Costruzione dell'albero sintattico

RAPPRESENTAZIONE DELLE FRASI

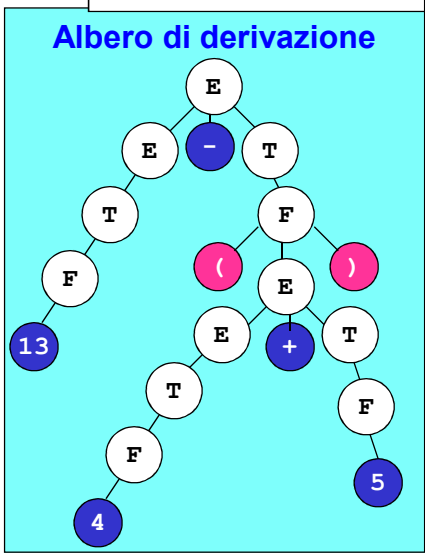
- La grammatica descrive la ***struttura effettiva delle frasi***, ossia la ***sintassi concreta*** del linguaggio.
  - Tale grammatica è studiata solitamente in modo che il linguaggio risulti non solo *ben definito*, ma anche ***chiaro per chi legge***
  - La sintassi concreta include quindi elementi (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*

Frase: `13-(4+5)`



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).*

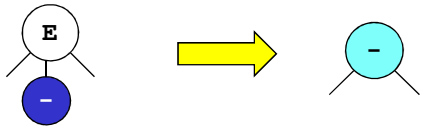
Quali sono i *nodi non indispensabili ?*

- 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**

ALBERI SINTATTICI ASTRATTI (2)

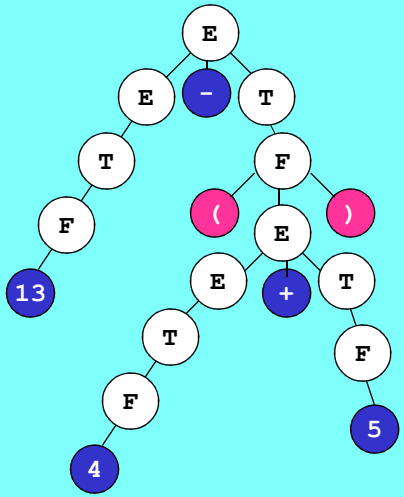
Come rendere eventualmente l'albero *binario*?

- **inserendo informazioni come *proprietà del nodo-padre anziché come sotto-nodo figlio***
  - nelle espressioni: **spostando "in su" i simboli degli operatori**
  - analogo XML: rappresentare una informazione come *proprietà* di un nodo anziché come *sotto-elemento*



ESEMPIO

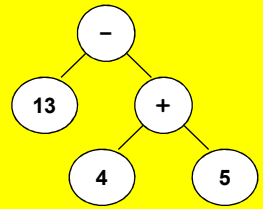
Albero di derivazione concreto



Frase: `13-(4+5)`

Le parentesi sono essenziali *durante* la derivazione, *per forzare la priorità* della somma rispetto alla sottrazione: *ma una volta fatto l'albero, non servono più!*

