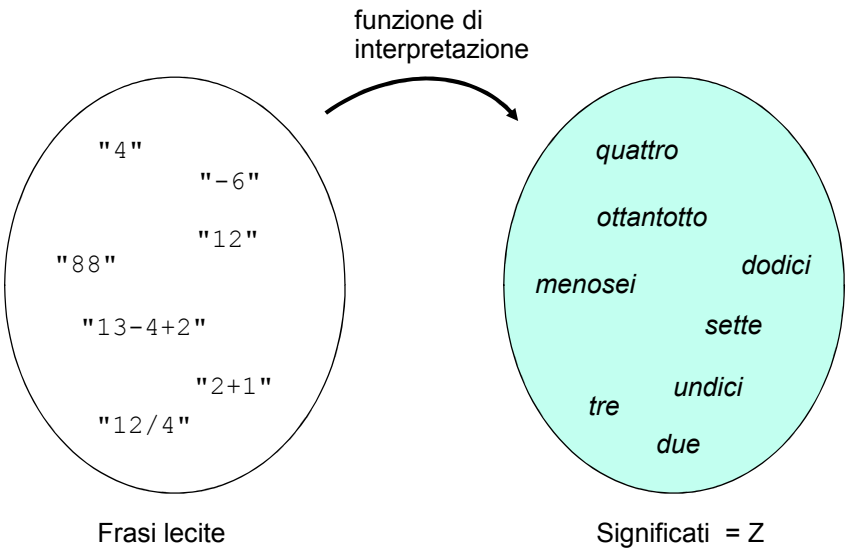
```
boolean parseFactor()
{ if (currentToken == '(') // caso Factor ::= ( Exp )
  { boolean innerExp;
    currentToken = nextToken();
    innerExp = parseExp();
    if (currentToken == ')')
    {  currentToken = nextToken();
       return innerExp;
    }
    else return false;
  }
  else if (isnumber(currentToken)) // caso Factor ::= num
  { currentToken = nextToken();
    return true;
  }
  else return false;
}
```

Dal riconoscitore al parser: specifica della semantica

SPECIFICARE LA SEMANTICA

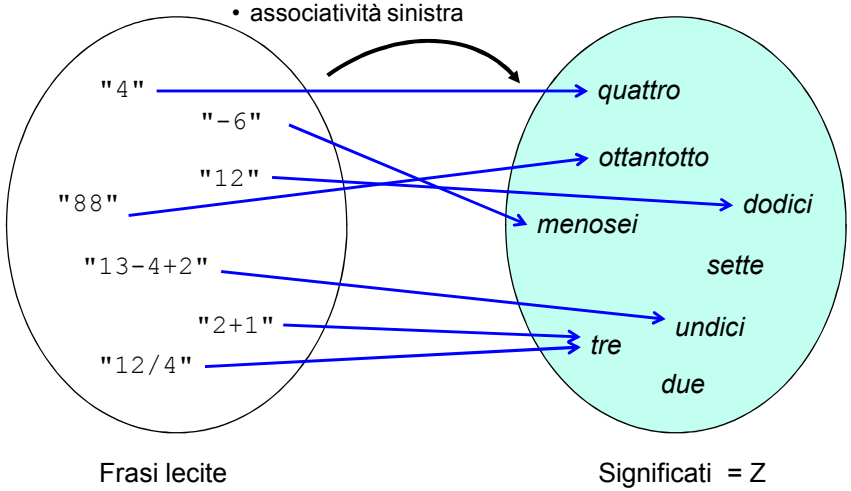
- **Occorre un *modo sistematico e formale* per stabilire *con precisione e senza ambiguità* il *SIGNIFICATO* di ogni possibile frase del linguaggio**
  - se il linguaggio è finito, basta un elenco
  - se è infinito, serve una *notazione finita* (applicabile a infinite frasi)
- **Un modo è definire una *funzione di interpretazione***
  - **DOMINIO: il linguaggio** (insieme delle frasi lecite, ossia *stringhe*)
  - **CODOMINIO: l'insieme dei possibili significati,** ossia l'insieme degli ***oggetti*** che si vogliono far corrispondere a tali frasi
  - ad esempio, per le espressioni sugli interi, il codominio è Z
- **Come definire tale funzione?**
  - se il linguaggio è finito, basta una tabella (stringhe → significati)
  - altrimenti, serve una *funzione* definita in modo *ricorsivo*

SPECIFICARE LA SEMANTICA (2)



