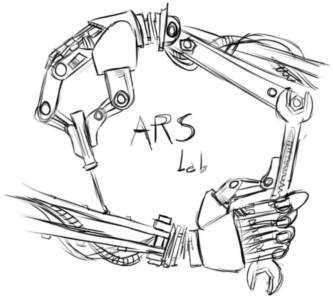
Programmazione Dichiarativa in Python con PHIDIAS

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



Programmazione Sistemi Robotici

PHIDIAS

- PHIDIAS *(PytHon Interactive Declarative Intelligent Agent System*) è un tool Python per la programmazione di *sistemi autonomi*—agenti o robot—utilizzando il modello **dichiarativo**
- L'obiettivo è offrire al programmatore un unico ambiente di programmazione che consenta di scrivere sia codice **imperativo** che codice **dichiarativo**
- E' disponibile su github: `http://github.com/corradosantoro/phidias`

- PHIDIAS è basato sul paradigma **belief-desire-intention (BDI)** e consente la specifica, all'interno di codice Python, di comportamenti basati su:
- Conoscenza e derivazione di conoscenza
- Regole "reattive" (modello event-condition-action)
- Regole "proattive"

Conoscenza in PHIDIAS

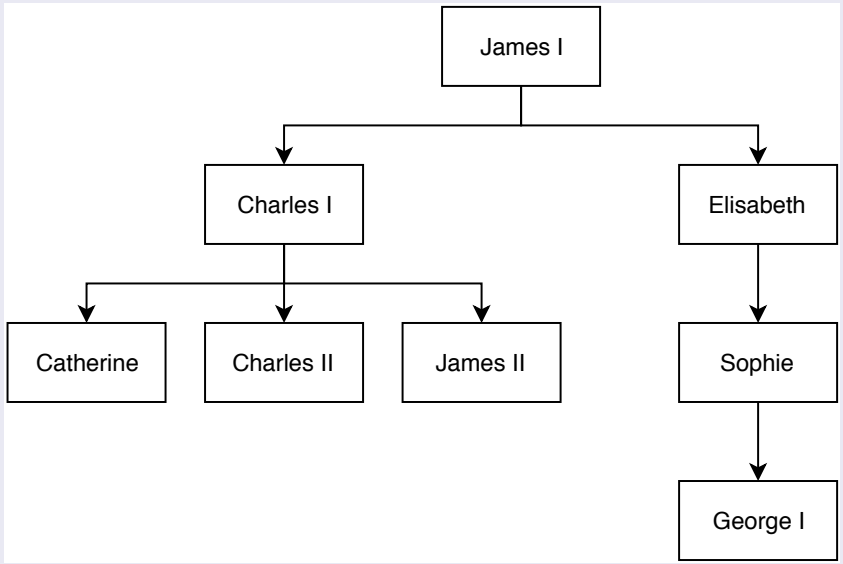
Conoscenza in PHIDIAS—Beliefs

- Il modello di conoscenza in PHIDIAS è simile a quello del Prolog
- La conoscenza di base è specificata attraverso i **Beliefs** (equivalenti ai "fatti" Prolog)
- I Beliefs devono essere preventivamente **dichiarati** come semplici sotto-classi della classe base `Belief`
- Successivamente possono essere espressi usando la classica sintassi basata su formule atomiche con termini ground:
  - `position(150,60)`
  - `block("red")`
  - `stack("cube","pyramid")`
- I Beliefs possono essere **aggiunti** o **rimossi** dalla **Knowledge Base (KB)**
- La KB può essere **interrogata** per controllare la presenza di determinati beliefs e comportarsi di conseguenza

Conoscenza in PHIDIAS—Goals

- La conoscenza **derivata** è rappresentata dai **Goal**
- I Goal devono essere preventivamente **dichiarati** come semplici sotto-classi della classe base `Goal`
- Successivamente possono essere espressi utilizzando formule logiche con connettivi AND, in logica del primo ordine
- Le formule possono contenere variabili quantificante universalmente o termini ground
- Le variabili utilizzate devono essere dichiarate, va cioè dichiarata la loro presenza ma non il loro tipo

Esempio: L'Albero Genealogico



Beliefs

- `male(X)` → X è un uomo
- `female(X)` → X è una donna
- `parent(X, Y)` → X è genitore di Y

