Capitolo 8
L'interprete esteso: assegnamenti, ambienti, sequenze

Corso di Laurea Magistrale in Ingegneria Informatica

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

ASSEGNAMENTO (1)

- Tutti i linguaggi di programmazione introducono a un qualche livello le nozioni di *variabile* e *assegnamento*
- Sebbene spesso si utilizzi l'operatore **=**, *l'assegnamento non è un'uguaglianza o equazione in senso matematico*
  - non è simmetrico né riflessivo: `x=25` è *molto diverso* da `25=x`
  - per questo alcuni linguaggi, come Pascal, utilizzano un simbolo di operatore "direzionale", che renda evidente la non riflessività (`:=`)
  - questa difformità è anche una delle principali ragioni che rendono difficile l'apprendimento iniziale dell'informatica in senso tradizionale

MOLTO DIVERSO!!

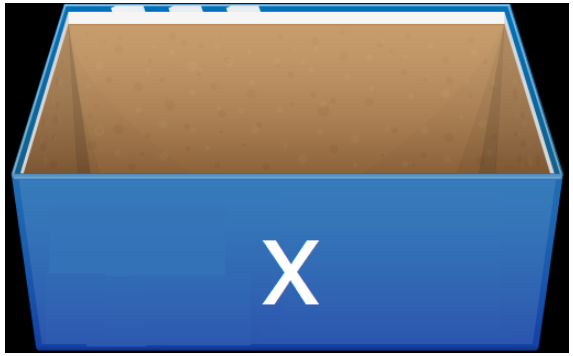
- Lo si vede molto bene nella scrittura

```
x = x + 1
```

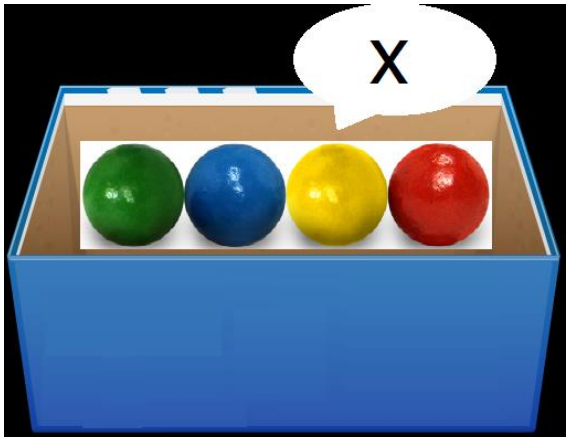
che come uguaglianza matematica sarebbe impossibile!

ASSEGNAMENTO (2)

- La peculiarità dell'assegnamento è che il simbolo di variabile ha *significato diverso* a destra e a sinistra dell'operatore =
- Il simbolo di variabile è *overloaded*
  - il suo significato dipende dalla *posizione* rispetto all'operatore
  - a sinistra, indica il *contenitore* a destra, il *contenuto*



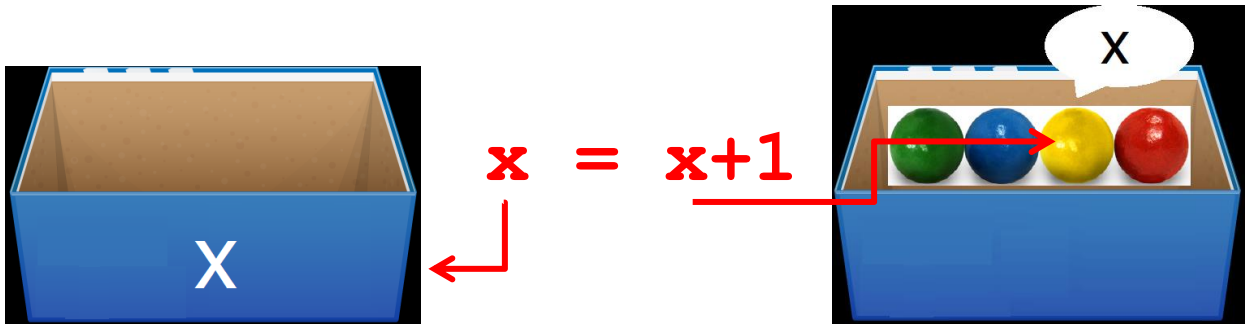
x = ...



... = x

ASSEGNAMENTO (3)

- La semantica informale può quindi essere così riassunta:
  1. Prendere il ***valore della variabile*** a destra dell'operatore =
  2. Usarlo per valutare il valore dell'espressione a destra dell' =
  3. Porre il risultato nella ***variabile*** specificata a sinistra dell' =
- Il simbolo di variabile significa *quindi*
  - "la variabile in quanto tale", quando compare a sinistra
  - "il contenuto di quella variabile", quando compare a destra



L-VALUE vs. R-VALUE

- Per questo si distingue fra *L-value* e *R-value*
  - L-value *(valore di sinistra):* il nome di variabile indica *la variabile in quanto tale* (tipicamente, la cella di memoria associata)
  - R-value *(valore di destra):* il nome di variabile indica *il contenuto della variabile* (ossia, il valore associato a quel simbolo)
- Linguaggi diversi possono fare scelte diverse in merito a varie questioni chiave:
  - assegnamento distruttivo / non distruttivo: si può cambiare il valore associato in precedenza a un simbolo di variabile?
  - è utile/opportuno distinguere sintatticamente L-value da R-value? Nei linguaggi di shell spesso si distingue: `x = $x+1`

L-VALUE vs. R-VALUE

- Difficoltà a capire «cosa fa» un programma semplicemente leggendone il testo
- MOTIVO: occorre simulare mentalmente l'evoluzione

- TRASPARENZA REFERENZIALE: un simbolo ha sempre lo stesso significato ovunque
- Pensa se le dimostrazioni matematiche non lo facessero..

- *della variabile* (ossia, il valore associato a quel simbolo)

Linguaggi imperativi

Linguaggi logici

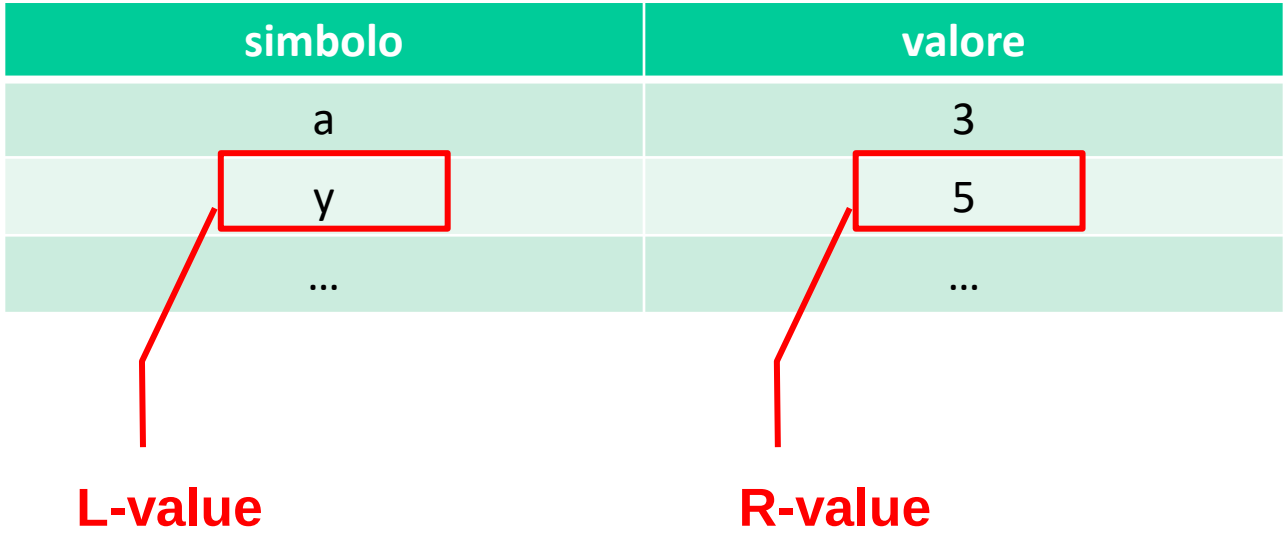
- a

- assegnamento distruttivo / non distruttivo: si può cambiare il valore associato in precedenza a un simbolo di variabile?
- è utile/opportuno distinguere sintatticamente L-value da R-value? Nei linguaggi di shell spesso si distingue: `x = $x+1`

ENVIRONMENT

- Per esprimere la semantica dell'assegnamento occorre introdurre il concetto di *environment*
  inteso come *insieme di coppie (simbolo, valore)*

  → una tabella a due colonne (mappa)



ASSEGNAMENTO: SEMANTICA (1)

- L'assegnamento modifica l'environment causando un *effetto collaterale* secondo la seguente semantica:

`x = valore`

1. se *non* è già presente una coppia con primo elemento **`x`**, si inserisce nell'environment la coppia (**`x`**, **`valore`**)

Esempio: x = 22

simbolo	valore
a	3
y	5
x	**22**
…	…

ASSEGNAMENTO: SEMANTICA (2a)

- L'assegnamento modifica l'environment causando un *effetto collaterale* secondo la seguente semantica:

`x = valore`

2. se invece esiste già una coppia (**`x`**, *v*), vi sono due possibilità:
   - *assegnamento distruttivo*: essa viene *eliminata* e sostituita dalla nuova coppia (**`x`**, **`valore`**)

Esempio: x = 6

simbolo	valore
a	3
y	5
x	~~22~~ **6**
…	…

ASSEGNAMENTO: SEMANTICA (2b)

- L'assegnamento modifica l'environment causando un *effetto collaterale* secondo la seguente semantica:

`x = valore`

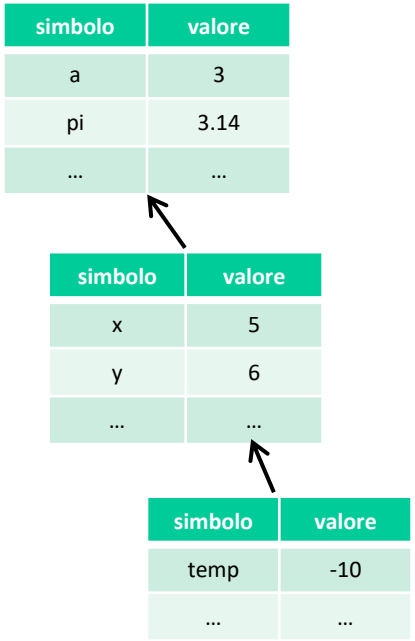
2. se invece esiste già una coppia (**`x`**, *v*), vi sono due possibilità:
   - *singolo assegnamento*: essa viene *mantenuta* e il tentativo di nuovo assegnamento a **`x`** dà luogo a ***errore***.

Esempio: x = 6 → Errore, rifiutato

simbolo	valore
a	3
y	5
x	**22**
…	…

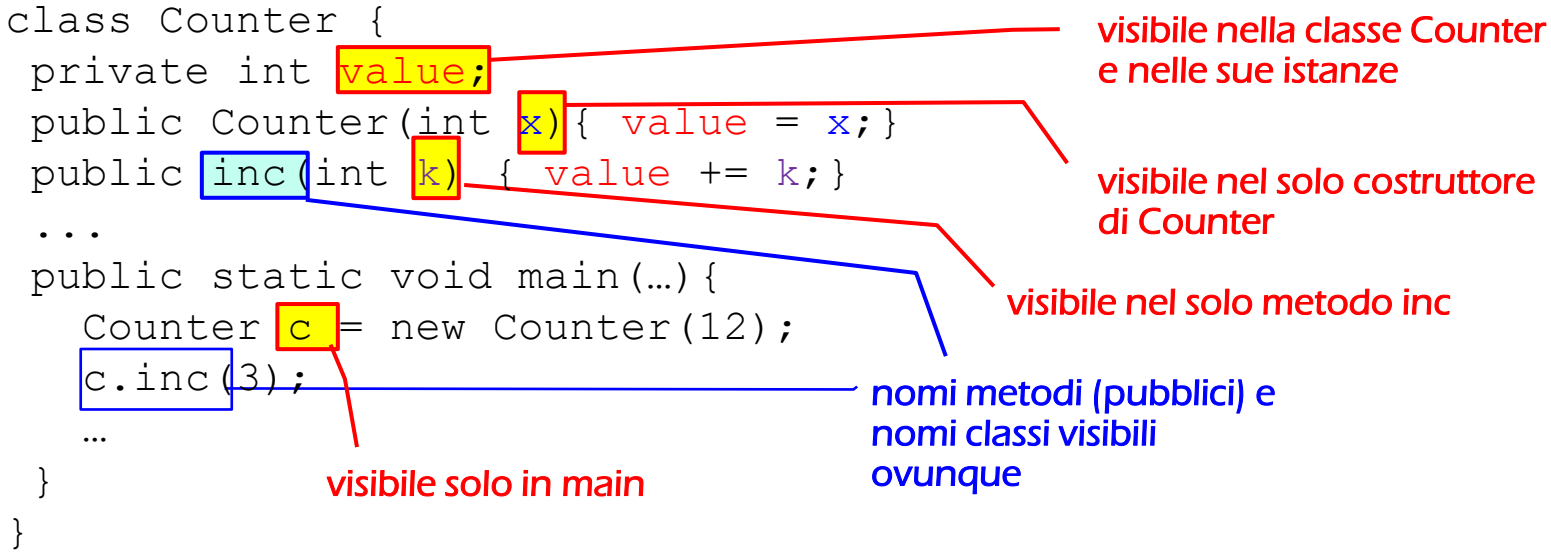
ENVIRONMENT MULTIPLI (1)

- Nei *linguaggi imperativi l'assegnamento è distruttivo* e produce *effetti collaterali* nell'environment.
- L'environment è spesso suddiviso in sotto-ambienti, di norma collegati al *tempo di vita* delle strutture run-time
  - ENVIRONMENT GLOBALE: contiene le coppie il cui tempo di vita è con l'intero programma
  - ENVIRONMENT LOCALI: contengono coppie il cui tempo di vita *non coincide* con l'intero programma – tipicamente legati all'attivazione di funzioni o altre strutture run-time.



ENVIRONMENT MULTIPLI (2)

- Ogni modello computazionale deve specificare il *campo di visibilità dei suoi simboli (scope),* ossia *quali environment sono visibili in quali punti* della struttura fisica del programma.



QUALE ASSEGNAMENTO ?

- Per introdurre l'assegnamento in un linguaggio occorre stabilire:
  - la sintassi dei nomi delle variabili
  - *se l'assegnamento sia distruttivo* o meno
  - se la scrittura di assegnamento **`x=valore`** sia *un'istruzione* o *una espressione*
    - ISTRUZIONE: effettua un'azione ma *non denota un valore*
    - ESPRESSIONE: effettua un'azione e *denota anche un valore* che costituisce il «risultato» dell'espressione
  - Quest'ultima scelta dipende dal fatto che si voglia o meno supportare l'*assegnamento multiplo.*

ASSEGNAMENTO MULTIPLO

- Di base, lo scopo dell'assegnamento *non è* denotare un valore ma *produrre un effetto collaterale* nell'environment → ha la natura di *istruzione*
  - così è infatti in vari linguaggi, come Pascal
- Tuttavia, tale approccio rende *impossibile comporre assegnamenti singoli in un assegnamento multiplo*

```
x = y = z = valore
```

  - MOTIVO: se l'assegnamento è un'istruzione, gli assegnamenti "successivi " nella catena sono privi di significato
  - Nell'esempio sopra: **`z=valore`** è chiaro, ma poi..? **`y=`**… ?

ESPRESSIONI DI ASSEGNAMENTO

- Consentire l'assegnamento multiplo implica *interpretare l'assegnamento come espressione*
  - è la scelta del linguaggio C e dei suoi derivati (ma non di Scala…)
  - occorre stabilire quale sia *il valore denotato dall'espressione*
- L'operatore di assegnamento è necessariamente *associativo a destra,* perché il valore è l'ultimo elemento
  - La frase precedente **`x = y = z = valore`** deve quindi essere interpretata come **`x = (y = (z = valore))`**
  - Informalmente: il valore "fluisce" da destra verso sinistra
  - La scelta opposta non ha significato: dopo il primo passo **`(x = y)`** non sarebbe possibile esprimere i successivi, dato che l'espressione risultante avrebbe la forma *valore* = *valore* (come 4 = 5 !!)

ASSEGNAMENTO MULTIPLO: ESEMPI

`x = y = 3-2-1`

significato equivalente a **`x=(y=0)`**, ovvero:

- **`y=0`** denota il valore 0 e *causa l'inserimento nell'environment della coppia (**`y`**,**`0`**)*
- **`x=0`** denota anch'essa il valore 0 (inutile) e *causa l'inserimento nell'environment della nuova coppia (**`x`**,**`0`**)*

`k + pigreco`

corretta purché **`k`** e **`pigreco`** siano definiti nell'ambiente al momento della valutazione

`y=(x=y=2)+3`

significato equivalente alla sequenza:

- **`x=(y=2)`** che denota il valore 2 e *causa l'inserimento nell'environment delle due coppie (**`y`**,**`2`**) e (**`x`**,**`2`**), in quest'ordine*
- **`y=2+3`** che denota il valore 5 e *causa l'inserimento nell'environment della coppia (**`y`**,**`5`**) in sostituzione della precedente (**`y`**,**`2`**)*

ESTENSIONE DELL'INTERPRETE

Per estendere il nostro interprete occorre aggiungere:

- l'*espressione di assegnamento* (operatore **=** o altro gradito)

variabile = espressione

- il concetto di *variabile* nelle sue due interpretazioni di
  - L-value: il nome di variabile indica il *contenitore* (**x =** …)
  - R-value: il nome di variabile indica il *contenuto* (… **= x+2**)
  - **Stessa sintassi destra/sinistra o sintassi differenziata?**

```
x = $x+1          x = [x]+1
Lx = Rx+1         &x = x+1
LET x = x+1       …
```

ESTENSIONE DELL'INTERPRETE

Per estendere il nostro interprete occorre aggiungere:

- l'*espressione di assegnamento* (operatore **=**)

Nuova *AssignExp* → nuova produzione per EXP

Nuovo concetto *L-Ident* → usato in *AssignExp*

hi di

- L-value: il nome di variabile indica il *contenitore* (**x =** …)
- R-value: il nome di variabile indica il *contenuto* (… = **x+2**)

Nuovo tipo di fattore → nuova produzione per FACTOR

```
x = $x+1          x = [x]+1
Lx = Rx+1         &x = x+1
LET x = x+1       …
```

ESTENSIONE DELL'INTERPRETE

Problema: così facendo la grammatica è LL(2)

- infatti, se non si differenzia sintatticamente l'identificatore "sinistro" da quello "destro", per distinguerli bisogna per forza verificare se il token successivo è l'operatore **=**
  - fattibile ma scomodo e meno efficiente
  - alternativa: differenziare sintatticamente L-Value da R_Value
- Nei parser veri (generati da strumenti automatici..) si accetta la scomodità dell'LL(2) [localmente]
- Nel nostro prototipo fatto a mano invece distigueremo *aggiungendo un **$** come in bash per R-value*

ESTENSIONE DELL'INTERPRETE

Nuovo concetto di *assegnamento*
→ nuova produzione per EXP

Essa richiede a sua volta il nuovo concetto di *identificatore*
→ nuova produzione per IDENT

```
TERM    ::=  FACTOR
TERM    ::=  TERM * FACTOR
TERM    ::=  TERM / FACTOR

FACTOR  ::=  num
FACTOR  ::=  ( EXP )
```

```
P = {
  …
EXP     ::=  ASSIGN
ASSIGN  ::=  IDENT =  EXP

FACTOR  ::=  $ IDENT
```

Nuovo tipo di fattore
→ nuova produzione per FACTOR

- **La grammatica così modificata è LL(1)** perché IDENT "destro" è ora preceduto da **$**, ergo IDENT da solo identifica la ASSIGN

ESTENSIONE DELL'INTERPRETE

Parimenti occorre *estendere la sintassi a*stratta

- Tre nuovi tipi di nodo per l'AST → *tre nuove classi Java*
  - il nodo *operazione di assegnamento* (=) — `AssignExp`
  - il nodo L-value (IDENT a sinistra dell'=) — `LIdentExp`
  - il nodo R-value (IDENT a destra dell'=) — `RIdentExp`
- Valutazione identificatori: semantica informale
  - valutare il nodo L-value significa *ricavare un modo per accedere al contenitore avente quel simbolo come nome*
  - valutare il nodo R-value significa *ricavare il valore associato nell'environment a quel simbolo*

ESTENSIONE DELL'INTERPRETE

Grammatica LL(1) con $

```
EXP      ::=  TERM
EXP      ::=  EXP + TERM
EXP      ::=  EXP – TERM

EXP      ::=  ASSIGN
ASSIGN   ::=  IDENT = EXP

TERM     ::=  FACTOR
TERM     ::=  TERM * FACTOR
TERM     ::=  TERM / FACTOR

FACTOR   ::=  num
              FACTOR
              ::=  $IDENT
FACTOR   ::=  ( EXP )
...
```

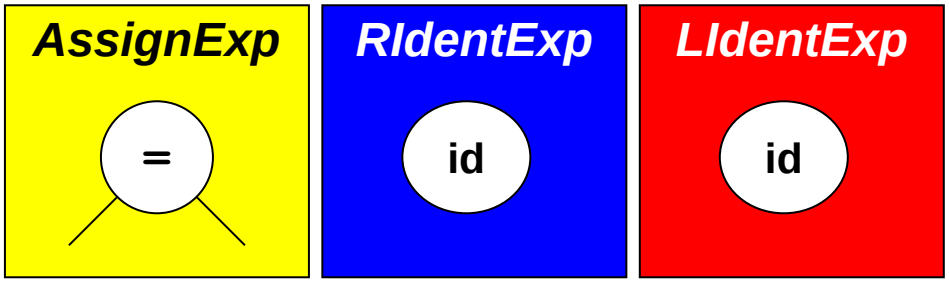
Nuovo tipo di nodo "assegnamento"

Nuove regole sintassi astratta:

```
EXP  ::=  IDENT = EXP   // AssignExp
EXP  ::=  $IDENT
```

Due tipi di nodo "identificatore" "LIdentExp" e "RIdentExp"

Tre nuovi tipi di nodo = *tre nuove classi*



ESTENSIONE DELL'INTERPRETE

Due classi distinte per rappresentare L-Value e R-Value

- **`LIdentExp`** ha un nome *che è usabile come "modo per accedere"*
  → CHIAVE nella mappa
- **`RIdentExp`** ha un nome e un valore associato (nell'environment)
  → VALORE nella mappa

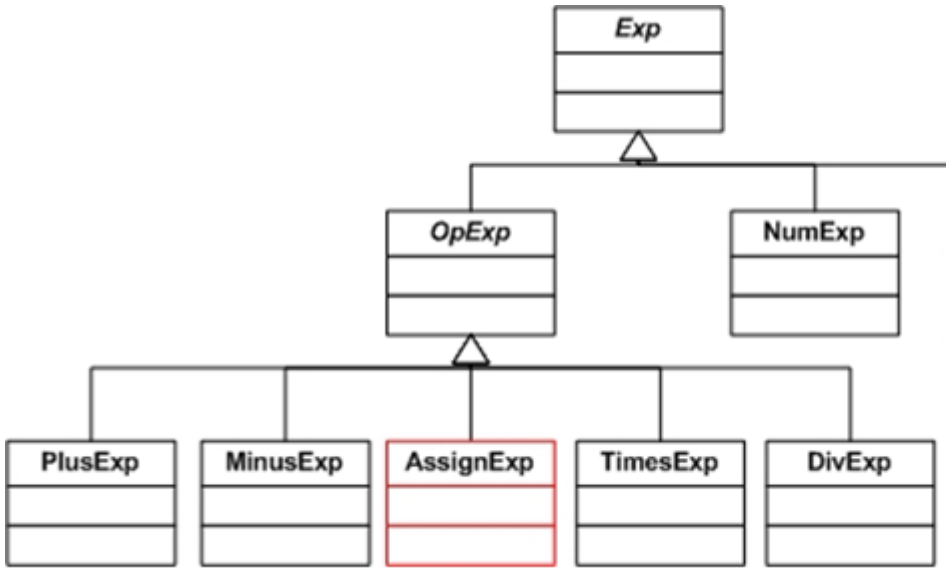
simbolo (chiave)	valore
a	3
y	5
…	…

L-value **R-value**

ESTENSIONE DELL'INTERPRETE

Nuova struttura

- **`LIdentExp`** ha un nome (usabile come chiave)
- **`RIdentExp`** ha un nome e un valore associato



R-value: il risultato della valutazione è un *numero* (come nelle altre espressioni)

L-value: il risultato della valutazione è il *modo per recuperarla* (ad esempio, il nome stesso)

SCHEMA DI PARSER

Cosa cambia nel parser?

- La classe **`Token`** ha un nuovo metodo di servizio **`isIdentifier`** che cattura la sintassi degli identificatori
- **`parseExp`** cattura anche la nuova espressione *Assign*, intercettando anche la sequenza IDENT **=** EXP
- **`parseFactor`** cattura anche il nuovo fattore *RIdent*, intercettando anche la sequenza **$** IDENT

parseExp com'era...

```
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;              // errore
   }
   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;              // errore
   }
   else return termSeq;
 } // end while
 return termSeq;
}
```

Aggiungere qui il nuovo caso *Assign*

parseExp .. e come diventa

```
public Exp parseExp() {
 Exp termSeq = parseTerm();
   ...
   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;            // errore
   } else if (currentToken.isIdentifier()) {
        String id = currentToken.toString();
        currentToken = scanner.getNextToken();
        if (!(currentToken.equals("="))) return null;
        currentToken = scanner.getNextToken();
        Exp rightExp = parseExp();
        return new AssignExp( new LIdentExp(id), rightExp);
   } else
        return termSeq; // next token non fa parte di L(Exp)
 } // end while
 return termSeq;    // next token è nullo -> stringa di input finita
}
```

parseFactor com'era...

```
public Exp parseFactor() {
  if (currentToken.equals("(")) {
     currentToken = scanner.getNextToken();
     Exp innerExp = parseExp();     // PDA: gestione self-embedding
     if (currentToken.equals(")")) {
           currentToken = scanner.getNextToken();
           return innerExp;          // le parentesi vengono omesse
     } else return null;             // errore
  }
  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
}
```

Aggiungere qui il nuovo caso
$ Ident

parseFactor ..e come diventa

```
public Exp parseFactor() {
  ...
    else // nuova regola: identificatori come R-Value
    if (currentToken.equals("$")) {
    currentToken = scanner.getNextToken();
    String id = currentToken.toString();
    currentToken = scanner.getNextToken();
    return new RIdentExp(id);
    }
  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
}
```

PICCOLO TEST

```
public class TestParser {
    public static void main(String args[]) {
        String[] expressions = {
            "12 + 4 - 11 ",
            "12 + 4 - 14 : 2 ",
            "3 + 4 * 5",
            "( 3 + 4 ) * 5 ",
            "x = 5 - 3",
            "y = 4 + $ x",
            "y = -4 * $ y",
            "z = x = $ y"
        };
        String expression = expressions[7];
        Scanner scanner = new MyStringScanner(expression);
        MyParser parser = new MyParser(scanner);
        Exp ast = parser.parseExp();
        System.out.println(ast); // da valutare poi
    }
}
```

Ad esempio, l'ultima espressione stampa **z=x=y**

VISITOR: LE NUOVE AZIONI SEMANTICHE

L'interfaccia (o classe astratta) `ExpVisitor` resta identica (altrimenti tutti i visitor già realizzati dovrebbero implementare i tre nuovi metodi): **si introduce una sua specializzazione `ExpAssignVisitor`**

```
interface ExpAssignVisitor extends ExpVisitor {
  public abstract void visit( AssignExp e );
  public abstract void visit( LIdentExp e );
  public abstract void visit( RIdentExp e );
}
```

Refactoring dei visitor

- Il visitor visualizzatore, *`ParExpVisitor`*, non ha problemi
- Il visitor valutatore, *`EvalExpVisitor`*, invece, non può più presumere che il risultato della valutazione *sia sempre un numero* perché nel caso di `IdentExp` non lo è → *generalizzare il tipo di ritorno delle funzioni*

LE NUOVE CLASSI `Exp` & CO.

```
class AssignExp extends OpExp {
  public AssignExp ( Exp l, Exp r) { super(l,r); }
  public String myOp() { return "=" ; }
  public void accept( ExpVisitor v) { ((ExpAssignVisitor)v).visit(this);}
}
class LIdentExp extends Exp {
  String name;
  public LIdentExp(String v) { name = v; }
  public String toString() { return name; }
  public String getName() { return name; }
  public void accept( ExpVisitor v) { ((ExpAssignVisitor)v).visit(this); }
}
class RIdentExp extends Exp {
  String name;
  int value;
  public RIdentExp(String v) { name = v; }
  public String toString() { return name; }
  public String getName() { return name; }
  public int getValue() { return value; }
  public void accept( ExpVisitor v) { ((AssignExpVisitor)v).visit(this);}
}
```

Cast *necessario* perché i nuovi metodi esistono solo in ExpAssignVisitor

IL VISITOR VISUALIZZATORE

Stampa dell'espressione visitando l'albero *in ordine anticipato*

```
class ParExpVisitor implements ExpVisitor, ExpAssignVisitor {

  ...
  public void visit( LIdentExp e ) {
    curs = e.getName();
  }
  public void visit( RIdentExp e ) {
    curs = e.getName();
  }
  public void visit( AssignExp e ) { visitOpExp(e); }
}
```

Estensione per gestire le tre nuove espressioni

SCELTA: il lato sinistro e destro sono stampati in modo identico

Volendo, si potrebbe differenziare la stampa dei lati sinistro e destro:

```
x = $x+1          x = [x]+1
Lx = Rx+1         &x = x+1
...
```

IL VISITOR VALUTATORE (1)

- Il risultato della valutazione *non è più sempre un numero* perché **`LIdentExp`** denota un qualche "riferimento" alla variabile
- Refactoring: le funzioni restituiscono un **`Object`** [o magari.. ?]

```
class EvalExpVisitor implements ExpVisitor {
  Object value;                          // memorizzazione risultato
  public Object getEvaluation() { return value; }

  public void visit(PlusExp e) {
    e.left().accept(this);
    int arg1 = ((Integer)getEvaluation());
    e.right().accept(this);
    int arg2 = ((Integer)getEvaluation());
    value = new Integer(arg1 + arg2);

  }
   // analogamente per le altre categorie di espressioni
}
```

IL VISITOR VALUTATORE (2)

- L'assegnamento richiede un *environment*
- Lo realizziamo tramite una **`Map<String,Integer>`** che memorizza le coppie (*nomevariabile*, *valore*)
- Valutazioni:
  - valutare **`LIdentExp`** significa *ricavare la chiave della mappa*
  - valutare **`RIdentExp`** significa *ricavare il valore corrispondente a una data chiave nella mappa*
  - valutare **`AssignExp`** significa *inserire nella mappa una nuova entry* (eventualmente rimpiazzando quella esistente con la stessa chiave, ossia lo stesso nome di variabile – semantica di assegnamento distruttivo)

IL VISITOR VALUTATORE (3)

```
class EvalAssignExpVisitor
 extends EvalExpVisitor implements ExpAssignVisitor {
 Map<String,Integer> environment = new HashMap<>();
 public void visit(AssignExp e) {
   e.left().accept(this);     // a sinistra c'è una IdentExp
                              // → recuperiamo la CHIAVE
   String id = ((LIdentExp)e.left()).getName();
   e.right().accept(this);    // a destra c'è una Exp qualsiasi
   value = getEvaluation();   // → recuperiamo il VALORE
   // è anche il risultato dell'exp → assegnamento multiplo OK
   environment.put(id, value);       // inseriamo la entry
 }
 ...
```

Map garantisce che se esiste già una entry con la stessa chiave, viene rimpiazzata

IL VISITOR VALUTATORE (4)

```
...
public void visit(LIdentExp e) {
  // valutare IndentExp → restituire la chiave (il nome)
  value = e.getName();
}

public void visit(RIdentExp e) {
  // valutare IndentValExp → restituire il valore corrisp.
  String id = e.getName();
  Integer val = environment.get(id); // recuperiamo il valore
  if (val != null) value = val;    // memorizzazione risultato
  else { // non dovrebbe mai accadere!
    throw new RuntimeException("invalid identifier");
  }
}
}
```

TEST VISITOR (1/3)

```
public class NewTest {
  // Creazione di espressioni di assegnamento
  // - a sinistra dell'operatore di assegnamento, un IDENT
  // - a destra di tale operatore, una espressione qualsiasi
  Exp s6 = new AssignExp( new LIdentExp("x") ,      // x = 5-3
             new MinusExp( new NumExp(5), new NumExp(3)) );
  Exp s7 = new AssignExp( new LIdentExp("y") ,      // y = 4+x
             new PlusExp( new NumExp(4), new RIdentExp("x")));
  Exp s8 = new AssignExp( new LIdentExp("y") ,      // y = -4*y
             new TimesExp( new NumExp(-4), new RIdentExp("y")));
  Exp s9 = new AssignExp( new LIdentExp("z") ,      // z = x = y
             new AssignExp( new LIdentExp("x") ,
              new RIdentExp("y")));
...
```

assegnamento multiplo (z=x=y)

L-value: il "valore" del nodo è la chiave

R-value: il "valore" del nodo è il valore della variabile

TEST VISITOR (2/3)

Visitor visualizzatore

```
...
ParExpVisitor visitor = new ParExpVisitor();
s6.accept(visitor);
System.out.println(s6 + "\t" + visitor.getVal() );
s7.accept(visitor);
System.out.println(s7 + "\t" + visitor.getVal() );
s8.accept(visitor);
System.out.println(s8 + "\t" + visitor.getVal() );
s9.accept(visitor);
System.out.println(s9 + "\t" + visitor.getVal() );
...
```

```
x=5-3    ( = x ( - 5 3 ) )
y=4+x    ( = y ( + 4 x ) )
y=-4*y   ( = y ( * -4 y ) )
z=x=y    ( = z ( = x y ) )
```

TEST VISITOR (3/3)

Visitor valutatore

```
 ...
 EvalAssignExpVisitor evisitor = new EvalAssignExpVisitor();
 s6.accept(evisitor);
 System.out.println(s6 + "  =  " + evisitor.getEvaluation() );
 s7.accept(evisitor);
 System.out.println(s7 + "  =  " + evisitor.getEvaluation() );
 s8.accept(evisitor);
 System.out.println(s8 + "  =  " + evisitor.getEvaluation() );
 s9.accept(evisitor);
 System.out.println(s9 + "  =  " + evisitor.getEvaluation() );
}
```

```
x=5-3   =   2
y=4+x   =   6
y=-4*y   =   -24
z=x=y   =   -24
```

NOTA: ora `getEvaluation` restituisce un `Object`, ma la stampa avviene tramite `toString` che è polimorfa, ergo tutto funziona senza modifiche:

- **la `toString` di `Integer` stampa il valore intero**
- **la `toString` di `String` stampa il nome variabile**

ALL TOGETHER NOW!

```
public class TestParser {
    public static void main(String args[]) {
        String[] expressions = {
                    ... // le stesse espressioni di prima
        };
    NicePrintExpVisitor printVisitor = new NicePrintExpVisitor();
    EvalExpVisitor evalVisitor = new EvalExpVisitor();
    for(String expression : expressions){
        Scanner scanner = new MyStringScanner(expression);
        MyParser parser = new MyParser(scanner);
        Exp ast = parser.parseExp();
        ast.accept(printVisitor);
        ast.accept(evalVisitor);
        System.out.print("L'espressione " + printVisitor.getResult()
                    + " ha come risultato " + evalVisitor.getResult());
        System.out.println("\tEnvironment: " + evalVisitor.getEnv());
    }
}
```

ALL TOGETHER NOW!

```
L'espressione (12+4)-11 ha come risultato 5        Environment: {}
L'espressione (12+4)-(14:2) ha come risultato 9    Environment: {}
L'espressione 3+(4*5) ha come risultato 23         Environment: {}
L'espressione (3+4)*5 ha come risultato 35         Environment: {}
L'espressione x=(5-3) ha come risultato 2          Environment: {x=2}
L'espressione y=(4+x) ha come risultato 6          Environment: {x=2, y=6}
L'espressione y=(-4*y) ha come risultato -24       Environment: {x=2, y=-24}
L'espressione z=(x=y) ha come risultato -24        Environment:
                                                   {x=-24, y=-24, z=-24}
```

ESPRESSIONI SEQUENZA

Nuovo obiettivo: aggiungere *espressioni-sequenza*

- COSA SONO: *sequenze di espressioni separate da virgola (`,`)*
- IPOTESI 1: la prima espressione è sempre un assegnamento
- IPOTESI 2: il valore complessivo è quello dell'exp *più a destra*

Sintassi astratta:

EXP ::= ASSIGN , EXP // *SeqExp*

NOTA: con questa estensione ci si avvicina sempre più alla sintassi e ai costrutti di un "vero" linguaggio di programmazione, in cui di solito:

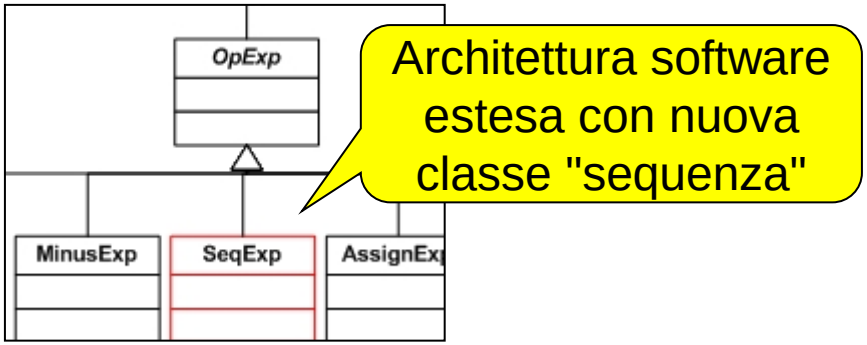
- prima si inizializzano le variabili
- poi si computa su di esse

ESTENSIONE DELL'INTERPRETE

Nuova grammatica di principio

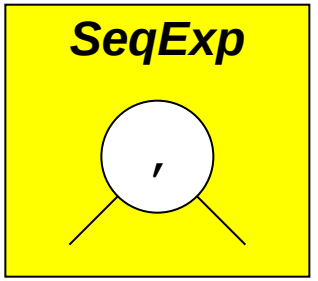
EXP	**::=**	**TERM**
EXP	**::=**	**EXP + TERM**
EXP	**::=**	**EXP – TERM**
EXP	**::=**	**ASSIGN**
EXP	**::=**	**ASSIGN , EXP**
ASSIGN	**::=**	**IDENT = EXP**

…



Architettura software estesa con nuova classe "sequenza"

Nuovo tipo di nodo "sequenza"



Sintassi astratta estesa:

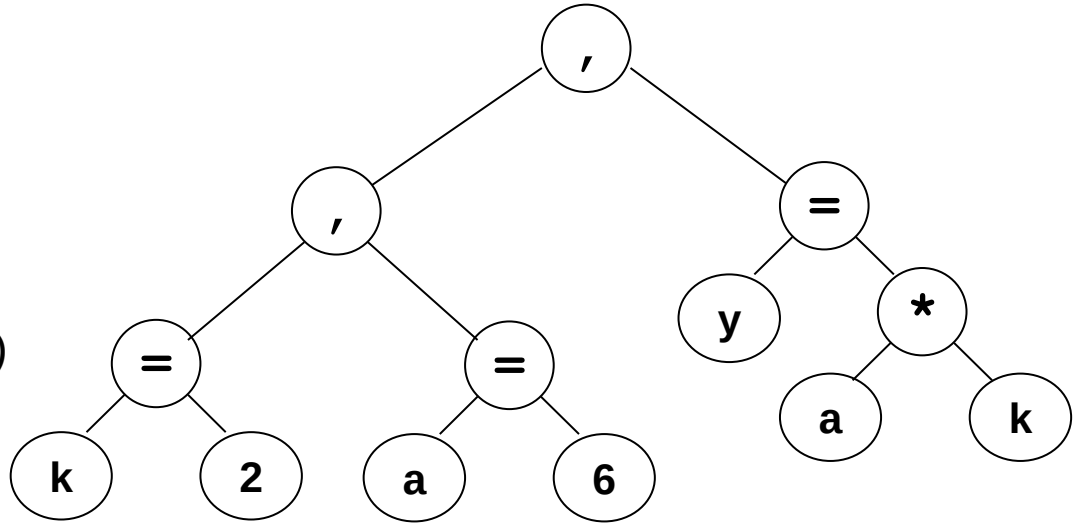
EXP ::= ASSIGN , EXP // *SeqExp*

SEMANTICA INFORMALE

- si valuta il nodo figlio ***sinistro*** *[produce effetti collaterali nell'env.]*
- si valuta il nodo figlio ***destro*** *[produce effetti collaterali nell'env.]*
- il valore del **nodo sequenza** coincide col valore del figlio destro

SEQUENZE: ESEMPI

`k=2, a=6, y=a*k`

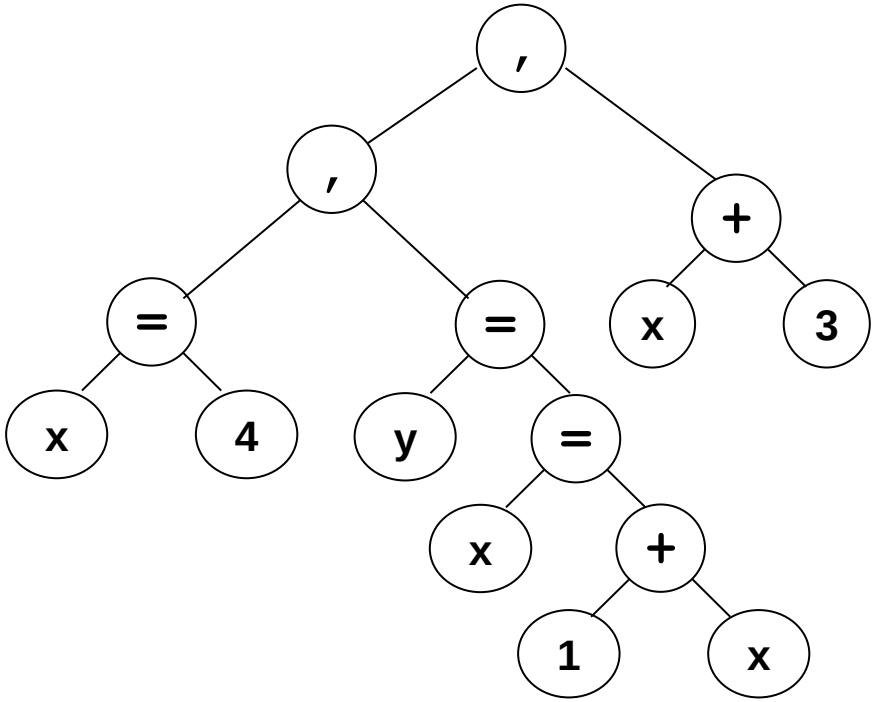


- Valuta le tre sub-espressioni, da sinistra a destra
- **`k=2`** causa l'inserimento nello environment della coppia *(**`k`**,2)*
  [la sub-espressione in sé vale 2 ma tale valore non è utilizzato]
- **`a=6`** causa l'inserimento nell'environment della coppia *(**`a`**,6)*
  [la sub-espressione in sé vale 6 ma tale valore non è utilizzato]
- **`y=a*k`** causa l'estrazione dall'environment del valore attuale di a e k, indi l'inserimento nell'environment della coppia *(**`y`**,12)*.
  Il risultato dell'espressione-sequenza in quanto tale è 12

SEQUENZE: ESEMPI

```
x=4, y=x=1+x, x+3
```

- Valuta le tre sub-espressioni sequenza da sinistra a destra
- **`x=4`** causa l'inserimento nell'environment della coppia ***(x,4)***
- **`y=x=1+x`** recupera dall'environment il valore attuale di x, indi inserisce nello environment la coppia *(**x**,5) [prima]* e la coppia *(**y**,5) [poi]*
- **`x+3`** recupera dall'environment il valore attuale di x e lo somma alla costante 3, ottenendo il risultato finale 8.



REVISIONE CRITICA

Però, se seguiamo esattamente questa grammatica:

- **`parseExp`** diventa lunga e poco leggibile (anche perché non abbiamo previsto una `parseAssign` esplicita.. ☹)
- soprattutto, ci tarpiamo le ali da soli se domani volessimo aggiungere nuovi tipi di espressioni atte a stare "anche davanti" alla virgola (come Assign)

Perciò, revisioniamo leggermente la grammatica:

- aggiungendo un nuovo livello, *la sequenza di espressioni,* che diventerà *il nuovo scopo*
- stabilendo che la sequenza sia semplicemente *una sequenza di espressioni,* senza imporre vincoli sul primo elemento

UNA GRAMMATICA PIÙ COMODA

Grammatica effettiva

```
SEQ      ::=  EXP
SEQ      ::=  SEQ , EXP

EXP      ::=  TERM
EXP      ::=  EXP + TERM
EXP      ::=  EXP – TERM

EXP      ::=  ASSIGN

ASSIGN   ::=  IDENT = EXP

…

FACTOR   ::=  num
FACTOR   ::=  ( SEQ )
FACTOR   ::=  $ IDENT
```

Nuovo scopo SEQ (anziché EXP)

Si espanderà in EBNF nel solito modo per gestire la ricorsione sinistra

Accettiamo SEQ fra parentesi come fattori (anziché EXP)

IMPLEMENTAZIONE DEL PARSER

```
public Exp parseSeq() {
   Exp expSequence = parseExp();
   while (currentToken != null) {
      if (currentToken.equals(",")) {
         currentToken = scanner.getNextToken();
         Exp nextExp = parseExp();
         if (nextExp != null)
            expSequence = new SeqExp(expSequence, nextExp);
         else
            return null;
      } else
         return expSequence;)
   } // end while
   return expSequence;
}
```

IMPLEMENTAZIONE DEL PARSER

```
public Exp parseFactor() {
   if (currentToken.equals("(")) {
      currentToken = scanner.getNextToken();
      Exp innerExp = parseSeq(); // SEQUENZA dentro parentesi
      if (currentToken.equals(")")) {
         currentToken = scanner.getNextToken();
         return innerExp;
      } else return null;
   }
   …
   …
}
```

PICCOLO TEST

```
public class TestParser {
    public static void main(String args[]) {
        String[] expressions = {
          ...
          "z = x = $ y",
          "x = 5 , y = $ x",
          "y = 4 + $ x , 3 - 5",
          "x = 7 , y = 4 + $ x",
          "x = 7 , y = 4 + $ x , w = $ y + 1",
          "( x = 7 , y = 4 + $ x ) , w = $ y + 1",
          "x = 7 , ( y = 4 + $ x , w = $ y + 1 )"
        };
        String expression = expressions[13];
        Scanner scanner = new MyStringScanner(expression);
        MyParser parser = new MyParser(scanner);
        Exp ast = parser.parseSeq();        // NUOVO SCOPO GRAMMATICA
        System.out.println(ast);            // da valutare poi
    }
}
```

Ad esempio, l'ultima espressione stampa **x=7,y=4+x,w=y+1**

ESTENSIONE DEI VISITOR VALUTATORI

Il nuovo **`ExpSeqVisitor`** specializza `ExpAssignVisitor` perché la sequenza, per come è definita, ha senso solo se esiste già la nozione di assegnamento.

```
interface ExpSeqVisitor extends ExpAssignVisitor {
  public abstract void visit( SeqExp e );
}
```

```
class SeqExp extends OpExp {
  public SeqExp ( Exp l, Exp r) { super(l,r); }
  public String myOp() { return "," ; }
  public void accept( ExpVisitor v) {
    ((ExpSeqVisitor)v).visit(this);}
}
```

I NUOVI VISITOR

```
class ParExpVisitor implements
   ExpAssignVisitor, ExpSeqVisitor {

  ...
  public void visit( SeqExp e ) { visitOpExp(e); }
}
```

Visitor visualizzatore

```
class EvalSeqExpVisitor
  extends EvalAssignExpVisitor
  implements ExpSeqVisitor {

  public void visit( SeqExp e ) {
    e.left().accept(this);   // figlio sx → va solo valutato
                             // per causare i side effect
    e.right().accept(this);  // figlio dx → va valutato e
    value = getEvaluation(); // anche restituito
  }
}
```

Visitor valutatore

TEST VISITOR (1/3)

```
Exp[] expressions = {
 new SeqExp( // x=5, y=x
   new AssignExp(new LeftIdentExp("x"),new NumExp(5)),
   new AssignExp(new LeftIdentExp("y"),new RightIdentExp("x"))),
 new SeqExp( // y=4+x, 3-5
   new AssignExp( new LeftIdentExp("y"),
                  new PlusExp( new NumExp(4), new RightIdentExp("x"))),
   new MinusExp(new NumExp(3), new NumExp(5)) ),
 new SeqExp( // x=7, y=4+x
   new AssignExp( new LeftIdentExp("x"), new NumExp(7) ),
   new AssignExp( new LeftIdentExp("y") ,new PlusExp(
                                  new NumExp(4), new RightIdentExp("x")))),
 new SeqExp( // (x=7, y=4+x), w=y+1
   new SeqExp( // x=7, y=4+x
      new AssignExp( new LeftIdentExp("x"), new NumExp(7) ),
      new AssignExp( new LeftIdentExp("y") ,
                     new PlusExp( new NumExp(4), new RightIdentExp("x")))),
   new AssignExp( new LeftIdentExp("w"),
                  new PlusExp( new RightIdentExp("y"),new NumExp(1))) )
};
```

TEST VISITOR (2/3)

```
public class TestVisitors  {
  public static void main(String args[]) {
    Exp[] expressions = {
        ...
    };
    NicePrintExpVisitor printVisitor = new NicePrintExpVisitor();
    EvalExpVisitor evalVisitor = new EvalExpVisitor();
    for(Exp expression : expressions){
        expression.accept(printVisitor);
        expression.accept(evalVisitor);
        System.out.print("L'espressione " + printVisitor.getResult()
                         + " ha come risultato " + evalVisitor.getResult());
        System.out.println("\tEnvironment: " + evalVisitor.getEnv());
    }
 }
```

Una serie di Exp (cioè di AST)

TEST VISITOR (3/3)

```
L'espressione 3-5 ha come risultato -2                    Environment: {}
L'espressione (2+(5*4))-1 ha come risultato 21            Environment: {}
L'espressione (3-5)-(1*5) ha come risultato -7            Environment: {}
L'espressione 5-(3-1) ha come risultato 3                 Environment: {}
L'espressione (5-3)-1 ha come risultato 1                 Environment: {}
L'espressione x=(5-3) ha come risultato 2                 Environment: {x=2}
L'espressione y=(4+x) ha come risultato 6                 Environment: {x=2, y=6}
L'espressione y=(-4*y) ha come risultato -24              Environment: {x=2, y=-24}
L'espressione z=(x=y) ha come risultato -24               Environment:
                                                          {x=-24, y=-24, z=-24}
L'espressione (x=5),(y=x) ha come risultato 5             Environment:
                                                          {x=5, y=5, z=-24}
L'espressione (y=(4+x)),(3-5) ha come risultato -2
                                                          Environment:
                                                          {x=5, y=9, z=-24}
L'espressione (x=7),(y=(4+x)) ha come risultato 11
                                                          Environment:
                                                          {x=7, y=11, z=-24}
L'espressione ((x=7),(y=(4+x))),(w=(y+1)) ha come risultato 12
                                                          Environment:
                                                          {w=12, x=7, y=11, z=-24}
```

ALL TOGETHER NOW!

```
public class TestParser {
    public static void main(String args[]) {
        String[] expressions = {
                    ... // le stesse espressioni di prima, come STRINGHE!
        };
    NicePrintExpVisitor printVisitor = new NicePrintExpVisitor();
    EvalExpVisitor evalVisitor = new EvalExpVisitor();
    for(String expression : expressions){
        Scanner scanner = new MyStringScanner(expression);
        MyParser parser = new MyParser(scanner);
        Exp ast = parser.parseSeq(); // NUOVO SCOPO GRAMMATICA
        ast.accept(printVisitor);
        ast.accept(evalVisitor);
        System.out.print(printVisitor.getResult()
                         + " vale " + evalVisitor.getResult());
        System.out.println("\tEnv: " + evalVisitor.getEnv());
    }
}
```

Una serie di stringhe

ALL TOGETHER NOW!

```
(12+4)-11 vale 5                          Env: {}
(12+4)-(14:2) vale 9                      Env: {}
3+(4*5) vale 23                           Env: {}
(3+4)*5 vale 35                           Env: {}
x=(5-3) vale 2                            Env: {x=2}
y=(4+x) vale 6                            Env: {x=2, y=6}
y=(-4*y) vale -24                         Env: {x=2, y=-24}
z=(x=y) vale -24                          Env: {x=-24, y=-24, z=-24}
(x=5),(y=x) vale 5                        Env: {x=5, y=5, z=-24}
(y=(4+x)),(3-5) vale -2                   Env: {x=5, y=9, z=-24}
(x=7),(y=(4+x)) vale 11                   Env: {x=7, y=11, z=-24}
((x=7),(y=(4+x))),(w=(y+1)) vale 12       Env: {w=12, x=7, y=11, z=-24}
((x=7),(y=(4+x))),(w=(y+1)) vale 12       Env: {w=12, x=7, y=11, z=-24}
(x=7),((y=(4+x)),(w=(y+1))) vale 12       Env: {w=12, x=7, y=11, z=-24}
```

RIASSUMENDO

- Il parser del "linguaggio espressioni ora è completo
  - valuta espressioni su interi, con variabili e sequenze
- Quanto è distante da un "vero" linguaggio?
  - concettualmente non molto, in realtà
  - mancano varie cosette, *ma non è difficile aggiungerle…*
    - le espressioni relazionali e logiche
    - le strutture di controllo `for`, `while`, `if`, blocchi `{…}`
    - le funzioni *(più complicato…),* i tipi…
- E se volessimo un compilatore?
  - basterebbe che un visitor *emettesse codice*
  - ..magari per *una Extended Arithmetic Stack Machine* con variabili e sequenze ☺

PAZZA IDEA: LE DERIVATE!

- **E se volessimo un visitor *derivatore*, che calcoli la derivata simbolica di un'espressione?**
  - escludendo, ovviamente, assegnamenti e sequenze
- Sarebbe facile!
  - basterebbe sintetizzare via via *una nuova Exp,* seguendo le note regole di derivazione
- Cosa cambierebbe?
  - la visita di **`NumExp`** deve restituire sempre 0
  - la visita di **`IdentRightExp`** deve restituire sempre 1 *se la variabile è quella di derivazione,* o sempre 0 altrimenti
  - la visita di **`PlusExp`** e **`MinusExp`** è banale
  - la visita di **`TimesExp`** e **`DivExp`** un po' meno.. MA SI FA!

TANTO PER GRADIRE... ☺

```
public class DeriverExpVisitor implements ExpVisitor {
  private Exp exp = null;
  private String derVariable="x";
  public void setDerivativeVariable(String x) {derVariable=x;}
  public Exp getResult() { return exp; } // accessor
  public void visit(PlusExp e) {
    e.getLeft().accept(this);  Exp arg1 = getResult();
    e.getRight().accept(this); Exp arg2 = getResult();
    exp = isZero(arg1) ? arg2 :
          (isZero(arg2) ? arg1 : new PlusExp(arg1,arg2));
  }
  public void visit(RightIdentExp e) {
    exp = e.getName().equals(derVariable) ? new NumExp(1) :
                                            new NumExp(0);
  }
...
```

DERIVATE IN PIÙ VARIABILI ☺ ☺

Derivata in x		Derivata in y		Derivata in z	
4+x	1	4+x	0	4+x	0
4*x	4	4*x	0	4*x	0
x+4	1	x+4	0	x+4	0
x*4	4	x*4	0	x*4	0
x*x	x+x	x*x	0	x*x	0
3*x*x	3*(x+x)	3*x*x	0	3*x*x	0
x*x*3	(x+x)*3	x*x*3	0	x*x*3	0
y*y	0	y*y	y+y	y*y	0
5*y*y	0	5*y*y	5*(y+y)	5*y*y	0
4:x	(0-4):(x*x)	4:x	0	4:x	0
x:4	4:(4*4)	x:4	0	x:4	0
4:z	0	4:z	0	4:z	(0-4):(z*z)
z:4	0	z:4	0	z:4	4:(4*4)
x*y	y	x*y	x	x*y	0
x*y*z	y*z	x*y*z	x*z	x*y*z	x*y