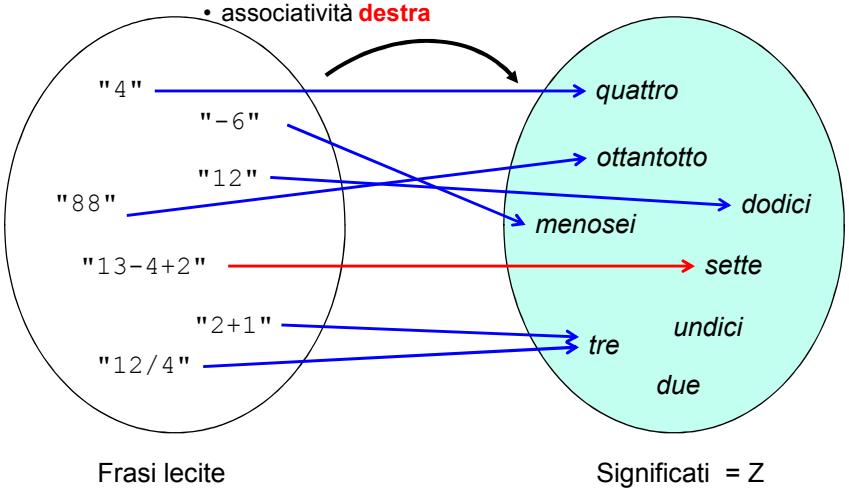
SPECIFICARE LA SEMANTICA (3)

Funzione di interpretazione TERRESTRE
- notazione posizionale decimale
- associatività sinistra



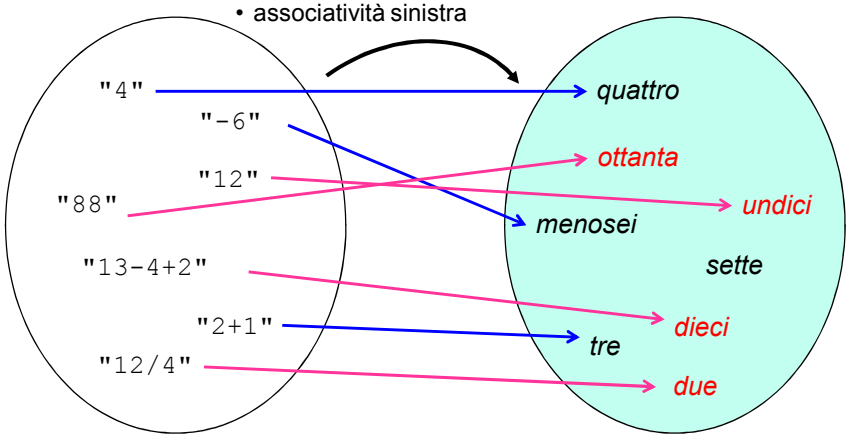
SPECIFICARE LA SEMANTICA (4)

Funzione di interpretazione TERRESTRE BIS
- notazione posizionale decimale
- associatività **destra**



SPECIFICARE LA SEMANTICA (5)

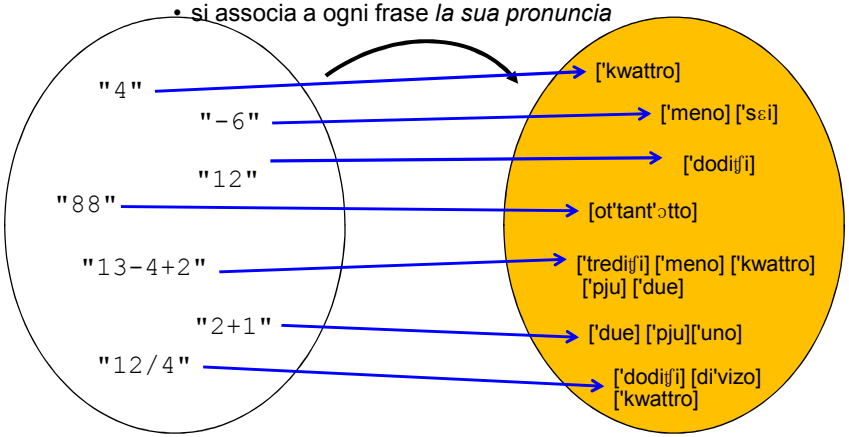
Funzione di interpretazione MARZIANA
- marziani con 9 dita → base 9
- associatività sinistra



Frasi lecite — Significati = Z

SPECIFICARE LA SEMANTICA (6)