Esempio: L'Albero Genealogico

Conoscenza Derivata: Goals

- `father(X, Y)` $\rightarrow$ X è il padre di Y:

$$\forall x, y : parent(x, y) \wedge male(x) \Rightarrow father(x, y)$$

- `mother(X, Y)` $\rightarrow$ X è la madre di Y:

$$\forall x, y : parent(x, y) \wedge female(x) \Rightarrow mother(x, y)$$

Esempio: L'Albero Genealogico (1)

```
# library imports
from phidias.Types  import *
from phidias.Main import *
from phidias.Lib import *

# definizione dei beliefs
class parent(Belief): pass
class male(Belief): pass
class female(Belief): pass

# definizione dei goal
class father(Goal): pass
class mother(Goal): pass

# definizione delle variabili usate
# nel programma (occhio agli apici!)
def_vars('X','Y')

# definizione dei goal come predicati logici
father(X,Y) << ( parent(X,Y) & male(X) )
mother(X,Y) << ( parent(X,Y) & female(X) )

# ....
```

Esempio: L'Albero Genealogico (2)

```
# ...
# inserimento iniziale nella knowledge base
PHIDIAS.assert_belief(male('james1')),
PHIDIAS.assert_belief(male('charles1'))
PHIDIAS.assert_belief(male('charles2'))
PHIDIAS.assert_belief(male('james2'))
PHIDIAS.assert_belief(male('george1'))
PHIDIAS.assert_belief(female('catherine'))
PHIDIAS.assert_belief(female('elizabeth'))
PHIDIAS.assert_belief(female('sophia'))
PHIDIAS.assert_belief(parent('james1', 'charles1'))
PHIDIAS.assert_belief(parent('james1', 'elizabeth'))
PHIDIAS.assert_belief(parent('charles1', 'charles2'))
PHIDIAS.assert_belief(parent('charles1', 'catherine'))
PHIDIAS.assert_belief(parent('charles1', 'james2'))
PHIDIAS.assert_belief(parent('elizabeth', 'sophia'))
PHIDIAS.assert_belief(parent('sophia', 'george1'))

# start dell'engine di PHIDIAS
PHIDIAS.run()

# start della shell interattiva
PHIDIAS.shell(globals())
```

Esempio di run: L'Albero Genealogico

```
PHIDIAS Release 1.1.0
Autonomous and Robotic Systems Laboratory
Department of Mathematics and Informatics
University of Catania, Italy (santoro@dmi.unict.it)


eShell:  main > kb
male('james1')                    male('charles1')
male('charles2')                  male('james2')
male('george1')                   female('catherine')
female('elizabeth')               female('sophia')
parent('james1', 'charles1')      parent('james1', 'elizabeth')
parent('charles1', 'charles2')    parent('charles1', 'catherine')
parent('charles1', 'james2')      parent('elizabeth', 'sophia')
parent('sophia', 'george1')
eShell:  main >
```

Esempio di run: L'Albero Genealogico (2)

```
eShell:  main > father(X,Y)
father('james1', 'charles1')
father('james1', 'elizabeth')
father('charles1', 'charles2')
father('charles1', 'catherine')
father('charles1', 'james2')
eShell:  main > mother(X,Y)
mother('elizabeth', 'sophia')
mother('sophia', 'george1')
eShell:  main >
```

Esempio: L'Albero Genealogico

Conoscenza Derivata

- **`sibling(X, Y)`** $\rightarrow$ X è fratello o sorella di Y (X e Y hanno lo stesso genitore):
  $\exists p, \forall x, y : parent(p, x) \wedge parent(p, y) \wedge x \neq y \Rightarrow sibling(x, y)$
- **`brother(X, Y)`** $\rightarrow$ X è fratello di Y:
  $\forall x, y : sibling(x, y) \wedge male(x) \Rightarrow brother(x, y)$
- **`sister(X, Y)`** $\rightarrow$ X è sorella di Y:
  $\forall x, y : sibling(x, y) \wedge female(x) \Rightarrow sister(x, y)$

```
sibling(X,Y) << ( parent(P, X) & parent(P, Y) & neq(X,Y) )

brother(X,Y) << (sibling(X,Y) & male(X))

sister(X,Y) << (sibling(X,Y) & female(X))
```

neq(X,Y) (not-equal) è un *belief speciale* (ActiveBelief) che controlla se gli argomenti (X,Y) sono diversi tra loro

Inside PHIDIAS

Inside PHIDIAS

Una espressione del tipo:

```
sibling(X,Y) << ( parent(P, X) & parent(P, Y) & neq(X,Y) )
```

viene interpretata da Python nel seguente modo:

- Essendo **`sibling`**, **`parent`** e **`neq`** delle classi Python, la scrittura **`parent(X,Y)`** equivale alla creazione di un oggetto della relativa classe
- Le variabili *X* e *Y* sono definite da **`def_vars(...)`**, funzione che crea, a runtime, degli oggetti di tipo **`Variable`**, assegnandogli il relativo nome, così che esse possano utilizzate nelle espressioni
- Attraverso la tecnica dell'**overloading degli operatori**, l'uso degli operatori `<<` e `&` attiva gli opportuni metodi delle classi base `Goal` e `Belief`, il cui codice provvede a creare, nell'engine di PHIDIAS, un'opportuna struttura dati che rappresenta il goal
- L'engine di PHIDIAS provvede ad analizzare tale struttura dati e interpretare le espressioni opportunamente (come predicato Prolog, in questo caso)

Programmazione del Comportamento in PHIDIAS

Comportamento in PHIDIAS

Il comportamento (parte computazione) in PHIDIAS è implementato attraverso dei **plans** (piani) i quali possono essere di tipo:

- **reattivo** basati sul paradigma **Event-Condition-Actions**
- **proattivo** basati su procedure vincolate ad una **pre-condizione**

Programmazione Reattiva con PHIDIAS

Piani Reattivi (Reactive Plans)

- I **piani reattivi** sono specificati utilizzando la seguente sintassi:
  evento "/" condizione ">>" "[" lista di azioni "]"
- L'**evento** può essere costituito da:

asserzione di un belief	`+bel(...)`
rimozione di un belief	`-bel(...)`

- La **condizione** è un predicato specificato su uno più beliefs presenti nella knowledge base e/o sulle variabili
- Essa può contenere variabili libere (assegnate a seconda del belief estratto) e utilizzare anche Goal precedentemente definiti
- La **lista di azioni** reppresenta le "cose da fare" quando il plan è eseguito; la lista può contenere i seguenti statement (separati da virgola):

asserzione di belief	`+bel(...)`
rimozione di belief	`-bel(...)`
chiamata a procedura	`proc(...)`
azioni di libreria o user-defined	`go_to(X, Y)`
expression Python	`"X = X + 1"`

Un Esempio “razionale”: Studenti e Laureati

- Desideriamo rappresentare un mondo nel quale abbiamo degli studenti che prima o poi ottengono la laurea
- Una volta che uno studente ottiene la laurea, non è più “student”
- Utilizziamo due beliefs:
  - `student(X)` per rappresentare il fatto che “**X**” è uno studente
  - `graduated(X)` per rappresentare il fatto che “**X**” è un laureato
- Dobbiamo dunque imporre le seguenti “regole di conoscenza”:
  - “**X**”, per poter essere `graduated(X)`, dovrà prima essere `student(X)`
  - Quando “**X**” diventa `graduated(X)`, egli non è più `student(X)`

```
class student(Belief): pass
class graduated(Belief): pass

def_vars('X')
+graduated(X) / student(X) >> [ -student(X),
     show_line("yeah ", X, "is now graduated!") ]
+graduated(X) >> [ show_line(X, "is not a student"),
                   -graduated(X) ]
```

Analizziamo l'esempio...

```
class student(Belief): pass
class graduated(Belief): pass

def_vars('X')
+graduated(X) / student(X) >> [ -student(X),
     show_line("yeah ", X, "is now graduated!") ]
+graduated(X) >> [ show_line(X, "is not a student"),
                   -graduated(X) ]
```

- Entrambi i plans hanno lo stesso **triggering event**: `+graduated(X)`
- Questi due piani rappresentano un **gruppo** in quanto condividono lo stesso evento
- Tuttavia **un solo piano** di un gruppo può essere eseguito
- La selezione è effettuata sulla base dell'**ordine di scrittura**: il primo piano che soddisfa la condizione verrà eseguito

Un Esempio “razionale”: Studenti e Laureati

Aggiungiamo un’ulteriore regola di consistenza della conoscenza:

- Un laureato non può tornare studente

```
class student(Belief): pass
class graduated(Belief): pass

def_vars('X')
+graduated(X) / student(X) >> [ -student(X),
     show_line("yeah ", X, " is now graduated!") ]
+graduated(X) >> [ show_line(X, " is not a student"),
                   -graduated(X) ]
+student(X) / graduated(X) >> \
  [ show_line(X, " is graduated and cannot be a student again"),
    -student(X) ]
```

Programmazione Reattiva: il Crivello di Eratostene

Algoritmo

- L'algoritmo del "Crivello di Eratostene" per la ricerca dei numeri primi opera considerando tutti i numeri da 1 a N, e procede "cancellando" via via i multipli
- I numeri rimanenti (non cancellati) saranno i numeri primi

Implementazione in PHIDIAS

- Consideriamo ogni numero rappresentato dal belief `number(X)`
- Utilizziamo un regola che si attiva con l'evento di asserzione del belief `number(X)`
- La condizione specifica la ricerca di un numero Y che sia sottomultiplo di X: nel caso esista, il numero X va rimosso dalla knowledge base

Implementazione in PHIDIAS

- Consideriamo ogni numero rappresentato dal belief **`number(X)`**
- Utilizziamo un regola che si attiva con l'evento di asserzione del belief **`number(X)`**
- La condizione specifica la ricerca di un numero Y che sia sottomultiplo di X: nel caso esista, il numero X va rimosso dalla knowledge base

```
from phidias.Types  import *
from phidias.Main import *
from phidias.Lib import *

class number(Belief): pass

def_vars("X","Y")
+number(X) / (number(Y) & neq(X, Y) & (lambda: (X % Y) == 0) ) >> [ -number(X) ]

# instantiate the engine
PHIDIAS.run()

# populate the KB (and run the rules)
for i in range(2,100):
    PHIDIAS.assert_belief(number(i))

# run the engine shell
PHIDIAS.shell(globals())
```

Implementazione in PHIDIAS e semantica di esecuzione

- La parte computazionale dell'algoritmo risiede nella regola:

```
+number(X) / (number(Y) & neq(X, Y) & (lambda: (X % Y) == 0) )
   >> [ -number(X) ]
```

- All'asserzione di un numero X, la condizione permette di trovare tutti i numeri Y sottomultipli di X
- Ogni numero Y trovato genera un'istanza (**intention**) di possibile esecuzione del piano
- Tuttavia solo la **prima istanza** (intention) viene eseguita

Programmazione Reattiva: il Crivello di Eratostene

Esempio di esecuzione

```
+number(X) / (number(Y) & neq(X, Y) & (lambda: (X % Y) == 0) )
   >> [ -number(X) ]
```

- Viene asserito il belief **+number(6)**
- Il sistema identifica le seguenti due intention:

```
+number(6) / (number(2) & neq(6, 2) & (lambda: (6 % 2) == 0) )
   >> [ -number(6) ]
```

```
+number(6) / (number(3) & neq(6, 3) & (lambda: (6 % 3) == 0) )
   >> [ -number(6) ]
```

- Viene eseguita la **prima intention**

```
+number(6) / (number(2) & neq(6, 2) & (lambda: (6 % 2) == 0) )
   >> [ -number(6) ]
```

Predicati e ActiveBeliefs

Lambda e ActiveBeliefs

L'algoritmo del crivello di Eratostene richiede la presenza di due condizioni:

- il numero estratto Y deve essere diverso dal numero inserito X
- Il numero estratto Y deve essere sottomultiplo di X

Queste condizioni possono essere espresse utilizzando delle funzioni **lambda**:

```
+number(X) /
(number(Y) & (lambda : X != Y) & (lambda: (X % Y) == 0) )
   >> [ -number(X) ]
```

Lambda e ActiveBeliefs

```
+number(X) /
(number(Y) & (lambda : X != Y) & (lambda: (X % Y) == 0) )
   >> [ -number(X) ]
```

Possono esistere tuttavia dei casi in cui:

- La condizione da specificare è più complessa di un semplice confronto e si vorrebbe utilizzare una funzione booleana ad-hoc, oppure...
- Si desidera poter effettuare il binding di una variabile con un dato (ad esempio) campionato da un sensore

In questi casi, è possibile utilizzare un oggetto di tipo **`ActiveBelief`** le cui caratteristiche sono:

- Esso **non rappresenta una conoscenza** pertanto non può essere inserito in KB
- Esso implementa un predicato/confronto nel metodo **`evaluate`**, il quale restituisce un booleano

Lambda e ActiveBeliefs

```
class Multiple(ActiveBelief):
    def evaluate(self, x, y):
        return (x() % y()) == 0

+number(X) / (number(Y) & (lambda : X != Y) & Multiple(X,Y) )
   >> [ -number(X) ]
```

- L'esempio mostra l'ActiveBelief **`Multiple`** il quale permette di valutare se il primo argomento è multiplo del secondo
- Esso è utilizzato all'interno della condizione del piano
- Nel metodo **`evaluate`**, gli argomenti sono **oggetti speciali**, pertanto il valore effettivo delle variabili va "estratto" utilizzando la scrittura **`x()`**

ActiveBeliefs di libreria

- La libreria PHIDIAS include già un insieme di ActiveBeliefs che implementano i classici confronti:

`eq(X,Y)`	$X == Y$
`neq(X,Y)`	$X != Y$
`gt(X,Y)`	$X > Y$
`geq(X,Y)`	$X >= Y$
`lt(X,Y)`	$X < Y$
`leq(X,Y)`	$X <= Y$

- Usando solo ActiveBeliefs, il piano del crivello di Eratostene diventa:

```
class Multiple(ActiveBelief):
    def evaluate(self, x, y):
        return (x() % y()) == 0

+number(X) / (number(Y) & neq(X, Y) & Multiple(X,Y) )
   >> [ -number(X) ]
```

Programmazione Proattiva con PHIDIAS

Piani Proattivi (Proactive Plans)

- I **piani proattivi** sono specificati utilizzando la seguente sintassi: *procedura "/" condizione ">>" "[" lista di azioni "]"*
- La **procedura** indica come è invocato il plan
- Essa è dichiarata come sottoclasse di **`Procedure`** e può contenere variabili o termini ground
- La **condizione** è un predicato specificato su uno più beliefs presenti nella knowledge base e/o sulle variabili
- Essa può contenere variabili libere (assegnate a seconda del belief estratto) e utilizzare anche Goal precedentemente definiti
- La **lista di azioni** reppresenta le "cose da fare" quando il plan è eseguito; la lista può contenere i seguenti statement (separati da virgola):

asserzione di belief	`+bel(...)`
rimozione di belief	`-bel(...)`
chiamata a procedura	`proc(...)`
azioni di libreria o user-defined	`go_to(X, Y)`
expression Python	`"X = X + 1"`

“Hello World” in PHIDIAS

Implementazione in PHIDIAS

```
from phidias.Types  import *
from phidias.Main import *
from phidias.Lib import *

class say_hello(Procedure): pass

say_hello() >> [ show_line("Hello world from Phidias") ]

PHIDIAS.run()
PHIDIAS.shell(globals())
```

Costrutti condizionali e iterazione

- PHIDIAS non include costrutti condizionali espliciti (**if**), né iterazione (**for**/**while**)
- Tuttavi l'uso delle condizioni e dell'iterazione permette di superare il problema

Esempio: il calcolo del fattoriale

```
from phidias.Types  import *
from phidias.Main import *
from phidias.Lib import *

class fact(Procedure): pass

def_vars("Acc", "N")
fact(N) >> [ fact(N, 1) ]
fact(1, Acc) >> [ show_line("the resulting factorial is = ", Acc) ]
fact(N, Acc) >> \
    [
        "Acc = N * Acc",
        "N = N - 1",
        fact(N, Acc)
    ]

PHIDIAS.run()
PHIDIAS.shell(globals())
```

Esempio: calcolo del massimo

```
from phidias.Types  import *
from phidias.Main import *
from phidias.Lib import *

import random

class number(Belief): pass

class compute_max(Procedure): pass

def_vars('X','TempMax')

compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]

# populate the KB
for i in range(1,50):
    PHIDIAS.assert_belief(number(random.uniform(0,50)))

# instantiate the engine
PHIDIAS.run()

# run the engine shell
PHIDIAS.shell(globals())
```

Calcolo del Massimo

Knowledge Base

```
number(5)        number(8)        number(2)
number(10)       number(3)        number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Viene invocata la procedura **`compute_max()`**, il sistema identifica le seguenti intention:

```
compute_max() / number(5) >> [ compute_max(5) ]
compute_max() / number(8) >> [ compute_max(8) ]
compute_max() / number(2) >> [ compute_max(2) ]
compute_max() / number(10) >> [ compute_max(10) ]
compute_max() / number(3) >> [ compute_max(3) ]
compute_max() / number(4) >> [ compute_max(4) ]
```

Calcolo del Massimo

Knowledge Base

```
number(5)      number(8)      number(2)
number(10)     number(3)      number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Viene eseguita la **prima intention**:

```
compute_max() / number(5) >> [ compute_max(5) ]
```

Calcolo del Massimo

Knowledge Base

```
number(5)        number(8)        number(2)
number(10)       number(3)        number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Invocata la procedura **`compute_max(5)`**, il sistema identifica le seguenti intention:

```
compute_max(5) / (number(8) & gt(8, 5)) >> [ compute_max(8) ]
compute_max(5) / (number(10) & gt(10, 5)) >> [ compute_max(10) ]
```

Calcolo del Massimo

Knowledge Base

```
number(5)      number(8)      number(2)
number(10)     number(3)      number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Viene eseguita la **prima intention**:

```
compute_max(5) / (number(8) & gt(8, 5)) >> [ compute_max(8) ]
```

Calcolo del Massimo

Knowledge Base

```
number(5)        number(8)        number(2)
number(10)       number(3)        number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Invocata la procedura **`compute_max(8)`**, il sistema identifica le seguenti intention:

```
compute_max(8) / (number(10) & gt(10, 8)) >> [ compute_max(10) ]
```

Calcolo del Massimo

Knowledge Base

```
number(5)      number(8)      number(2)
number(10)     number(3)      number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Viene eseguita la **prima intention**:

```
compute_max(8) / (number(10) & gt(10, 8)) >> [ compute_max(10) ]
```

Calcolo del Massimo

Knowledge Base

```
number(5)        number(8)        number(2)
number(10)       number(3)        number(4)
```

Esempio di esecuzione

```
compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

Invocata la procedura **`compute_max(10)`**, il sistema identifica la seguente intention e l'iterazione si conclude:

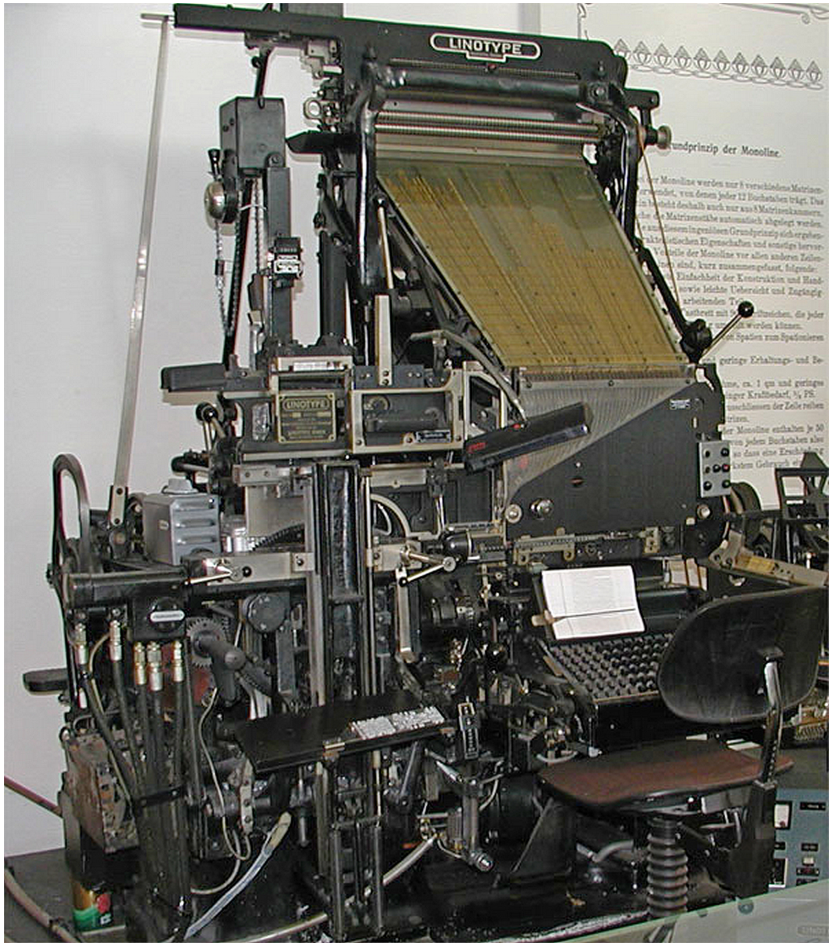
```
compute_max(10) >> [ show_line("The maximum is ", 10) ]
```

Case Study
SHRDLU: il mondo dei blocchi

Il mondo di SHRDLU

- **SHRDLU** è uno dei primi programmi di intelligenza artificiale (1968-1970), esso supportava **ragionamento/planning** e processing del linguaggio naturale
- SHRDLU “vive” in un ambiente virtuale composto da alcuni “blocchi” con colori e forme differenti (cubi, piramidi, prismi, cilindri, ecc.)
- Il programma era in grado di comprendere comandi come “Pick up a big red block” o “Grasp the pyramid” e comportarsi di conseguenza
- Lo strano nome **SHRDLU** viene dalla sequenza ETAOIN SHRDLU la quale è la stessa sequenza di tasti che si trova sulla macchina “Linotype” (tale sequenza è basata sulla frequenza delle lettere nella lingua Inglese)

La Linotype e la sequenza SHRDLU



SHRDLU e PHIDIAS

- Implementiamo una piccola versione di SHRDLU con PHIDIAS (file "`SHRDLU.py`")
- Il nostro mondo è basato sugli oggetti: **cube**, **cylinder**, **prism**
- Questi oggetti possono essere presenti sul **tavolo** e possono essere **catturati** da un robot
- Sul tavolo, gli oggetti possono essere **impilati**, l'uno sull'altro, in modo da formare una **torre**
- Un oggetto può essere catturato solo se è **libero**, cioè non ha altri oggetti sopra di esso

SHRDLU e PHIDIAS

BELIEFS

- `obj(X)`, rappresenta la presenza del blocco **X** sul tavolo
- `owned(X)`, rappresenta il fatto che il blocco **X** è stato afferrato dal robot
- `upon(X,Y)`, il blocco **X** è posto sopra il blocco **Y**

PROCEDURES

- `pick(X)`, richiesta di afferrare il blocco **X**, se possibile
- `put(X)`, richiesta di posizionare il blocco **X** sul tavolo
- `put(X,Y)`, richiesta di posizionare il blocco **X** sul blocco **Y**

SHRDLU plans: cattura degli oggetti

- Se l'oggetto X è sul tavolo, ma un altro oggetto Y è sopra X, allora X non può essere catturato

```
pick(X) / (obj(X) & upon(Y, X)) >> \
  [ show_line("Cannot pick ", X, " since it is under the ", Y) ]
```

- Se la condizione precedente è **falsa** e l'oggetto X è sul tavolo ed è sopra un oggetto Y, allora X può essere catturato

```
pick(X) / (obj(X) & upon(X, Y)) >> \
          [
            show_line(X, " has been picked"),
            -obj(X), -upon(X, Y), +owned(X)
          ]
```

SHRDLU plans: cattura degli oggetti

- Se la condizione precedente è **falsa** e l'oggetto X è sul tavolo (nessun altro oggetto su di esso) allora X può essere catturato

```
pick(X) / obj(X) >>  [ show_line(X, " picked"),
                       -obj(X), +owned(X)  ]
```

- Se la condizione precedente è **falsa** e l'oggetto X è già posseduto dal robot, non può essere catturato nuovamente

```
pick(X) / owned(X) >> [ show("you've still got ", X) ]
```

- Se tutte le condizioni precedenti sono **false**, l'ultimo piano verrà eseguito: ciò implica che l'oggetto X non esiste

```
pick(X) >> [ show_line("cannot pick ", X,
                       " since it is not present") ]
```

SHRDLU plans: rilascio oggetti

Rilasciare un oggetto sul tavolo

- Se l'oggetto X è posseduto dal robot allora posiamo X sul tavolo

```
put(X) / owned(X) >>  [ show_line(X, " is now on the table"),
                        -owned(X), +obj(X) ]
```

- L'oggetto X è già sul tavolo, allora non c'e' nulla da fare

```
put(X) / obj(X) >> [ show_line(X, " is already on the table") ]
```

- L'oggetto X non è né posseduto dal robot né sul tavolo, allora l'oggetto non esiste

```
put(X) >> [ show_line(X, " does not exist") ]
```

SHRDLU plans: rilascio oggetti

Rilasciare un oggetto X su un altro oggetto Y

- L'oggetto X è posseduto dal robot, e l'oggetto Y è sul tavolo, ma Y ha un altro oggetto Z sopra di esso, allora non possiamo fare l'azione

```
put(X, Y) / (owned(X) & obj(Y) & upon(Z, Y) ) \
    >> [ show_line(Y, " has ", Z, " on its top") ]
```

- L'oggetto X è posseduto dal robot, l'oggetto Y è sul tavolo e dato che la condizione precedente è **falsa**, Y è libero, allora possiamo piazzare X sopra Y

```
put(X, Y) / (owned(X) & obj(Y)) >> \
    [ -owned(X), +obj(X), +upon(X, Y),
      show_line("done") ]
```

Case Study
“Change Coins”

“Change Coins”

- Desideriamo automatizzare il procedimento di erogazione del resto da parte di un distributore automatico di bevande
- Il distributore possiede dei serbatoi per le monete da:
  - 50 centesimi
  - 20 centesimi
  - 10 centesimi
  - 5 centesimi
- Detta *M* la cifra da restituire, il sistema dovrà erogare le opportune monete, partendo sempre da quelle di taglio più grosso e considerando l’eventualità di serbatoi vuoti

“Change Coins”

- Utilizziamo i seguenti beliefs che indicano il numero di monete presenti nei relativi serbatoi:
  - `fifty(N)` numero di monete da 50 centesimi
  - `twenty(N)` numero di monete da 20 centesimi
  - `ten(N)` numero di monete da 10 centesimi
  - `five(N)` numero di monete da 5 centesimi
- Definiamo una procedura `change(M)` che ha il compito di erogare il resto moneta per moneta
- Essa sarà una procedura ricorsiva che identifica la moneta di taglio più alto da erogare, la eroga, e aggiorna i beliefs di conseguenza

La procedura `change(M)`

- Pensiamo per “singola moneta” e per “singolo taglio”
- Sia C il numero di monete di taglio t, allora...
- ... se $M >= t$ e $C > 0$, possiamo erogare una moneta di taglio t, decrementare C, e chiamare la procedura $change(M - C)$:

Procedure Change(M);
if $(M >= t) \ \wedge \ (C > 0)$ **then**
    $C := C - 1$;
    $M := M - t$;
    $Change(M)$;
end

La procedura `change(M)`

```
# controllo monete da 50 cent
change(M) / (fifty(C) & geq(M, 50) & gt(C, 0) ) >> \
    [
        show_line("50 cent"),
        -fifty(C), "C = C - 1", +fifty(C),
        "M = M - 50", change(M)
    ]

# controllo monete da 20 cent
change(M) / (twenty(C) & geq(M, 20) & gt(C, 0) ) >> \
    [
        show_line("20 cent"),
        -twenty(C), "C = C - 1", +twenty(C),
        "M = M - 20", change(M)
    ]

...
```

La procedura `change(M)`

```
...
# controllo monete da 10 cent
change(M) / (ten(C) & geq(M, 10) & gt(C, 0) ) >> \
    [
        show_line("10 cent"),
        -ten(C), "C = C - 1", +ten(C),
        "M = M - 10", change(M)
    ]

# controllo monete da 5 cent
change(M) / (five(C) & geq(M, 5) & gt(C, 0) ) >> \
    [
        show_line("5 cent"),
        -five(C), "C = C - 1", +five(C),
        "M = M - 5", change(M)
    ]

change(M) >> \
    [
        show_line("End of change, remaning ", M, " cents")
    ]
```

“Change Coins” e SingletonBeliefs

“Change Coins” e SingletonBeliefs

- I vari piani legati alla procedura “change” necessitano di **modificare** il parametro del belief che rappresenta il taglio della moneta da erogare
- A tale scopo viene utilizzato il frammento di codice:

```
...
        -ten(C), "C = C - 1", +ten(C),
...
```

- Ossia viene **rimosso** il belief, modificato il parametro e **asserito nuovamente** lo stesso belief
- Questo perchè non è direttamente possibile modificare un parametro di un belief

“Change Coins” e SingletonBeliefs

- Tuttavia i beliefs:
  - `fifty(N)`
  - `twenty(N)`
  - `ten(N)`
  - `five(N)`

  sono caratterizzati dal fatto che possono esistere in **singola istanza**
- Cioè, dal punto di vista dell’applicazione, **non ha senso** avere, nella KB, due belief `five` con parametri diversi
- In questi casi, i belief che possono esistere in singola istanza possono essere dichiarati come `SingletonBelief`
- Pertanto, l’operazione di “aggiunta” ($+$) su un belief di questo tipo provoca la modifica della singola istanza (qualora essa sia presente)

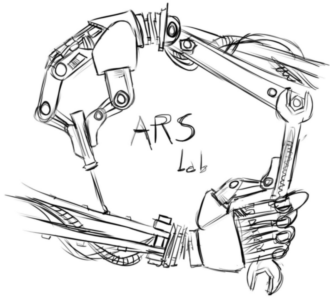
`change(M)` con SingletonBelief

```
class fifty(SingletonBelief): pass
class twenty(SingletonBelief): pass
class ten(SingletonBelief): pass
class five(SingletonBelief): pass

# controllo monete da 50 cent
change(M) / (fifty(C) & geq(M, 50) & gt(C, 0) ) >> \
    [
        show_line("50 cent"),
        "C = C - 1", +fifty(C),
        "M = M - 50", change(M)
    ]

# controllo monete da 20 cent
change(M) / (twenty(C) & geq(M, 20) & gt(C, 0) ) >> \
    [
        show_line("20 cent"),
        "C = C - 1", +twenty(C),
        "M = M - 20", change(M)
    ]
...
```

Programmazione Dichiarativa in Python con PHIDIAS

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



Programmazione Sistemi Robotici