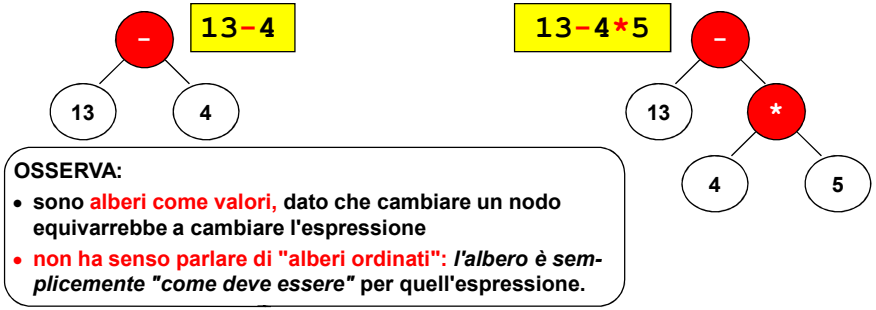
Albero sintattico astratto



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.**

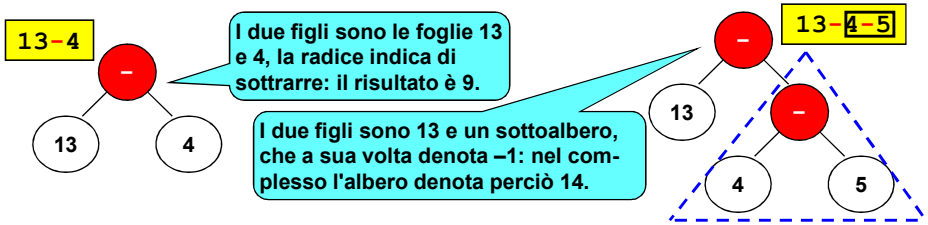


OSSERVA:
- **sono alberi come valori, dato che cambiare un nodo equivarrebbe a cambiare l'espressione**
- **non ha senso parlare di "alberi ordinati": *l'albero è semplicemente "come deve essere"* per quell'espressione.**

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**



ANALISI TOP-DOWN

OBIETTIVO: *estendere il parser* facendogli generare un opportuno APT

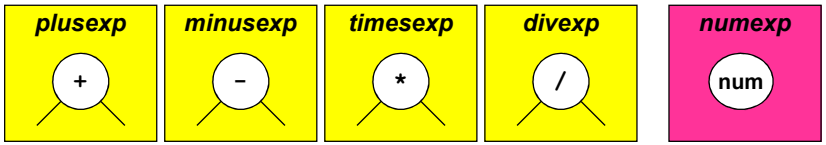
- **Analisi ricorsiva discendente (top down)**
- **Ogni funzione restituisce:**
  - **NEL 1° CASO: un *boolean*** *(puro riconoscitore)*
  - **NEL 2° CASO: un *intero*** *(valutazione immediata)*
  - **ORA: un *opportuno valore/oggetto* che descriva un APT**

Quale oggetto?
Di che classe?
C'è una tassonomia?

TIPI DI NODI DELL'APT

- **Per far generare al parser un opportuno APT, occorre prima stabilire come dev'essere fatto, ossia *quali e quanti tipi diversi di nodo dovrà avere.***
  - **Serve un nodo diverso per ogni possibile tipo di espressione**
  - **Ci sono le 4 operazioni + le costanti numeriche = 5 tipi di nodo**

- Una possibile scelta potrebbe essere questa:



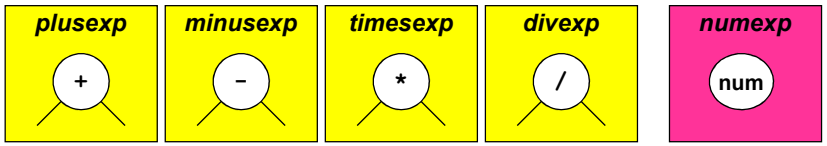
Però, una serie di «figurine» non è una descrizione formale..

SINTASSI ASTRATTA

- **Una *sintassi astratta* ha precisamente l'obiettivo di descrivere come è fatto l'Abstract Syntax Tree (AST)**
  - **Non è destinata all'utente: descrive una rappresentazione interna**
  - **Ogni produzione indica *un possibile nodo,* ossia un possibile *modo di costruire quel "tipo" di espressione***

I cinque tipi di nodo introdotti poco fa possono essere definiti dalla sintassi astratta illustrata a lato:

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



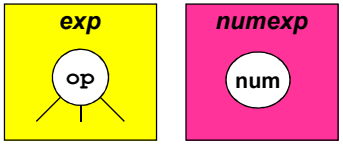
SINTASSI ASTRATTA

- **La sintassi astratta *non* è *unica***
  - **Poiché descrive una possibile rappresentazione interna, varia al variare di quest'ultima**
  - **Sintassi astratte diverse descrivono *sistemi software diversi***

La sintassi astratta a lato descrive un sistema software basato su un *diverso insieme di nodi*

- **Il mattoncino Exp riassume in un unico tipo di nodo, *disponibile in 4 versioni*, 4 precedenti tipi diversi di nodo.**
- **esso potrebbe essere realizzato o come nodo binario (con op memorizzato come proprietà nel nodo) o ternario (con op come terzo figlio)**

```
EXP ::= EXP OP EXP
EXP ::= num
OP ::= + | - | * | /
```



UNA POSSIBILE IMPLEMENTAZIONE

Seguendo la metodologia indicata si possono introdurre le seguenti classi Java:

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

```
abstract class Exp {} // la classe-base
```

```
abstract class OpExp extends Exp {
  Exp left, right; // due campi, tanti quanti i sottocomp.
  protected OpExp( Exp l, Exp r){ left=l; right=r;}
  abstract String myOp();
  public String toString(){
    return left.toString() + myOp() + right.toString();
  }
}
...
```

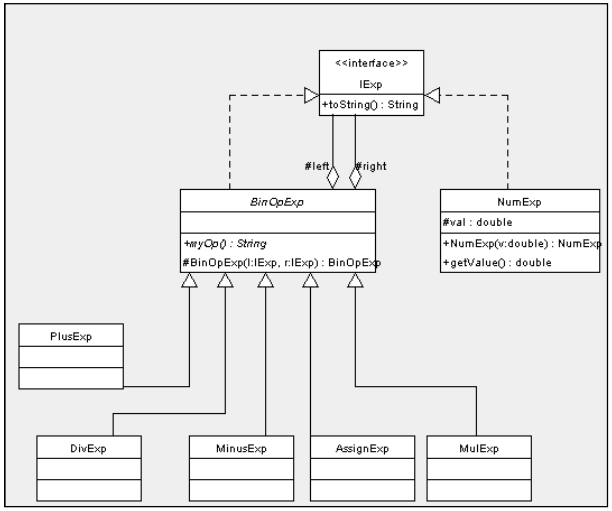
UNA POSSIBILE IMPLEMENTAZIONE

Seguendo la metodologia indicata si possono introdurre le seguenti classi Java:

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

```
class PlusExp extends OpExp {
  public PlusExp( Exp l, Exp r) {super(l,r);}
  public String myOp() { return "+" ; }
}
class MinusExp extends OpExp {
  public MinusExp( Exp l, Exp r) {super(l,r);}
  public String myOp() { return "-" ; }
}
class NumExp extends Exp {
  int val;
  public NumExp(int v) { val = v; }
  public String toString() {return "" + val;}
  public int getValue() { return val; }
}
```

MODELLO DEL SISTEMA SOFTWARE



VERIFICA DEL MODELLO (1)

Ogni frase del linguaggio deve quindi poter essere rappresentata da un APT componendo in modo opportuno istanze di queste classi, *simulando ciò che farà il parser.*

```
Exp s1 = new MinusExp(
           new NumExp(3), new NumExp(5)  );

Exp s2 = new MinusExp(
           new PlusExp(
                 new NumExp(2),
                 new TimesExp(
                       new NumExp(5), new NumExp(4) )
           ),
           new NumExp( 1 ) );

Exp s3 = new MinusExp( s1,
           new TimesExp(
                 new NumExp(1),new NumExp(5) )
           );
```

```
3-5

2+5*4-1

3-5-1*5
```

VERIFICA DEL MODELLO (2)

La costruzione dell'APT tiene conto della priorità degli operatori

- la frase **s5** dà priorità all'operatore di sottrazione più a sinistra
- la frase **s4** dà priorità all'operatore di sottrazione più a destra
- l'APT è diverso nei due casi, ma **toString** genera la stessa stringa rendendoli *indistinguibili* → non va bene, dovremo provvedere

```
Exp s4 = new MinusExp(
           new NumExp(5),
           new MinusExp(
                new NumExp(3), new NumExp(1) )
           );

Exp s5 = new MinusExp(
           new MinusExp(
                new NumExp(5), new NumExp(3) ),
           new NumExp(1)
           );
```

```
5-3-1

5-3-1
```

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

NUOVO ESEMPIO: *parser che genera un APT*

- **ogni funzione restituisce *non più* un boolean o un int, ma bensì un *oggetto di classe Exp***

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

```
public Exp parseExp() {
 Exp termSeq = parseTerm();
 while (currentToken != null){
   if (currentToken.equals("+")) {
      currentToken = scanner.getNextToken();
      Exp nextTerm = parseTerm();
      if (nextTerm != null)        // costruzione APT a sinistra
         termSeq = new PlusExp(termSeq, nextTerm);
      else return null;
   }
   else if (currentToken.equals("-")) {
      currentToken = scanner.getNextToken();
      Exp nextTerm = parseTerm();
      if (nextTerm != null)        // costruzione APT a sinistra
         termSeq = new MinusExp(termSeq, nextTerm);
      else return null;
   }
   else return termSeq;         // next token non fa parte di L(Exp)
 } // end while
 return termSeq;   // next token è nullo -> stringa di input finita
}
```

Alternativa: lanciare eccezione

Alternativa: lanciare eccezione

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

```
public Exp parseTerm() {
 Exp factorSeq = parseFactor();
 while (currentToken != null){
   if (currentToken.equals("*")) {
      currentToken = scanner.getNextToken();
      Exp nextFactor = parseFactor();
      if (nextFactor != null)      // costruzione APT a sinistra
         factorSeq = new TimesExp(factorSeq, nextFactor);
      else return null;
   } else
   if (currentToken.equals("/")) {
      currentToken = scanner.getNextToken();
      Exp nextFactor = parseFactor();
      if (nextFactor != null)      // costruzione APT a sinistra
         factorSeq = new DivExp(factorSeq, nextFactor);
      else return null;
   }
   else return factorSeq;       // next token non fa parte di L(Term)
 } // end while
 return factorSeq;  // next token è nullo -> stringa di input finita
}
```

Alternativa: lanciare eccezione

Alternativa: lanciare eccezione

SCHEMA DI PARSER (3/3)

Cerca una SINGOLO FATTORE.
Se trova un token non pertinente al suo sotto-linguaggio, restituisce null per segnalare il problema

```
public Exp parseFactor() {
  if (currentToken.equals("(")) {
     currentToken = scanner.getNextToken();
     Exp innerExp = parseExp();     // PDA: gestione self-embedding
     if (currentToken.equals(")")) {
           currentToken = scanner.getNextToken();
           return innerExp;          // parentesi vengono omesse
     } else return null;
  }
  else                  // dev'essere un numero
  if (currentToken.isNumber()) {
     int value = currentToken.getAsInt();
     currentToken = scanner.getNextToken();
     return new NumExp(value);
  }
  else                  // errore: non è un fattore
     return null;       // non si è costruito nulla, restituiamo null
}
```

Alternativa: lanciare eccezione

Alternativa: lanciare eccezione

ARCHITETTURA: SET-UP e USO

Un cliente dovrà dunque:

- **creare l'istanza di scanner** *(qui, specifica per una stringa)*
- **creare l'istanza di parser che usa tale scanner**
- **invocare `parseExp` per riconoscere la frase *ottenendo come risultato l'APT***

```
public class TestInterpreteAPT {
  public static void main(String args[]) {
     String expression = "( 3 - 1 ) * ( 4 - 5 )";
     MyScanner scanner = new MyScanner(expression);
     MyParser parser = new MyParser(scanner);
     Exp apt = parser.parseExp();
     System.out.println(apt); // APT da valutare poi
  }
}
```

AST: VARIANTE (1/2)

Partendo dalle produzioni a lato, si sarebbe ottenuto un *diverso sistema software:*

```
EXP  ::= EXP BINOP EXP
EXP  ::= num
BINOP ::= + | - | * | /
```

```
abstract class Exp {} // la classe-base
```

```
class OpExp extends Exp { // non è più abstract
  Exp left, right; BinOp op; // TRE campi
  protected OpExp( Exp l, BinOp op, Exp r){
        left=l; this.op=op; right=r;}
  public String toString(){ return left.toString() +
        op.toString() + right.toString();
  }
}
abstract class BinOp {
  abstract String myOp();
  public String toString(){ return myOp(); }
}
```

AST: VARIANTE (2/2)

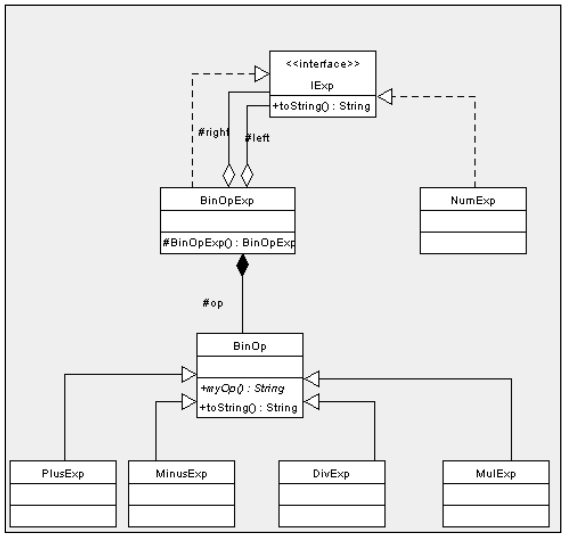
Partendo dalle produzioni a lato, si sarebbe ottenuto un *diverso sistema software:*

```
EXP  ::= EXP BINOP EXP
EXP  ::= num
BINOP ::= + | - | * | /
```

```
class NumExp extends Exp {                  // resta come prima
  int val;
  public NumExp(int v) { val = v; }
  public String toString() {return "" + val;}
  public int getValue() { return val; }
}
class PlusOp extends BinOp {
  public String myOp(){ return "+" ; }
}
...
```

Questa versione non categorizza le espressioni in "espressioni somme", "espressioni prodotto", ecc.; riconosce invece *un'unica categoria OpExp*, e delega ai componenti BinOp il compito di precisare l'operatore.

MODELLO
DEL NUOVO SISTEMA SOFTWARE



Verso la valutazione dell'albero sintattico astratto

ALBERI SINTATTICI ASTRATTI & SINTASSI ASTRATTA

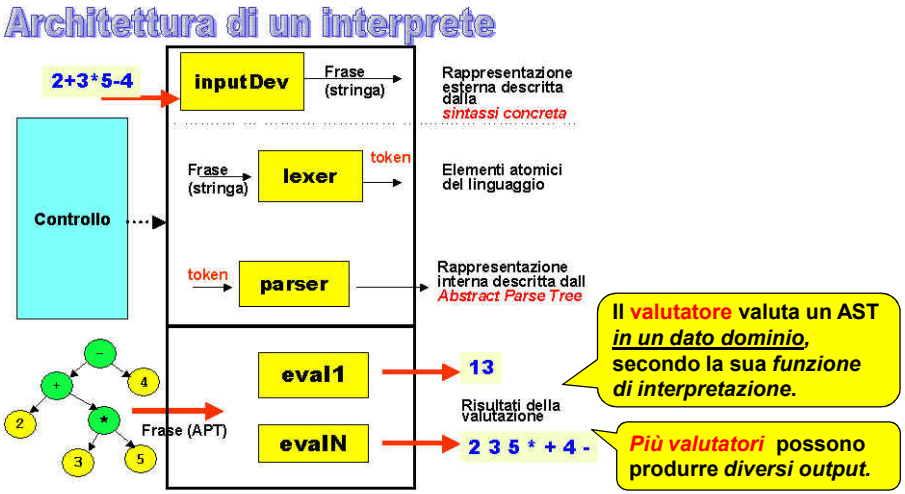
In definitiva, dunque, l'AST:

- ***non rispecchia più, strutturalmente, la sintassi concreta del linguaggio***
- **riflette invece una *sintassi astratta* invisibile all'utente e all'esterno, *utile solo a specificare il formato interno* usato dall'interprete.**

Conseguenza: *la sintassi astratta può essere ambigua*

- **infatti, non serve a guidare il riconoscitore…**
- ***…ma solo a descrivere come è fatto l'AST !***

ARCHITETTURA DI UN INTERPRETE



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 - -`

NOTAZIONE PREFISSA: priorità

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 - 4 - 1`	`9 - (4 - 1)`
PREFISSA:	`+ 3 * 4 5`	`* + 3 4 5`	`- - 9 4 1`	`- 9 - 4 1`

L'operatore `+` si applica ai due operandi `3` e `* 4 5`.
La sottoespressione `* 4 5` evidenzia l'operatore `*` applicato agli operandi `4` e `5`.

Secondo le usuali interpretazioni dei simboli, la sottoespressione `* 4 5` denota il valore *venti.* L'operatore `+` applicato ai valori *tre* e *venti* denota quindi il risultato finale *ventitre.*

Qui, invece, è l'operatore `*` che si applica agli operandi `+ 3 4` e `5`.
La sottoespressione `+ 3 4` indica ovviamente l'operatore `+` applicato a `3` e `4`.

La sottoespressione `+ 3 4` denota *sette,* ergo l'operatore `*` applicato ai due valori *sette* e *cinque* denota il valore finale *trentacinque.*

NOTAZIONE PREFISSA: associatività

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 - 4 - 1`	`9 - (4 - 1)`
PREFISSA:	`+ 3 * 4 5`	`* + 3 4 5`	`- - 9 4 1`	`- 9 - 4 1`

Il primo operatore `-` si applica agli operandi `- 9 4` e `1`.
La sottoespressione `- 9 4` evidenzia l'operatore `-` applicato agli operandi `9` e `4`.

Secondo le usuali interpretazioni dei simboli, la sottoespressione `- 9 4` denota il valore *cinque.* Perciò, il primo operatore `-`, appli-cato ai valori *cinque* e *uno,* denota il risultato finale *quattro.*

Qui, invece, il primo operatore `-` si applica agli operandi `9` e `- 4 1`.
La sottoespressione `- 4 1` indica ovviamente l'operatore `-` applicato a `4` e `1`.

La sottoespressione `- 4 1` denota *tre,* ergo il primo operatore `-` applicato ai due valori *nove* e *tre* dà come risultato *sei.*

NOTAZIONE POSTFISSA: priorità

INFISSA:	3 + 4 * 5	(3 + 4) * 5	9 – 4 – 1	9 – (4 – 1)
POSTFISSA:	3 4 5 * +	3 4 + 5 *	9 4 – 1 –	9 4 1 – –

Analogamente, ma dualmente rispetto a prima, il primo operatore * si applica qui agli operandi 4 e 5, mentre il successivo operatore + si applica agli operandi 3 e 4 5 *.

La sottoespressione 4 5 * denota il valore *venti,* perciò, il successivo operatore + applicato ai due valori *tre* e *venti* dà come risultato *ventitre.*

Qui il primo operatore + si applica agli operandi 3 e 4, mentre il successivo operatore * si applica ai due operandi 3 4 + e 5.

Poiché la sottoespressione 3 4 + denota il valore *sette,* il successivo operatore * applicato ai due valori *sette* e *cinque* denota come risultato *trentacinque.*

NOTAZIONE POSTFISSA: associatività

INFISSA:	3 + 4 * 5	(3 + 4) * 5	9 – 4 – 1	9 – (4 – 1)
POSTFISSA:	3 4 5 * +	3 4 + 5 *	9 4 – 1 –	9 4 1 – –

Il primo operatore – si applica agli operandi – 9 4, mentre il secondo – si applica ai due operandi – 9 4 e 1.

La sottoespressione – 9 4 denota perciò, secondo le usuali interpretazioni dei simboli, il valore *cinque.* Perciò, il successivo operatore – applicato ai due valori *cinque* e *uno* denota il risultato *quattro.*

Qui, invece, il primo operatore – si applica agli operandi 4 e 1, mentre il secondo operatore – si applica agli operandi 9 e 4 1 –.

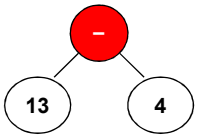
La sottoespressione 4 1 – denota *tre,* ergo il successivo operatore – applicato ai valori *nove* e *tre* deno-ta come risultato *sei.*

VISITE DI ALBERI-ESPRESSIONE

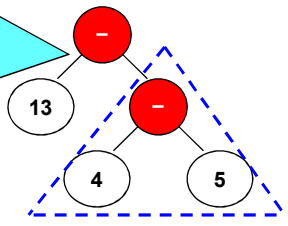
Se un'espressione è un albero, cosa si ottiene visitandolo nei diversi modi?

- 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
INFISSA: 13 – 4
POSTFISSA: 13 4 –



PREFISSA: – 13 R
INFISSA: 13 – R
POSTFISSA: 13 R –
dove R è il sottoalbero:
PREFISSA(R): – 4 5
INFISSA(R): 4 – 5
POSTFISSA(R): 4 5 –



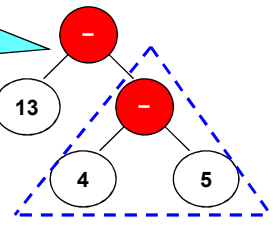
VISITE DI ALBERI-ESPRESSIONE

Se un'espressione è un albero, cosa si ottiene visitandolo nei d

- la visita pre-order
- la visita in-order dà luogo alla notazione A
- la visita post-order dà luogo alla notazi OSTFISSA

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!*

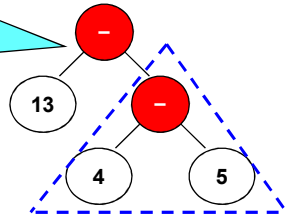
VISITE DI ALBERI-ESPRESSIONE

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*



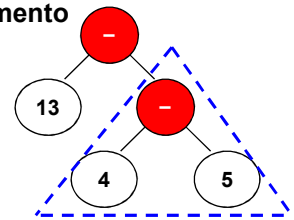
NOTAZIONE POSTFISSA.. e oltre

- *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
```

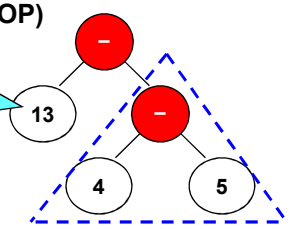


LA MACCHINA A STACK

- 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 - -

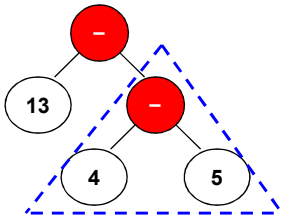
CODICE MACCHINA:
```
push 13
push  4
push  5
sub  [include 2 pop+1push]
sub  [include 2 pop+1push]
[segue pop  finale]
```



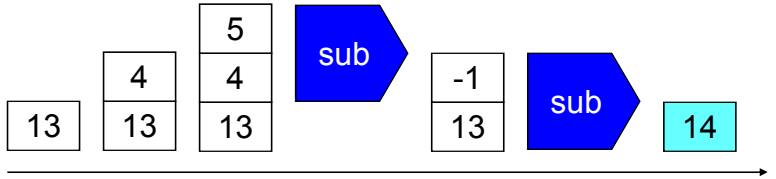
LA MACCHINA A STACK: ESEMPIO

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:



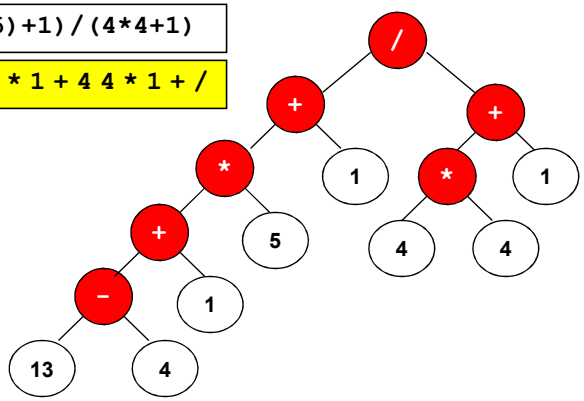
UN ALTRO ESEMPIO

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

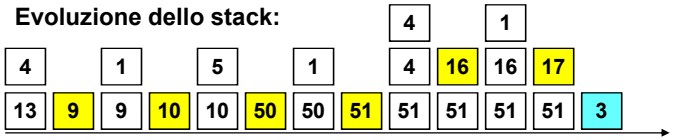
POSTFISSA: `13 4 - 1 + 5 * 1 + 4 4 * 1 + /`

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:



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