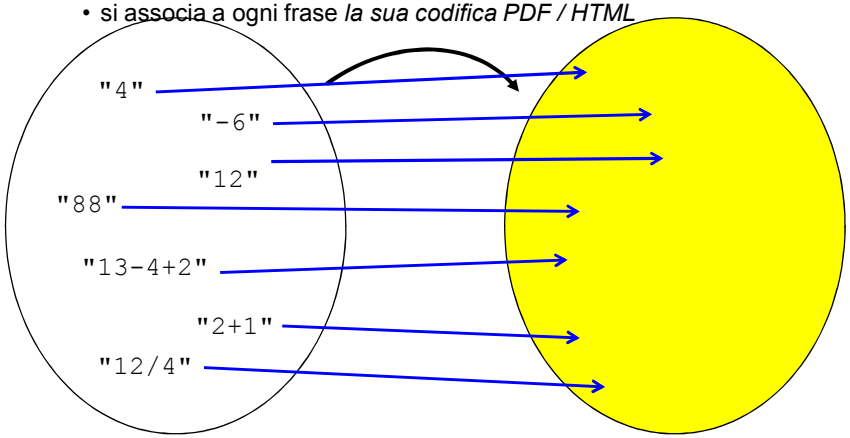
Funzione di interpretazione per SINTESI VOCALE
- **cambia il codominio: non più Z, ma *fonetica***
- si associa a ogni frase *la sua pronuncia*



Frasi lecite — Significati = *trascrizione fonetica*

SPECIFICARE LA SEMANTICA (7)

Funzione di interpretazione per GENERAZIONE PDF/HTML
- **cambia il codominio: non più Z, ma *codici PDF o HTML***
- si associa a ogni frase *la sua codifica PDF / HTML*



Frasi lecite — Significati = *codici PDF*

SEMANTICA DENOTAZIONALE

Quando la semantica di un linguaggio è espressa in questo modo si parla di semantica denotazionale

- **ESPRESSIONI ARITMETICHE: SEMANTICA FORMALE**
  - linguaggio infinito → *semantica definita tramite funzione ricorsiva*
- ***COME DEFINIRLA?***
  - potremmo inventarci una funzione di interpretazione qualunque..
  - *..ma poi sarebbe complicato esplicitare la corrispondenza con le singole frasi* (che, essendo infinite, sono generate dalla grammatica)
- **IDEA FURBA: seguire pari pari la sintassi!**
  - per ogni regola sintattica, *una regola semantica!*
  - non si rischiano dimenticanze, mapping pulito e chiaro da leggere
  - nel nostro caso la sintassi prevede **Exp, Term** e **Factor**
    → la semantica prevederà tre funzioni f_{Exp}, f_{Term} e f_{Factor}

SEMANTICA DENOTAZIONALE PER LA GRAMMATICA "STANDARD"

- **Si specifica un comportamento *per ogni possibile struttura di frase*.**
- Ad esempio, se un'espressione ha la forma `Exp + Term`, la funzione di interpretazione da eseguire sarà:

  f_{Expr}(exp+term)=
  f_{Expr} (exp) + f_{Term} (term)

  che stabilisce che il valore dell'espressione che compare prima del simbolo + debba essere ***sommato*** *(perché questo è il significato dell'operazione + nel dominio Z)* al valore del termine che compare dopo il +.
- **Essendo un termine definito come fattore o prodotto di fattori, eventuali moltiplicazioni presenti nella espressione verranno eseguite *prima* delle somme.**

f_{Expr} (term) = f_{Term} (term)
f_{Expr} (exp + term) = f_{Expr} (exp) + f_{Term} (term)
f_{Expr} (exp - term) = f_{Expr} (exp) - f_{Term} (term)
f_{Term} (factor) = f_{Factor} (factor)
f_{Term} (term * factor) = f_{Term} (term) × f_{Factor} (factor)

$$f_{Term}(term/factor) = \frac{f_{Term}(term)}{f_{Factor}(factor)}$$

f_{Factor} ((exp)) = f_{Expr} (exp)
f_{Factor} (num) = ***valueof*** (num)

+, -, *, /, (,) :
(in rosso): simboli sintattici della grammatica

+, -, *, /, (,) :
(in nero): operazioni "note" sul dominio N

SEMANTICA DENOTAZIONALE ESEMPIO CONCRETO (1/3)

Espressione: `3 + 4 * 18 / (7 – 1)`

L'espressione ha la forma `Exp + Term`, cui corrisponde l'interpretazione:

f_{Expr} (exp + term) = f_{Expr} (exp) + f_{Term} (term)

ossia, nel caso in esame:

f_{Expr} (3 + 4 * 18 / (7 – 1)) = f_{Expr} (3) + f_{Term} (4 * 18 / (7 – 1))

L'espressione `3` ha la forma `Term`, cui corrisponde l'interpretazione:

f_{Expr} (term) = f_{Term} (term)

ossia f_{Expr} (3) = f_{Term} (3)

Questo nuovo termine `3` ha la forma `Factor`, da cui l'interpretazione:

f_{Term} (3) = f_{Factor} (3) = ***valueof*** (3) = ***tre***

Il termine `4 * 18 / (7 – 1)` ha la forma `Term / Factor`, da cui:

$$f_{Term}(4 * 18 / (7 - 1)) = \frac{f_{Term}(4 * 18)}{f_{Factor}(\ (7 - 1)\ )}$$

SEMANTICA DENOTAZIONALE ESEMPIO CONCRETO (2/3)

`4 * 18` / `(7 – 1)`
`Term` `Factor`

Ora, il termine `4 * 18` ha la forma `Term * Factor`, da cui:

f_{Term} (4 * 18) = f_{Term} (4) × f_{Factor} (18)

Questo nuovo termine `4` ha la forma `Factor`, da cui l'interpretazione:

f_{Term} (4) = f_{Factor} (4) = ***valueof*** (4) = ***quattro***

Analogamente, il fattore `18` ha la forma ***num***, da cui l'interpretazione:

f_{Factor} (18) = ***valueof*** (18) = ***diciotto***

Ergo, il termine `4 * 18` denota nel complesso il valore ***quattro*** × ***diciotto***

Qui, il fattore `(7-1)` ha la forma `(Exp)`, da cui:

f_{Factor} ((7 – 1)) = f_{Expr} (7 – 1)

da cui, nell'ordine:

f_{Expr} (7 – 1) = f_{Expr} (7) – f_{Term} (1) =
= f_{Term} (7) – f_{Term} (1) =
= f_{Factor} (7) – f_{Factor} (1) =
= ***valueof*** (7) – ***valueof*** (1) =
= ***sette*** – ***uno***

SEMANTICA DENOTAZIONALE ESEMPIO CONCRETO (3/3)

Espressione: `3 + 4 * 18 / (7 – 1)`

Significato:

tre + f_{Term} (4 * 18 / (7 – 1))
f_{Term} (4 * 18) / f_{Factor} ((7 – 1))
f_{Term} (4) × f_{Factor} (18) f_{Expr} (7 – 1)
quattro × ***diciotto*** ***sette*** – ***uno***
settantadue / ***sei***
dodici
tre + ***dodici***
quindici

SCHEMA DI INTERPRETE

Ogni funzione *analizza il sotto-linguaggio di pertinenza*

- `parseExp` analizza L(EXP),
  per il quale `+`, `-` e TERM sono l'alfabeto terminale
- `parseTerm` analizza L(TERM),
  per il quale `*`, `/` e FACTOR sono l'alfabeto terminale
- `parseFactor` analizza L(FACTOR),
  il cui alfabeto terminale è costituito da `(`, `)` ed EXP

2° ESEMPIO: *interprete con valutazione immediata*

- ogni funzione restituisce un *int* o un *double*
- dipende dal *dominio di interpretazione* che si sceglie

SCHEMA DI INTERPRETE (1/3)

Cerca una sequenza di TERMINI.
Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

```
int parseExp()
{ int nextTerm,termSeq = parseTerm();
  while (currentToken ≠ '\0')
  { if (currentToken == '+')        // caso Term + Term …
    { currentToken = nextToken();
      nextTerm = parseTerm();
      termSeq = termSeq + nextTerm;
    }
    else if (currentToken == '-') // caso Term - Term …
    { currentToken = nextToken();
      nextTerm = parseTerm();
      termSeq = termSeq - nextTerm;
    }
    else return termSeq;           // caso Exp ::= Term
  }
  return termSeq;                  // input terminato
}
```

SCHEMA DI PARSER (2/3)

Cerca una sequenza di FATTORI.
Si ferma quando trova un token non pertinente al suo sotto-linguaggio o quando la stringa di input termina

```
int parseTerm()
{ int nextFactor,factSeq = parseFactor();
  while (currentToken ≠ '\0')
  { if (currentToken == '*') // caso Factor * Factor …
    { currentToken = nextToken();
      nextFactor = parseFactor();
      factSeq = factSeq * nextFactor;
    }
    else if (currentToken == '/') // caso Factor / Factor …
    { currentToken = nextToken();
      nextFactor = parseFactor();
      factSeq = factSeq / nextFactor;
    }
    else return factSeq;           // caso Term::= Factor
  }
  return factSeq;                  // input terminato
}
```

SCHEMA DI PARSER (3/3)

```
int parseFactor()
{ if (currentToken == '(')  // caso Factor ::= ( Exp )
  { int innerExp;
    currentToken = nextToken();
    innerExp = parseExp();
    if (currentToken == ')')
    {  currentToken = nextToken();
       return innerExp;
    }
    else error();
  } else if (isnumber(currentToken)) // caso Factor ::= num
  { int val = valueof(currentToken);
    currentToken = nextToken();
    return val;
  }
  else error();
}
```

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi