UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 6
Dai riconoscitori ai traduttori

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

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

QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

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..

… AI TRADUTTORI

Un *traduttore* 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"!***

TRADUTTORE: 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 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 }

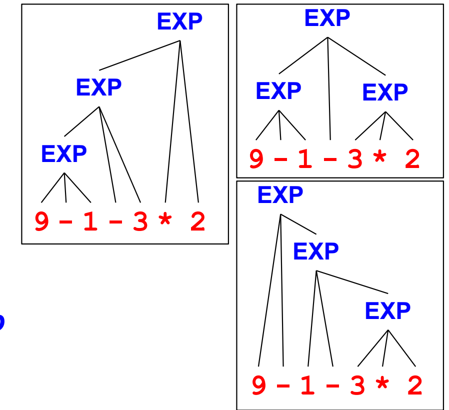
S = EXP

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.***

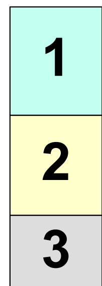
VN = { EXP, TERM, FACTOR }

VT = { +, *. –, /, (,), num }

P = {

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

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

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)**

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

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**

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

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)**

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

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**

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

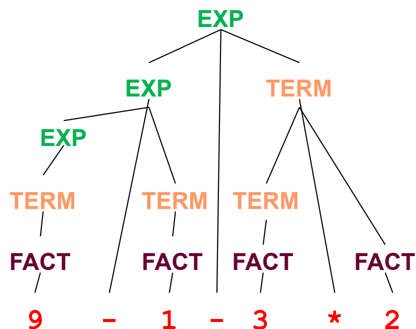
ASSOCIATIVITÀ DEGLI OPERATORI

- **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**

Regole	Associatività
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)

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!**
- ***Come fare?***
  - **se eliminiamo la ricorsione a 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!*

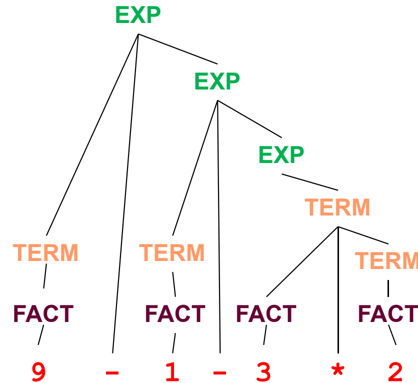
VARIANTE 1 – ASSOCIATIVA A DESTRA

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!*



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 a sinistra né a 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

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 6.1
Dalla grammatica al parser

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

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*

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*

DALLA GRAMMATICA EBNF AL RICONOSCITORE

- Riscriviamo la grammatica in EBNF:

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

- Questa versione *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 c'è + o - , si prosegue con la corrispondente produzione
  - altrimenti, si considera che ci sia la stringa vuota e l'iterazione ha termine.
  - Analogamente per FACTOR (usando * o /).

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)

```
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
}
```

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

SCHEMA DI PARSER (2/3)

```
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
}
```

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

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;
}
```

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 6.2
Dal riconoscitore al parser: specifica della semantica

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

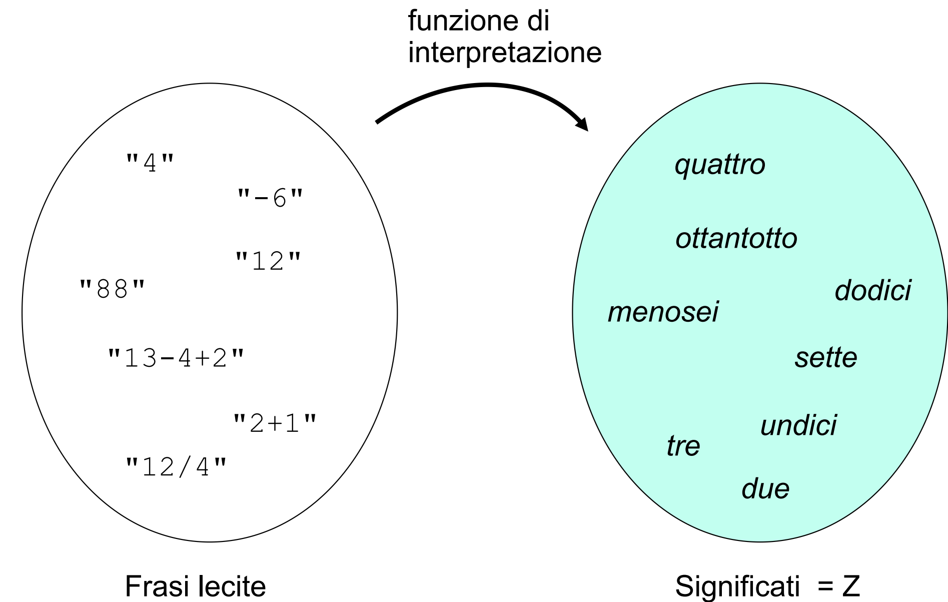
Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

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)



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) = $\dfrac{f_{Term}\text{(term)}}{f_{Factor}\text{(factor)}}$

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 (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

UNIVERSITÀ DEGLI STUDI DI FERRARA

- EX LABORE FRUCTUS -

Capitolo 6.2

Interprete per le espressioni

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

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

QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

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)

```
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
}
```

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

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();
}
```

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 6.2.1
Interprete per le espressioni Haskell

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

Schema in Haskell

EXP ::= TERM { (+ | -) TERM }

```
expr :: String -> (Int,String)
expr xs =
    let (val,rest) = term xs
    in  termList val rest

termList val ('+':xs) =
    let (val1,rest) = term xs
    in  termList (val+val1) rest
termList val ('-':xs) =
    let (val1,rest) = term xs
    in  termList (val-val1) rest
termList val xs = (val,xs)
```

Schema in Haskell

```
factor ('(':xs) =
     let (val,rest) = expr xs
     in if head rest == ')'
          then (val,tail rest)
          else (val,('(':xs))
factor (x:xs) = (estraiValore(x), xs)
```

FACTOR ::= (EXP)

FACTOR ::= num

UNIVERSITÀ DEGLI STUDI DI FERRARA

- EX LABORE FRUCTUS -

Capitolo 6.3
Costruzione dell'albero sintattico

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

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

QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

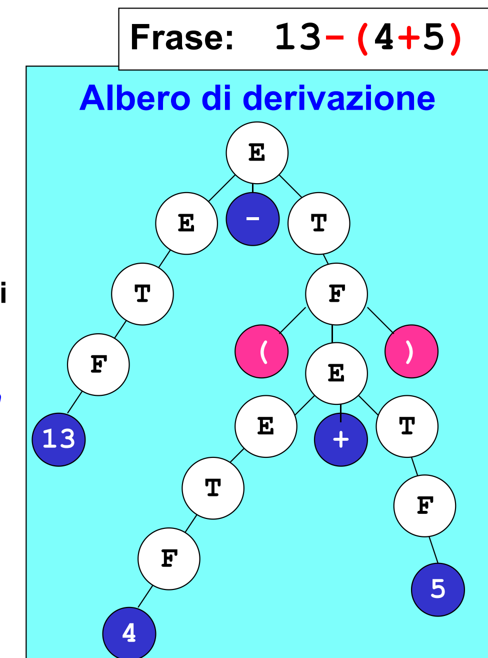
RAPPRESENTAZIONE DELLE FRASI

- **La grammatica descrive la *struttura effettiva delle frasi*, ossia la *sintassi concreta* del linguaggio.**
  - **Tale grammatica è studiata solitamente in modo che il linguaggio risulti non solo *ben definito*, ma anche *chiaro per chi legge***
  - **La sintassi concreta include quindi elementi (punteggiatura, parole chiave, ..) introdotti spesso non perché realmente necessari, ma per *motivi di chiarezza e leggibilità* delle frasi.**
- **Se la valutazione non è immediata, il parser rappresenta le frasi sintatticamente corrette tramite una opportuna *rappresentazione interna ad albero.***

ALBERI SINTATTICI

Si potrebbe usare *l'albero di derivazione,* ma in generale esso darebbe luogo a una rappresentazione *ridondante*

- **l'albero di derivazione illustra tutti i singoli passi di derivazione, ma molti di essi servono solo durante la costruzione dell'albero, ossia *per ottenere proprio "quell'albero" e non un altro***
- **inoltre, un albero con molti nodi e livelli è *complesso da visitare* (la visita è ricorsiva), determinando quindi inutili *inefficienze***



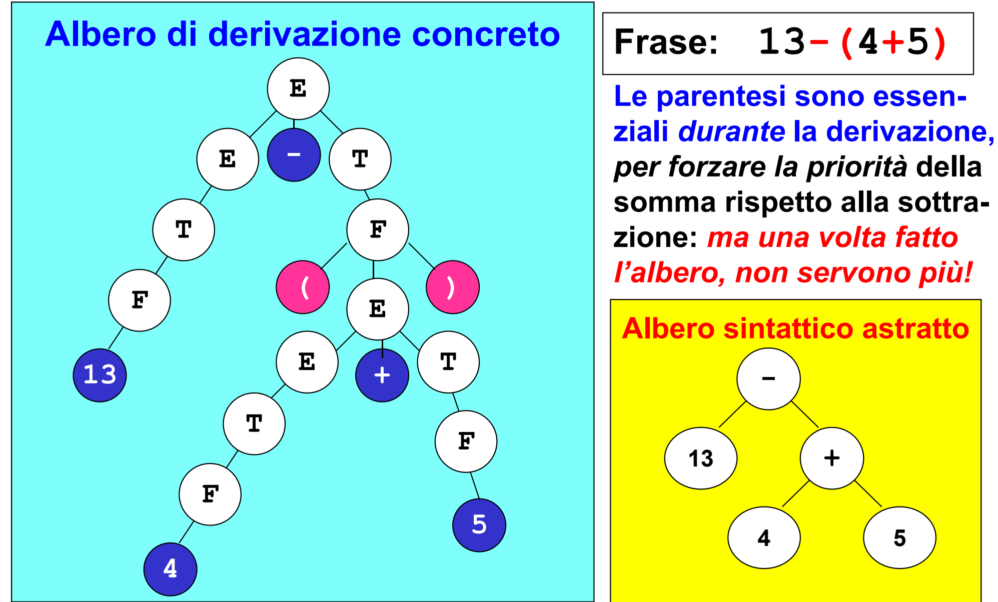
ALBERI SINTATTICI ASTRATTI (1)

- Conviene adottare un *albero più compatto,* che contenga *solo i nodi indispensabili* e possibilmente sia *binario.*
- Tale albero è detto *Abstract Parse Tree (APT)* o *Abstract Syntax Tree (AST).*

Quali sono i *nodi non indispensabili ?*

- i nodi *terminali* (foglie) non legati ad alcunché di significativo sul piano semantico
  - **segni di punteggiatura,** zucchero sintattico in genere
  - *qui non ne abbiamo*
- i nodi *terminali* (foglie) *che, pur utili durante la costruzione dell'albero per ottenere "quell'albero e non un altro", esauriscono con ciò la loro funzione*
  - nel caso delle espressioni: **parentesi tonde**
- i nodi *non terminali che hanno un unico nodo figlio*
  - nel caso delle espressioni: **sequenze Exp → Term → Factor**

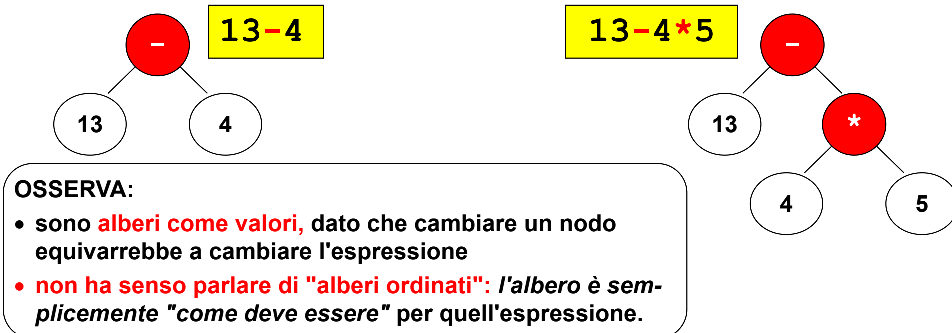
ESEMPIO



ESPRESSIONI & ALBERI ASTRATTI

Nel caso delle espressioni, l'AST è così definito:

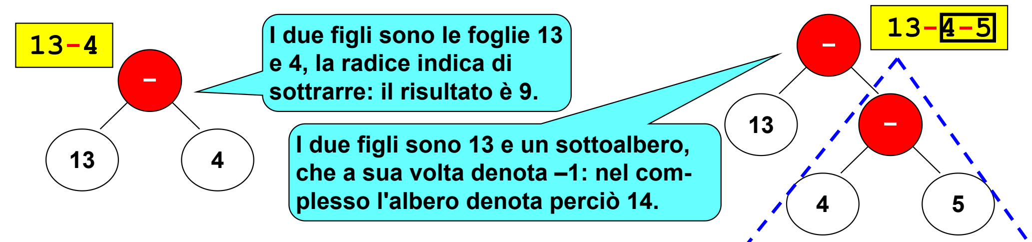
- **ogni operatore è un nodo con due figli:**
  - il figlio sinistro è il primo operando
  - il figlio destro è il secondo operando
- **i valori numerici sono le foglie.**



ESPRESSIONI E ALBERI

Così, la rappresentazione è univoca:

- una espressione è rappresentabile in *un solo modo, senza ambiguità*
- **la struttura dell'albero fornisce intrinsecamente *l'ordine corretto di valutazione***
  - è impossibile valutare un nodo senza disporre prima dei due figli
  - quindi, occorre PER FORZA valutare prima la parte "in basso"
  - si risale fino a valutare la radice, che fornisce il risultato



AST in Haskell

```
data AST =
     Sum AST AST |  Diff AST AST |
     Mul AST AST |  Div AST AST |
     Val Int
```

- O anche, considerando un tipo generico:

```
data AST a =
     Sum (AST a) (AST a) |
     Diff (AST a) (AST a) |
     Mul (AST a) (AST a) |
     Div (AST a) (AST a) |
     Val a
```

Schema in Haskell

```
data AST =  Sum AST AST |     Diff AST AST |
      Mul AST AST |     Div AST AST |    Val Int
expr xs =
    let (val,rest) = term xs
    in  termList val rest

termList val ('+':xs) =
    let (val1,rest) = term xs
    in  termList (Sum val val1) rest
termList val ('-':xs) =
    let (val1,rest) = term xs
    in  termList (Diff val val1) rest
termList val xs = (val,xs)
```

Schema in Haskell

```
data AST =  Sum AST AST |     Diff AST AST |
      Mul AST AST |     Div AST AST |    Val Int


factor ('(':xs) =
    let (val,rest) = expr xs
    in if head rest == ')'
        then (val,tail rest)
        else (val,('(':xs))
factor (x:xs) = ( Val  estraiValore(x), xs)
```

```
> expr "9-5-1"
(Diff (Diff (Val 9) (Val 5)) (Val 1),"")
```

Haskell: higher order

A questo punto, è anche possibile implementarla come funzione di ordine superiore

```
expr xs sum diff ... =
    let (val,rest) = term xs sum diff ...
    in  termList val rest sum diff ...

termList val ('+':xs) sum diff ...=
    let (val1,rest) = term xs sum diff ...
    in  termList (sum val val1) rest sum diff ...
termList val ('-':xs) sum diff ... =
    let (val1,rest) = term xs sum diff ...
    in  termList (diff valval1) rest sum diff ...
termList val xs = (val,xs)
```

Haskell Higher Order

```
factor ('(':xs) sum diff conv ... =
    let (val,rest) = expr xs sum diff ...
    in if head rest == ')'
        then (val,tail rest)
        else (val,('(':xs))
factor (x:xs) = (conv x, xs)
```

```
> expr "9-5-1" (+) (-) leggiInt
(3,"")

> expr "9-5-1" Sum Diff (\x -> Val (leggiInt x))
(Diff (Diff (Val 9) (Val 5)) (Val 1),"")
```

UNIVERSITÀ
DEGLI STUDI
DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 6.4
Valutazione dell'AST

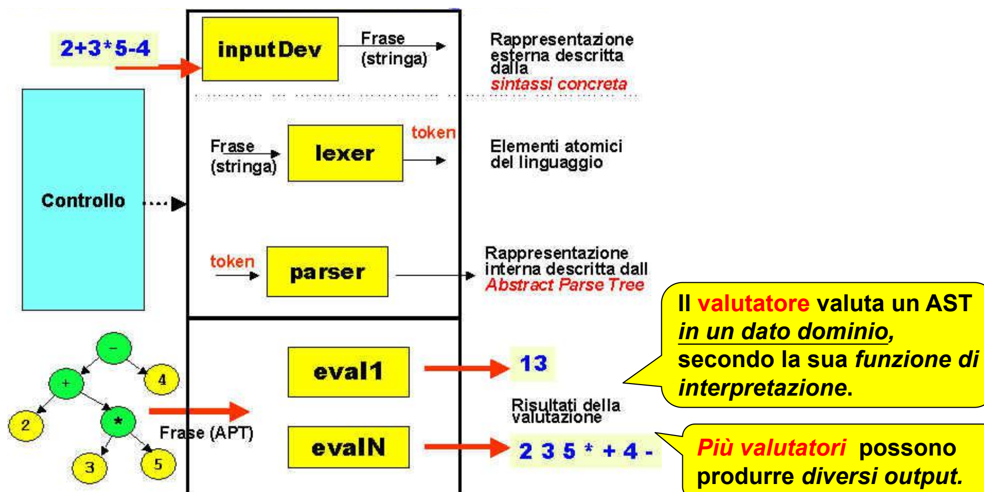
Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

ARCHITETTURA



VALUTARE ALBERI

- **Assodato che l'albero sintattico astratto sia il modo più compatto per rappresentare un'espressione.. *come lo valutiamo?***
- **La teoria degli alberi introduce il concetto di *VISITA***
  - ***Pre-order:*** *radice, figli (da sinistra a destra)*
  - ***Post-order:*** *figli (da sinistra a destra), radice*
  - ***In-order:*** *figlio sinistro, radice, figlio destro*
- **Occorre capire cosa producono i diversi tipi di visita**
  - ***Pre-order:*** *operatore, 1° operando, 2° operando*
  - ***Post-order:*** *1° operando, 2° operando, operatore*
  - ***In-order:*** *1° operando, operatore, 2° operando*

VALUTARE ALBERI

- **Assodato che l'albero sintattico astratto sia il modo più compatto per rappresentare un'espressione..** ***come lo valutiamo?***
- **La teoria degli alberi introduce il concetto di** ***VISITA***
  - ***Pre-order:*** ***radice, figli (da sinistra a destra)***
  - ***Post-order:*** ***figli (da sinistra a destra), radice***
  - ***In-order:*** ***figlio sinistro, radice, figlio destro***
- **Occorre capire cosa producono i diversi tipi di visita**
  - ***Pre-order:*** **notazione PREFISSA**
  - ***Post-order:*** **notazione POSTFISSA**
  - ***In-order:*** **notazione INFISSA classica**

OLTRE LA NOTAZIONE INFISSA

- **L'uso della notazione infissa è un aspetto culturale**
  - ci siamo abituati, la conosciamo da sempre, al punto di pensare che sia l'unica "sensata" o financo "possibile", ma…
  - .. in realtà, è *solo uno* dei modi per rappresentare espressioni *e neanche il più felice!*
- **In effetti,** ***è lo scrivere l'operatore in mezzo agli operandi che rende necessarie priorità, associatività e parentesi!***
  - **È la notazione infissa a creare ambiguità nell'ordine di esecuzione delle operazioni!** `9-4-1 = ?`
  - **Adottando ad esempio una classica notazione funzionale, il problema non si pone!** Nell'esempio a lato, `f` denota la sottrazione. `f(9,f(4,1))` `f(f(9,4),1)`
  - ***OSSERVA: parentesi e virgole qui sono puramente estetiche!***

OLTRE LA NOTAZIONE INFISSA

- **L'uso della notazione infissa è un aspetto culturale**
  - ci siamo abituati, la conosciamo da sempre, al punto di pensare che sia l'unica "sensata" o financo "possibile", ma…
  - .. in realtà, è *solo uno* dei modi per rappresentare espressioni *e neanche il più felice!*
- **In effetti,** ***è lo scrivere l'operatore in mezzo agli operandi che rende necessarie priorità, associatività e parentesi!***
  - **È la notazione infissa a creare ambiguità nell'ordine di esecuzione delle operazioni!** `9-4-1 =`
  - **Adottando ad esempio una classica notazione funzionale, il problema non si pone!** Nell'esempio a lato, `f` denota la sottrazione. `f 9 f 4 1` `f f 9 4 1` `- - 9 4 1`
  - ***Parentesi e virgole estetiche RIMOSSE!***

NOTAZIONI PREFISSE E POSTFISSE

- **Due alternative sono la** ***notazione prefissa*** **o** ***postfissa***
  - **NOTAZIONE PREFISSA: prima l'operatore, poi gli operandi (è la tipica notazione funzionale)**
  - **NOTAZIONE POSTFISSA: prima gli operandi, poi l'operatore**
- **Non richiedono di definire alcuna nozione di** ***priorità e associatività – e quindi neppure parentesi –*** **perché non c'è ambiguità sull'ordine di esecuzione delle operazioni**
  - notazione infissa `9-4-1`
  - in notazione *prefissa*, ogni operatore si applica sempre ai due operandi che lo *seguono* `- - 9 4 1` `f f 9 4 1`
  - in notazione *postfissa*, ogni operatore si applica sempre ai due operandi che lo *precedono* `9 4 - 1 -`

NOTAZIONI PREFISSE E POSTFISSE

- Due alternative sono la *notazione prefissa* o *postfissa*
  - NOTAZIONE PREFISSA: **prima l'operatore, poi gli operandi (è la tipica notazione funzionale)**
  - NOTAZIONE POSTFISSA: **prima gli operandi, poi l'operatore**
- **Non richiedono di definire alcuna nozione di *priorità e associatività – e quindi neppure parentesi* – perché non c'è ambiguità sull'ordine di esecuzione delle operazioni**
  - notazione infissa `9-(4-1)`
  - in notazione *prefissa*, ogni operatore si applica sempre ai due operandi che lo *seguono* `- 9 - 4 1` / `f 9 f 4 1`
  - in notazione *postfissa*, ogni operatore si applica sempre ai due operandi che lo *precedono* `9 4 1 - -`

VISITE DI ALBERI-ESPRESSIONE

Se un'espressione è un albero, cosa si ottiene visitandolo nei diversi modi?

- la visita pre-order dà luogo alla notazione PREFISSA
- la visita in-order dà luogo alla notazione INFISSA
- la visita post-order dà luogo alla notazione POSTFISSA

VISITE DI ALBERI-ESPRESSIONE

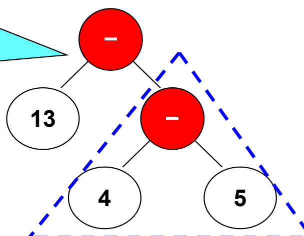
Se un'espressione è un albero, cosa si ottiene visitandolo nei diversi modi?

- la visita pre-order dà luogo alla notazione PREFISSA
- la visita in-order dà luogo alla notazione INFISSA
- la visita post-order dà luogo alla notazione POSTFISSA

ATTENZIONE ALLA NOTAZIONE INFISSA!
Questa espressione dovrebbe essere scritta come `13-(4-5)`, perché secondo le nostre usuali convenzioni `13-4-5` è un'altra cosa!!

PREFISSA: `- 13 - 4 5`
INFISSA: `13 - 4 - 5`
POSTFISSA: `13 4 5 - -`

ATTENZIONE ALLA NOTAZIONE INFISSA!
Non evidenziando il "livello" (tramite parentesi o altri artifici), l'algoritmo può dar luogo a frasi che, *secondo le usuali convenzioni, useremmo per un'espressione diversa!*



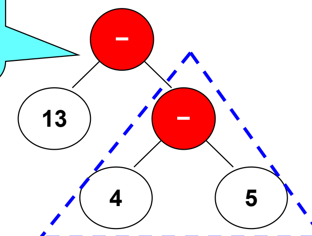
VISITE DI ALBERI-ESPRESSIONE

Se un'espressione è un albero, cosa si ottiene visitandolo nei diversi modi?

- la visita pre-order dà luogo alla notazione PREFISSA
- la visita in-order dà luogo alla notazione INFISSA
- la visita post-order dà luogo alla notazione POSTFISSA

PREFISSA: `- 13 - 4 5`
INFISSA: `(13-(4-5))`
POSTFISSA: `5 - -`

Occorre modificare l'algoritmo di visita in-order in modo che INTRODUCA UN'INDICAZIONE DEL LIVELLO VISITATO, evitando così di "appiattire" l'albero e perdere informazione → *PARENTESI*



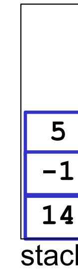
Notazione postfissa

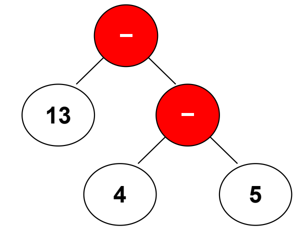
- Chiamata anche Notazione Polacca Inversa (Reverse Polish Notation o RPN)
- La valutazione di un'espressione può essere effettuata tramite uno stack
  - Ogni volta in cui il prossimo simbolo dell'input è un valore, viene inserito (push) nello stack
  - Ogni volta in cui il simbolo è un operatore (con *n* operandi) si fa una pop di *n* valori, si applica l'operatore e si inserisce il risultato nello stack (push)

Esempio RPN

infissa: `13 - (4-5)`

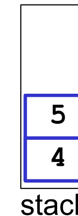
RPN: `13 4 5 - -`

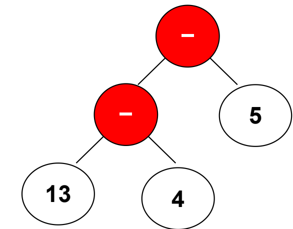




infissa: `13 - 4 - 5`

RPN: `13 4 - 5 -`





NOTAZIONE POSTFISSA.. e oltre

- ***La notazione POSTFISSA è adattissima a un elaboratore*** **perché fornisce prima gli operandi**
  - **che possono così essere caricati nei registri del processore o in altre zone opportune di memoria, pronti per l'ALU**

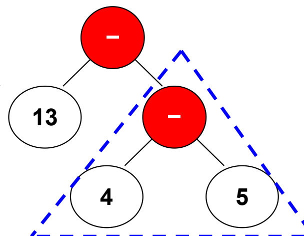
e solo dopo "comanda" l'esecuzione dell'operazione.

- **Lo stesso principio è utilizzato anche nei compilatori:**
  - **ogni nodo-operatore viene tradotto nell'operazione assembler**
  - **ogni nodo-valore viene tradotto nel caricamento di tale valore in un registro macchina**

POSTFISSA: `13 4 5 - -`

CODICE MACCHINA:

```
load 13, r1
load  4, r2
load  5, r3
sub  r2, r3
sub  r1, r2
```



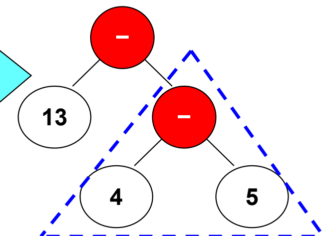
LA MACCHINA A STACK

- **E se i registri non bastano?**
- **E se non si vuole preoccuparsi di quali registri usare per i vari operandi?**
- **Si può usare una macchina a stack!**
  - **ogni nodo-valore carica un valore sullo stack (PUSH)**
  - **ogni nodo-operatore causa il prelievo di due valori dallo stack (POP) e il collocamento sullo stack del risultato (PUSH)**
  - **alla fine si preleva il risultato dallo stack (POP)**

POSTFISSA: `13 4 5 - -`

CODICE MACCHINA:

```
push 13
push  4
push  5
sub  [include 2 pop+1push]
sub  [include 2 pop+1push]
[segue pop  finale]
```

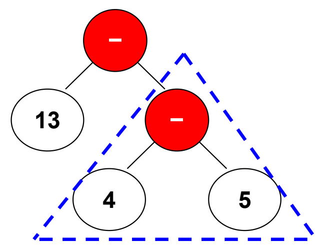


LA MACCHINA A STACK: ESEMPIO

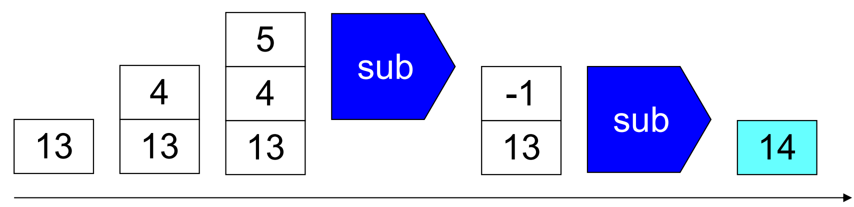
POSTFISSA:
`13 4 5 - -`

CODICE MACCHINA:

```
push 13
push  4
push  5
sub  [include 2 pop+1push]
sub  [include 2 pop+1push]
[segue pop  finale]
```



Evoluzione dello stack nel tempo:



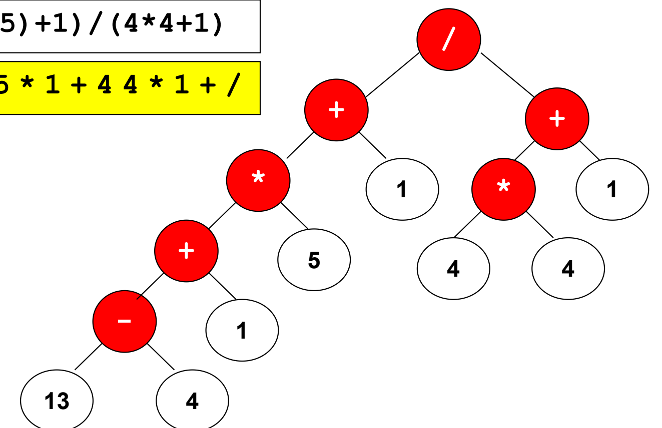
UN ALTRO ESEMPIO

INFISSA: `(((13-4)+1)*5)+1)/(4*4+1)`

POSTFISSA: `13 4 - 1 + 5 * 1 + 4 4 * 1 + /`

CODICE MACCHINA:

```
push 13
push  4
sub
push  1
sum
push  5
mul
push  1
sum
push  4
push  4
mul
push  1
sum
div
[pop  finale]
```



Evoluzione dello stack: