

# Progetto di Linguaggi e Modelli Computazionali (aa. 2010-2011)

## Sviluppo di un plugin Eclipse per tuProlog

*Andrea Mordenti*

*Marco Prati*

*Francesco Fabbri*

### Relazione

L'obiettivo prospettato dal progetto prevede la re implementazione dell'IDE tuProlog nella sua ultima versione 2.3.1alfa come plug-in per la piattaforma di sviluppo Eclipse Helios 3.6.2, con una perspective e features dedicate. I risultati ottenuti lungo le oltre 100 ore pro capite da noi impiegate sono soddisfacenti e abbastanza significativi.

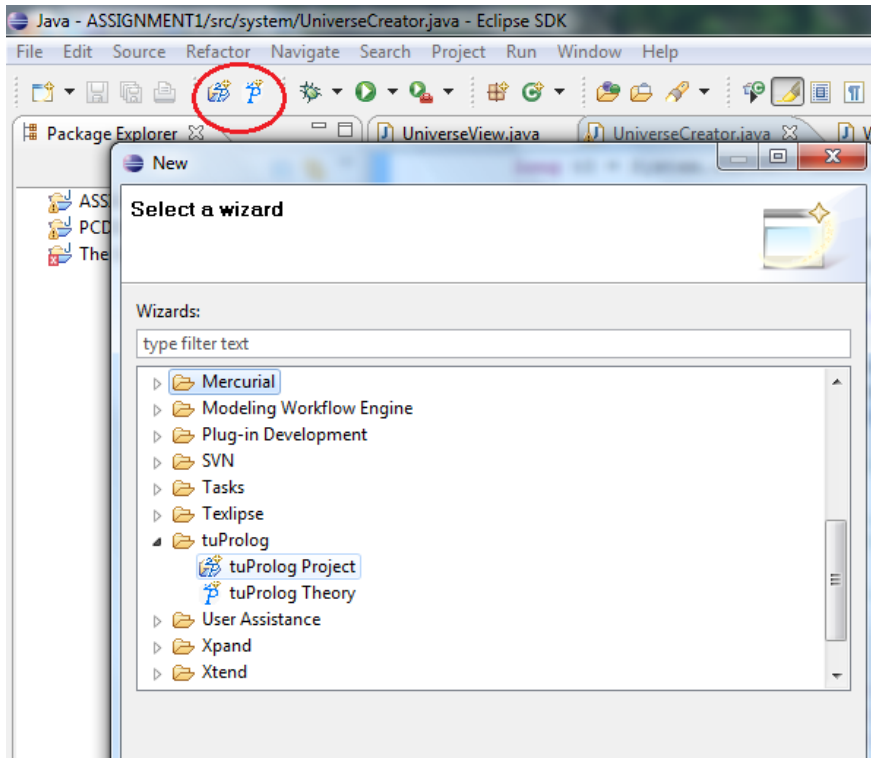
È stato prodotto un plugin piuttosto stabile e funzionante, che permette di poter programmare in linguaggio Prolog nel rispetto della sua sintassi ma con un upgrade alla versione corrente tramite l'introduzione di package di interfacce stabili all'interno dell'engine stesso (in questo modo risulta facilitata anche la scalabilità del plugin a versioni successive). In aggiunta, sono state sviluppate features che forniscono un più che valido supporto allo sviluppo software soprattutto per quanto concerne la fase di checking e individuazione degli errori: per questi sono state integrate funzionalità di segnalazione warning ed errori, anche multi linea, nella riga associata, accompagnate dalla descrizione del problema incontrato dal parser tuProlog.

È stata rivisitata anche l'operazione di esecuzione delle query, rendendola il più possibile simile a come si presentava nell'IDE tuProlog dal quale abbiamo preso spunto: si considera come ambiente di esecuzione la teoria correntemente aperta nell'editor e il suo engine associato (che mantiene le relative librerie); una volta eseguita, i risultati della query sono mostrati tramite le viste già esistenti nel precedente plugin con la possibilità di visualizzare i binding delle variabili in un apposito frame come avveniva per l'IDE classico e, in una finestra dedicata, l'AST delle clausole corrette che compongono la teoria sfruttando il parser tuProlog.

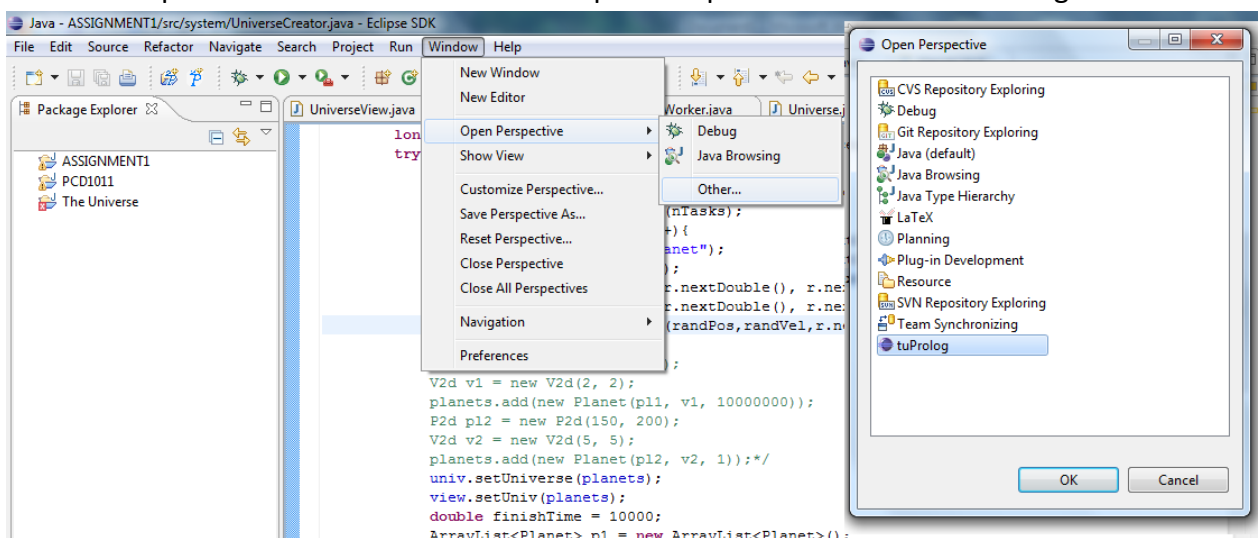
Di seguito viene mostrato l'utilizzo del plugin step-by-step, dalla creazione di un nuovo progetto alla interrogazione di una determinata teoria. Per quel che riguarda l'installazione del plugin, sarà sufficiente copiare il JAR fornito nella cartella "plugins" di Eclipse.

L'esecuzione di una query per una certa teoria, prevede prima di tutto la creazione di un nuovo progetto tuProlog e l'inserimento in esso di un file Prolog (estensione\*.pl): all'apertura dell'IDE di Eclipse saranno già presenti i pulsanti "New tuProlog Project" e "New tuProlog Theory", premendo il primo sarà possibile scegliere il nome del progetto da creare e associare ad esso le librerie Prolog desiderate (saranno già listate quelle di default), alternativamente si può seguire il percorso: File -

> New -> Other -> tuProlog -> tuProlog Project. Successivamente premendo il pulsante selezionato per la creazione di un nuovo file Prolog (o, come sopra: File -> New -> Other -> tuProlog -> tuProlog Theory) possiamo andare a selezionare il “Container” ovvero il progetto tuProlog al quale apparterrà tale file, e di seguito a scrivere la teoria.



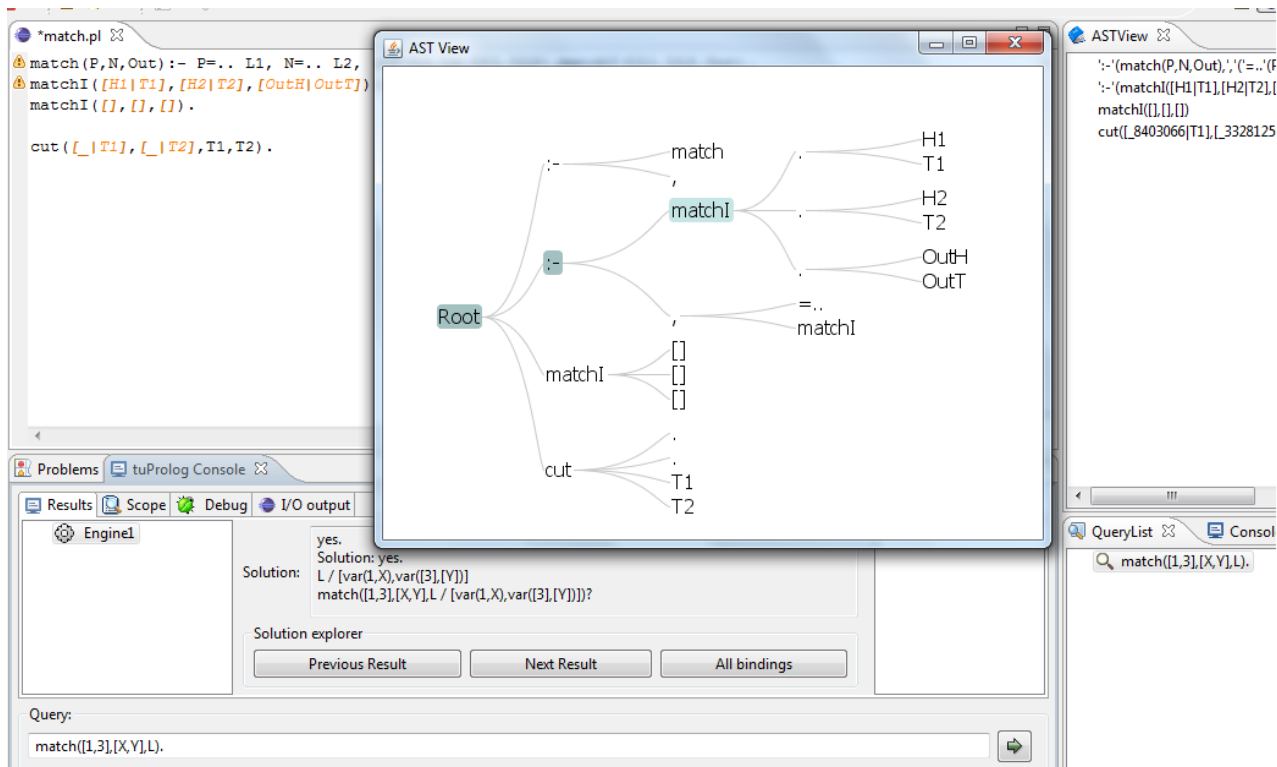
Per preparare l'IDE all'esecuzione di query Prolog sarà necessario aprire la perspective dedicata, da noi re implementata tramite Window -> Open Perspective -> Other -> tuProlog



Una volta completata la scrittura della teoria occorre salvare il file \*.pl (è necessario farlo ogni volta che si vuole eseguire una query) e poi, posizionandosi nella view in basso “tuProlog Console” si potrà scrivere la query desiderata; verrà messa in esecuzione da tastiera, premendo “Enter” oppure tramite l'apposito pulsante “Solve” di fianco alla casella di testo.

I risultati saranno così visualizzabili:

- tuProlog Console view: riporta il/ risultato/i della query eseguita insieme ai binding delle variabili (premendo il pulsante “All Bindings” in una tabella)
- I/O Output view: i messaggi d’uscita
- Query List view: vista laterale nella quale verrà riportata la lista della sequenza di query eseguite, permettendo così all’utente di poterle andare a rieseguire con un solo clic su di esse
- AST view: riporta dinamicamente le clausole inserite dall’utente e tramite la pressione sul bottone “Info” verrà mostrato l’AST di tali clausole all’uscita dal Parser



Dopo aver mostrato brevemente quello che è l’utilizzo del plugin da noi rivisitato ed esteso, sotto viene presentato quello che è stato il percorso seguito in ambito di progettazione e sviluppo che ci ha permesso di arrivare al prodotto finito: da una prima fase di analisi dei requisiti e dei possibili tool da utilizzare per lo sviluppo di plugin come le proprietà da noi pretese, fino allo studio dell’esistente plugin per una re ingegnerizzazione dello stesso.

## Progettazione e Sviluppo

Come idee progettuali portanti per la scelta di questo tipo di lavoro sono stati considerati, da una parte alcuni aspetti riguardanti il supporto ai *software developers* che lavorano in ambiente Eclipse e dall’altra, la necessità di re ingegnerizzare l’interfacciamento di tuProlog con l’utente.

Considerando il primo aspetto, grazie alla intrinseca modularità di Eclipse (una delle sue caratteristiche maggiormente utili e potenti) si è pensato che fosse utile progettare un plug-in apposito in modo da:

- Non avere un IDE separato con il quale sviluppare in tuProlog
- Creare progetti ordinati con una più chiara visualizzazione dei file utilizzati ed eventuali relazioni tra essi
- Avere una visione trasparente delle query effettuate

L'altro punto concerne l'evidente necessità di completare/estendere il supporto che l'IDE di tuProlog fornisce: difatti mancano o sono stati male implementati (o solo in parte) alcuni servizi fondamentali che assistono lo sviluppatore nella progettazione come:

- Syntax highlighting
- Content assist
- Segnalazione di riga per errori sintattici

Una volta che sono state individuate quelle che sono le proprietà da noi desiderate per il plug-in si è passati ad analizzare quali fosse la migliore tra le varie alternative progettuali considerate.

Si sono dunque cercate, quelle che potevano essere le metodologie più performanti per poter costruire un plug-in per un nuovo linguaggio di programmazione e per riuscire a fornire le funzionalità sopra citate sia a livello di codice che prettamente legato alla costruzione di una perspective adatta.

La prima strada battuta (che è quella alla quale è stata dedicata la parte più importante del tempo) è stata quella che prevede l'utilizzazione di una piattaforma esistente per la generazione automatica di lexer e parser a partire dalla grammatica del linguaggio logico Prolog, con conseguente realizzazione di parte delle features target.

Anche in questo caso sono stati valutati diversi tools i quali incorporavano, a grandi linee, funzionalità da noi desiderate:

- Xtext
- DLTk
- IMP

## **Xtext**

<http://www.eclipse.org/Xtext/>

*(Studiato e testato da Prati, Mordenti, Fabbri, relazione di Mordenti)*

È un language development framework che permette di creare sia linguaggi general purpose, che piccoli linguaggi domain-specific (DSL) semplicemente scrivendo la EBNF (in maniera JavaCC-like) del linguaggio che desideriamo, successivamente si fa girare il file \*.mwe2 il quale genera

automaticamente il parser per il nostro linguaggio, il lexer e l'AST () implementato usando l'Eclipse Modeling Framework (EMF) che fornisce diverse funzionalità utili nel contesto di Xtext. Avviando quindi il progetto come una applicazione Eclipse abbiamo il workbench con plug-in già installato, possiamo creare nuovi file con estensione da noi definita in precedenza e abbiamo subito pronto un editor che supporta il nostro linguaggio di programmazione con funzionalità di content assist, syntax highlighting e sottolineature in presenza di errori sintattici.

Xtext fornisce poi possibilità per poter estendere ulteriormente il plug-in come funzionalità di quick fixes per gli errori, l'inserimento di menu/viste, project wizard e syntax coloring per poter associare un determinato colore a certe keyword o parti dell'input file.

Quindi abbiamo un tool che ci permette di poter ottenere molto velocemente un plug-in che supporti un nostro linguaggio domain-specific con notevoli potenzialità nell'estensione di questo in termini di features.

Il problemi incontrati per lo sviluppo del plug-in sono tutti legati alla natura del linguaggio tuProlog.

In primo luogo prevede la definizione dinamica di operatori (binari e unari) e quindi costruendo il nostro parser con EBNF statica non verranno parsati operatori dichiarati precedentemente nel codice segnalandoci un errore sintattico, visto che il nuovo operatore non è tra le "frasi" della grammatica iniziale (problema insormontabile visto che le librerie di tuProlog prevedono un massiccio quantitativo di operatori definiti dinamicamente, vedi DCG...).

Altro comportamento indesiderato è dovuto al fatto che Prolog non è un linguaggio LL(1) e in questo caso quando si hanno più alternative per una qualche regola della grammatica, nella fase di parsing si considera solo la prima e ci si dimentica delle altre (problema nel riconoscimento dei float: il punto è considerato come token che termina un fatto della teoria Prolog oppure come inizio dei decimali del numero?).

Considerate queste problematiche è stata scartata questa via di implementazione non essendo possibile poter rimediare a queste inefficienze.

Di seguito viene riportata la sintassi del linguaggio TuProlog ricavata dall'analisi del parser esistente in forma simil EBNF XText (senza la definizione degli operatori dinamici).

```
grammar it.unibo.TuProlog with org.eclipse.xtext.common.Terminals
```

```
generate tuProlog http://www.unibo.it/TuProlog
```

```
Theory:
```

```
    (facts+=Fact ' ')*;
```

```
Fact:
```

```
    left=ExprList (':-' right=NotEmptyExprList)?;
```

```
ExprList:
```

```
    (exprs+=Expr) ? (',' exprs+=Expr)*;
```

```
NotEmptyExprList:
```

```
    exprs+=Expr (',' exprs+=Expr)*;
```

```

Expr:
    left=Expr2 (op=Op right=Expr)? ;

Expr2:
    term=Term | op=Op unOpExpr=Expr;

Term:
    atom=Atom | var=Variable | num=Number | list=List | str=Str | '!';

Atom:
    functor=Functor ('(' args+=Term (',' args+=Term)* ')')?;

List:
    '[' (head+=Term (',' head+=Term)* ('|' tail=Term )? )? ']';

Str:
    '"' (Letter | Number | Symbol)* '"';

Functor:
    LCaseLetter (Letter)*;

Variable:
    '_' | UCaseLetter (Letter)*;

Letter:
    UCaseLetter | LCaseLetter;

Number:
    (Digit)+ ('.' (Digit)+)?;

Op:
    (Symbol | Letter)+;

terminal UCaseLetter: 'A' .. 'Z';

terminal LCaseLetter: 'a' .. 'z';

terminal Digit: '0' .. '9';

terminal Symbol: '\\' | '$' | '&' | '^' | '@' | '#' | ':' | ';' | '=' |
'<' | '>' | '+' | '-' | '*' | '/' | '~';

```

## DLTK (Dynamic Languages ToolKit)

<http://www.eclipse.org/dltk/>

*(Studiato e testato da Prati, Mordenti, Fabbri, relazione di Prati)*

Dynamic Language ToolKit è un framework che si pone alla base della creazione di un IDE basato su uno specifico linguaggio. Ciò che fornisce DLTK, alla pari di Xtext, è un set di strumenti che permettono di costruire in maniera incrementale un'IDE per uno specifico linguaggio che comprenda tutte le feature che uno sviluppatore Java ha con JDT sotto Eclipse, ovvero content-assist (grazie all'AST generato dal parser DLTK), syntax highlighting, quickfix, viste, perspective ecc...

Nel nostro caso abbiamo iniziato a guardare DLTk in quanto volevamo farci un'idea di quale fosse lo strumento più adatto al nostro scopo, ma che al contempo rimanesse semplice e facilmente manipolabile.

Un primo sguardo al framework DLTk ci ha fatto però capire che non poteva essere incluso nel nostro progetto in quanto, analizzando una presentazione (reperibile all'indirizzo <http://www.eclipse.org/proposals/dltk/Dynamic%20Languages%20Toolkit%20-%20Creation%20Review%20Slides.pdf>) creata da un membro del team di sviluppo di DLTk, ci siamo imbattuti in una slide che chiariva il ruolo di DLTk nei confronti dei più comuni linguaggi dinamici:

## Which Languages is DLTk for?

<b>Perfect Fit</b> <ul style="list-style-type: none"><li>—Python</li><li>—Ruby</li><li>—Perl</li><li>—PHP</li><li>—VBScript</li><li>—Smalltalk</li><li>—ActionScript 3</li><li>—ECMAScript 4</li><li>—and much more...</li></ul>	<b>Good Fit</b> <ul style="list-style-type: none"><li>—TCL</li><li>—Lua</li><li>—ECMAScript 3</li><li>—Objective C</li><li>—C++</li></ul>
	<b>Don't Fit</b> <ul style="list-style-type: none"><li>—Lisp</li><li>—Prolog</li><li>—Scheme</li></ul>

Copyright © 2006, Xored Software, Inc. — Made available under the EPL v1.0

15

Nel nostro caso quindi DLTk non sarebbe stata una scelta vincente e grazie alla preventiva documentazione abbiamo evitato ulteriori approfondimenti che sarebbero poi sfociati in tempo perso, cosa che invece non è risultata possibile con Xtext.

## IMP (IDE Meta-tooling Platform)

<http://www.eclipse.org/imp/>

*(Studiato e testato da Prati, Mordenti, Fabbri, relazione di Fabbri)*

Piattaforma ideata da IBM allo scopo di semplificare e velocizzare il processo di sviluppo di IDE per linguaggi di ogni tipo, lasciando concentrare lo sviluppatore sul linguaggio e scaricandolo dalla conoscenza dell'infrastruttura API di Eclipse.

IMP si contraddistingue dalle altre piattaforme per versatilità, l'unico requisito essenziale è un AST (Abstract Syntax Tree) e un mapping di esso sul testo. L'AST può essere generato tramite tool o proprietario, ed eventualmente basato su EMF (Eclipse Modelling Framework), ed anche relativamente al parser IMP lascia la libertà di generarlo con tool come LPG, ANLR, JavaCC o di

utilizzarne uno proprietario nel caso ad esempio questo sia già disponibile (filosofia *“Don’t reinvent the wheel”*).

La piattaforma fornisce diverse features utili per l’editing come: syntax highlight, outline view, code folding, hover help e content assist. Le feature possono essere implementate indipendentemente l’una dall’altra, specializzando classi astratte del framework. Per facilitare l’implementazione IMP mette a disposizione per ogni feature una classe scheletron che contiene anche la documentazione su come personalizzarla, in generale tutte le feature sono implementate visitando l’AST.

Nell’ottica dello sviluppo del plugin per tuProlog abbiamo valutato due possibili approcci: generazione di parser e AST a partire da grammatica LPG (The LALR Parser Generator) e integrazione dell’esistente parser di tuProlog. Il primo approccio è simile a quello utilizzato da Xtext con la differenza che permette di ottenere da una grammatica tipo BNF in modo automatico un parser look-ahead left-to-right (più potente del parser ANTLR usato da Xtext) implementato in Java e con generazione di AST. Tuttavia questo approccio presenta due problematiche, la prima dovuta alla scarsissima documentazione di LPG, la seconda intrinseca nella sintassi stessa di tuProlog in riferimento agli operatori dinamici, difficilmente gestibili da un parser standard (il parser tuProlog li gestisce e riconosce a runtime). L’unico approccio attuabile rimane quindi quello di integrare il parser tuProlog con il framework IMP, questo è realizzabile implementando un’interfaccia per la gestione del parser e una per il mapping tra i nodi dell’AST e la regione di codice corrispondente. In questo caso però il parser tuProlog dovrà essere esteso con le funzionalità richieste.

Dopo aver esaminato i framework sopra citati si è giunti alla conclusione che, adottando come strategia la generazione automatica di un parser per tuProlog, la realizzazione del plugin sarebbe risultata inefficiente per le specifiche da noi preposte nella pianificazione del progetto. In particolar modo per quanto riguarda la gestione degli operatori dinamici di cui tuProlog fa un uso massiccio (soprattutto nelle librerie).

Date queste problematiche nella generazione del parser, si è pensato allora di poter prendere una strada in cui il plugin si sarebbe basato sul parser tuProlog; senza la necessità di un framework per doverlo generare, l’implementazione del plugin allora avrebbe riguardato unicamente l’inserimento delle varie features per il supporto allo sviluppo elencate all’inizio della relazione.

È stato analizzato dunque, dopo ricerche in rete, un ambiente di sviluppo per plugin molto utilizzato e propriamente documentato: PDE.

## **PDE (Plugin Development Environment)**

<http://www.eclipse.org/pde/>

*(Studiato e testato da Prati, Mordenti, Fabbri, relazione di Mordenti)*



Il framework PDE è un ambiente che mette a disposizione strumenti per creare e distribuire plugin per la piattaforma Eclipse.

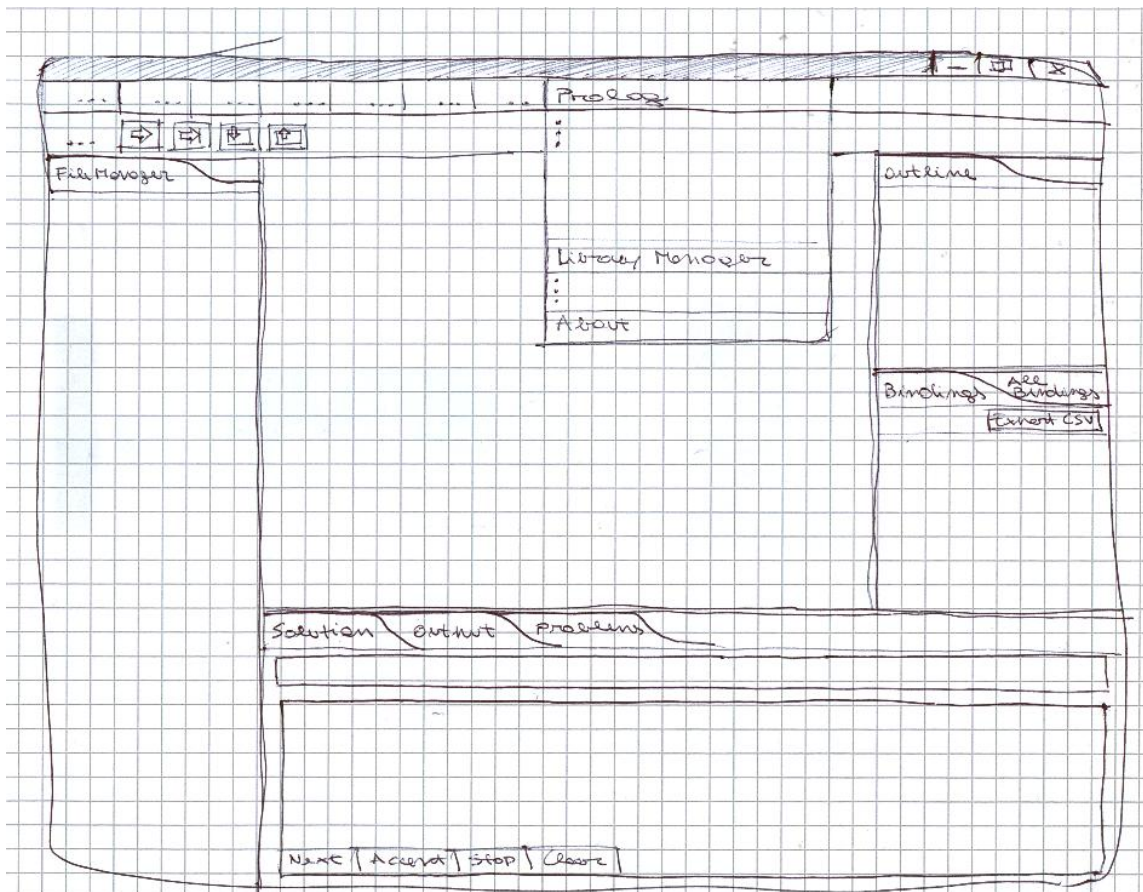
Questo framework comprende molte funzionalità, fra cui la possibilità di creare e personalizzare un'interfaccia per il plugin all'interno della piattaforma Eclipse, delle API che permettono di mantenere organizzata la documentazione riguardante il plugin e uno strumento che facilita la fase di building del plugin. Fra queste feature quella che risulta più utile ai fini del nostro progetto è quella di creazione di UI.

Il componente di PDE che permette di gestire l'interfaccia grafica mette a disposizione dello sviluppatore un sistema di estensione modulare che si basa sul concetto di "extension Points". Questi "punti di estensione" permettono di aggiungere al plugin nuove funzionalità in maniera modulare. Ad esempio è possibile creare un editor di testo associato al plugin semplicemente aggiungendo il relativo extension point.

Per le funzionalità che intendavamo sviluppare per il plugin di tuProlog su piattaforma eclipse avevamo pensato di aggiungere alcuni elementi grafici che potessero aiutare nel rendere più semplice ed immediata la scrittura di un programma Prolog:

- Editor di testi con sottolineatura della sintassi
- Wizard per la creazione di un nuovo progetto prolog
- Pagina di preferenze in cui specificare vari parametri relativi all'engine tuProlog
- Sistema di viste organizzato in perspective, con l'aggiunta di elementi come la console nella quale eseguire le query, o la vista dei binding correnti associati alla query.

Uno schema "hand made" è di seguito fornito:



I vantaggi nell'utilizzo di questo approccio sarebbero evidenti in quanto non avremmo vincoli di alcun tipo con quelle che sono le parti relative al "Core" del sistema (engine tuProlog), inoltre avremmo piene facoltà di modifica, al contrario del caso nel quale viene utilizzato in framework come xText o DLTK o IMP.

Gli svantaggi sono però stringenti, nel caso di un progetto con tempo di lavoro tarato su circa 90 ore pro capite; infatti utilizzando l'approccio PDE e quindi lo sviluppo da 0 del plugin (senza quindi avvalersi di codice generato) porterebbe ad una distensione dei tempi che andrebbe ben oltre le 90 ore di progetto, e questo di fatto esclude la possibilità di scelta di questo tipo di approccio.

Considerando quindi le ore rimaste da dedicare alla progettazione è stata presa in considerazione una ulteriore alternativa: essendo già esistente un plugin di tuProlog per Eclipse, sviluppato incrementalmente come argomento di tesi da alcuni studenti della prima Facoltà di Ingegneria, si è deciso di prenderne in considerazione l'ultima versione valutandone le caratteristiche fondamentali per poi estenderlo/re implementarlo, aggiungendo le principali features da noi pretese e eventualmente ponendo alcune correzioni a malfunzionamenti presenti.

Per poterci dividere in maniera equa il lavoro e poter sviluppare il progetto parallelamente nel modo più ottimale possibile le ore rimaste, è stato deciso di creare un progetto sul portale JavaForge.com (community di sviluppatori di progetti software open source) chiamandolo **tuclipse** (<http://www.javaforge.com/project/4226>) ; questo sito permette, a seconda dei ruoli che


si ricoprono nella pagina del progetto, di avere accesso a repository SVN condivise ed è per questo che tutto verrà sviluppato sfruttando il plugin per Eclipse di SVN.

Oltre ad avere i sorgenti condivisi, la necessità di lavorare in parallelo e molto spesso in maniera concorrente. SVN in quest'ottica permette di gestire tutte le questioni riguardanti i conflitti che si potrebbero generare nello sviluppo software da parte di più entità distribuite, fornisce infatti la funzionalità di *locking* di una certa classe/package.

Grazie a questo tool e alla pagina su Java Forge ogni membro del progetto ha quindi potuto seguire lo sviluppo del programma senza pericolo di modificare codice già in uso, oppure di usare una versione obsoleta del plugin (sulla pagina del progetto, ogni *commit* viene segnalato, con il numero della relativa versione e la descrizione delle modifiche applicate).



Una volta sincronizzati sulla condivisione e l'importazione del progetto in Eclipse da SVN si è passati all'analisi delle features da (re)implementare / aggiungere / eliminare per poter così poi dividere il lavoro; è stata quindi editata la pagina del progetto in modo da tener traccia di quelle che erano le operazioni di cui ogni membro si era preso carico, in modo da seguire mano a mano l'evoluzione delle varie sezioni.


**tuClipse » Wiki**  
Version 55 (diff) (Apr 28 17:28 by andrea.mordenti) | No comments or attachments | No child pages

[Edit](#)
[Edit in Word](#)
[Properties](#)
[New Child Page](#)
[more »](#)

**Tags:** No tags associated yet.

tuClipse is a plugin for Eclipse that want to improve user experience when developing prolog-based application within tuProlog framework

TODO:

- Upgrade a prolog 2.3.x
  - Creazione package interfacce (lato tuprolog). (FRANZ)
  - Refactoring del codice del plugin sulla base delle interfacce (gestione generica delle eccezioni da parte del plugin). (FRANZ)
  - Modifica del parser tuprolog per ripristinare l'errore di linea. (FRANZ)
- Query
  - Rendere stabile e corretta la funzionalità. (MARCO feat. MORDE)
  - Rivedere la view 'console' integrando il campo di query (come nell'IDE java), dato che attualmente i query sono gestite in una view separate e immesse tramite dialog box. (MORDE)
  - Rivisitazione dei tab per visualizzare gli output e il risultato della query, separatamente. (MORDE)
  - (New) Refactoring dell'esecuzione delle query: automatica selezione dello scope corrente da associare alla query da eseguire. (MORDE)
- Notifica degli errori
  - Attualmente non supporta gli operatori dinamici definiti nella teoria del file corrente. (FRANZ)
  - Segnalazione degli errori successivi al primo. (MARCO)
  - Segnalazione di warning per i termini in body non definiti nella teoria. (FRANZ)
- Syntax highlight
  - Token coloring dinamico per operatori e funtori. (MARCO)
- Outline view
  - Le view 'Method' e 'Theory' che mostrano rispettivamente i fatti della teoria sul file corrente così come estratti dal parser e la lista dei fatti presenti su un engine (divisi secondo un criteri comprensibile), saranno eliminati per essere sostituiti da una vista che farà vedere come si forma dinamicamente l'AST della teoria immessa dall'utente. (MORDE)
  - La nostra idea è di creare una vista 'Outline' che mostra l'AST della teoria sul file corrente (mentre il file viene editato).
- Binding view
  - Ripristinare le funzionalità di visualizzazione dei bindings in apposite view. (MORDE)

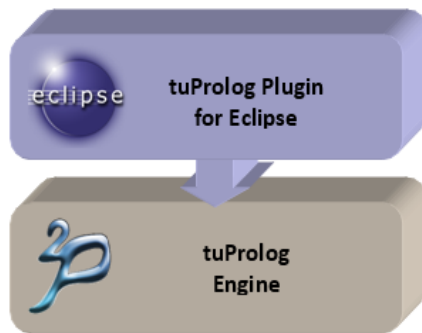
Sotto riportiamo l'elenco delle modifiche essenziali da noi apportate al plugin esistente con i relativi commenti per l'implementazione, suddivise per sezioni e sottosezioni di sviluppo in termini di classi e package del programma.

## 1. Upgrade del plugin alle nuove versioni di tuProlog 2.3.x

### a. Reingegnerizzazione del plugin

*(Analizzato e implementato da Fabbri, relazione di Fabbri)*

Il problema principale del plugin esistente è che funziona solamente con engine tuProlog fino alla versione 2.2. Questo succede perchè il plugin stesso utilizza direttamente i componenti dell'engine (parser e interprete), e quindi dipende dalle classi contenute nel jar alla versione 2.2 (come si può vedere nell'immagine sottostante). Nelle versioni 2.3.x sono cambiate interfacce e comportamento di alcuni componenti e questo ha reso il plugin inutilizzabile con queste versioni.

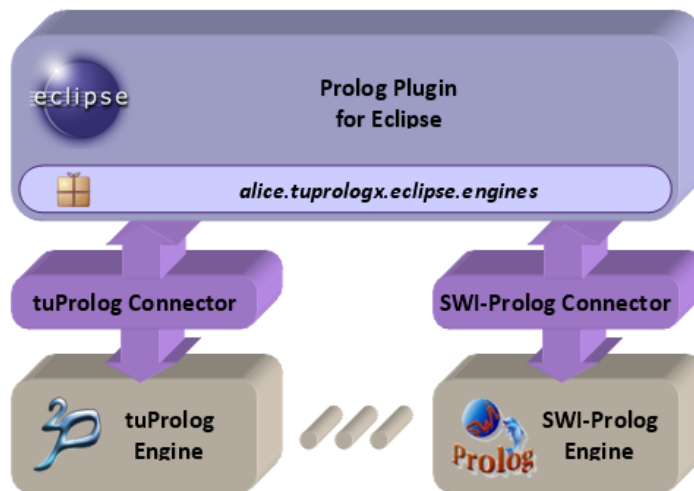


Quello che abbiamo fatto affinché il plugin sia utilizzabile per tuProlog alle versioni 2.3.x ed immune a tutte le variazioni successive è stato di introdurre un package di interfacce stabili all'interno dell'engine stesso, e di utilizzare poi dal plugin solo le interfacce e non i componenti direttamente. Il package introdotto è "alice.tuprolog.interfaces" e contiene interfacce e classi "factory" per la gestione indiretta dei componenti della business logic utilizzati dal plugin: Parser, Prolog, Engine, OperatorManager, PrimitiveManager. I componenti di modello, come ad esempio le tassonomie che rappresentano i termini prolog (Term, Struct, Var, Num, ecc...) contenuti nel package "alice.tuprolog" e gli eventi generati dai componenti di business logic dell'engine contenuti nel package "alice.tuprolog.event", sono stati invece utilizzati direttamente in quanto ritenuti abbastanza stabili.



Sviluppi futuri potrebbero generalizzare l'uso del plugin a diverse implementazioni di Prolog (GNUProlog, SWI-Prolog, ecc...). Affinchè ciò sia possibile dovrebbe essere introdotto un nuovo package lato plugin che generalizzi l'astrazione dell'engine

Prolog e dei componenti del linguaggio. Ogni engine dovrebbe essere poi interfacciato tramite un componente “connector” dedicato.



## b. Modifiche a tuProlog 2.3.1 beta

*(Analizzato e implementato da Fabbri, relazione di Fabbri)*

Per realizzare il plugin è stato necessario analizzare diversi componenti dell’engine tuProlog, alcuni dei quali sono stati modificati secondo le nostre esigenze per aggiungere funzionalità o correggere bachi esistenti. Sono di seguito elencate le modifiche più rilevanti.

### i. Tokenizer

Dalla versione 2.3 l’engine tuProlog in caso di errore restituisce sempre il numero di linea -1, come si può vedere scrivendo una teoria sintatticamente incorretta sull’editor Java. L’errore si verifica all’interno del Tokenizer che, attraverso la funzione “lineno” dovrebbe restituire la linea corrente. La funzione non è direttamente implementata ma ereditata dalla classe StreamTokenizer di Java che presenta il problema. L’idea è stata di quindi di mantenere un offset locale (calcolato sulla base del carattere iniziale e della lunghezza del token) per indicare la posizione del tokenizer e di incrementarla e decrementarla wrappando in appositi metodi le operazioni di consume e push-back. Attraverso l’offset è possibile quindi calcolare attraverso la funzione “offsetToRowColumn” la regione di testo (riga,

colonna) in cui è localizzato il token e attraverso la funzione override “lineno” è ora possibile leggere la corretta linea corrente. Abbiamo inoltre corretto la funzionalità di tokenizzazione che, in alcuni casi patologici, presentava loop infiniti (es. apertura di un commento multilinea senza chiusura).

## **ii. Parser**

Per realizzare le feature pensate per il plugin, come la notifica degli errori e dei warning per termini in body non dichiarati in teoria è emersa la necessità di mappare un nodo dell’AST prodotto dal parser sulla regione di testo della teoria corrispondente (riga, colonna). A tale scopo abbiamo modificato il parser facendo produrre, oltre all’AST una map tra termine prolog e l’offset sul testo della teoria corrispondente. Al momento il mapping dei termini non è completo ma si limita alle sole strutture di interesse.

## **iii. Tassonomia termini.**

L’AST prodotto dal parser tuProlog è rappresentato da una struttura ad albero costituita dalle classi Struct, Num e Var che generalizzano la classe astratta Term. Dato che diverse delle funzionalità del nuovo plugin consistono nella visita dell’AST, per sfruttare in queste operazioni al meglio il concetto di polimorfismo è stato introdotto il design-pattern “visitor” alla tassonomia dei termini introducendo nel package “alice.tuprolog” l’interfaccia “TermVisitor”.

## **2. Re implementazione delle modalità di inserimento / esecuzione delle query**

Uno degli aspetti sul quale è stata considerata necessaria una certa riorganizzazione è il generale interfacciamento che ha il plug-in esistente con l’utente per quanto riguarda le query, soprattutto per rendere l’interazione sviluppatore – perspective il più vicino possibile a quanto avveniva con l’IDE standard di tuProlog.

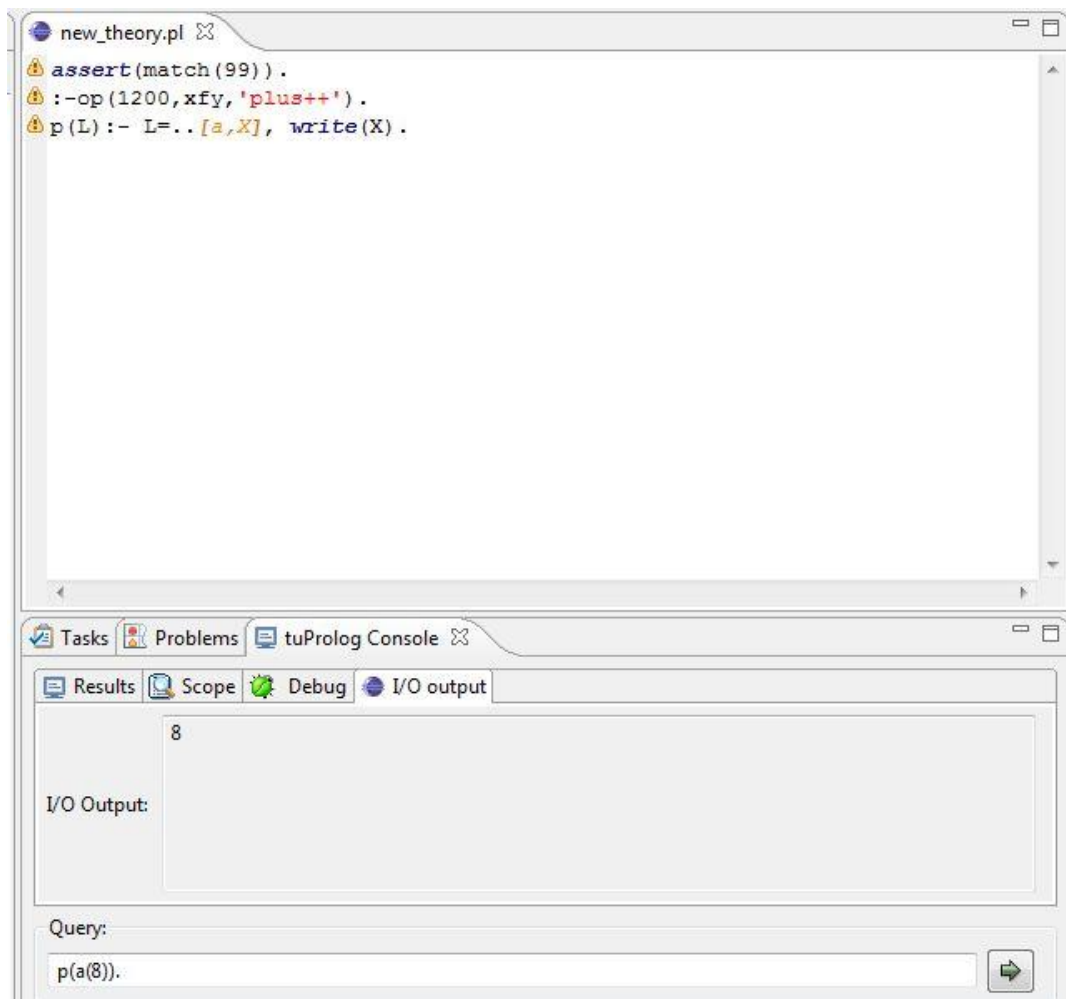
### **a. Rendere stabile la funzionalità e rivedere la view “Console”**

*(Analizzato e implementato da Prati, Mordenti, relazione di Prati, Mordenti)*



In prima battuta ci siamo concentrati sul rendere stabile e corretta la funzionalità, il quale era ovviamente un passo obbligato, soprattutto per le verifiche future delle nuove features. Questo è stato fatto correggendo un piccolo errore in fase di impostazione della query, nel metodo *executeQuery* della classe *PrologQueryFactory* in modo da poter avere sempre una teoria aggiornata in fase di esecuzione della query.

Notando che la visualizzazione degli (eventuali) Output risultava molto sacrificata nella “ConsoleView” essendo integrata insieme alla visualizzazione dei risultati è stato deciso di separare la parte dedicata agli Output inserendoli in un tab dedicato.



Successivamente ci siamo concentrati sulla re implementazione dell’inserimento delle query nel motore Prolog: la modalità precedente prevedeva l’aggiunta di una query tramite un opportuno pulsante (+) nella vista alla destra chiamata “Theory”, la quale faceva comparire l’apposito *QueryDialog* nel quale l’utente inseriva l’interrogazione nell’area di testo e successivamente (dopo aver selezionato file e motore associato) l’aggiungeva alla lista.



Questo procedimento prevede un numero eccessivo di passaggi: come per l'IDE classico, abbiamo ritenuto molto più intuitivo integrare l'area di query alla base della perspective nella vista "tuPrologConsole", sacrificando così il pulsante per l'aggiunta delle query inserite; in questo modo l'utente ha un impatto visivo immediato con il cuore dell'esecuzione, ovvero campo di inserimento e pulsante *solve*<sup>1</sup>.

#### **b. Refactoring dell'esecuzione delle query: selezione automatica dello scope**

*(Analizzato e implementato da Mordenti, relazione di Mordenti)*

Oltre all'inserimento delle query anche la sua esecuzione si presenta meno intuitiva di come è invece per l'IDE tuProlog, soprattutto per quanto riguarda la selezione da parte dell'utente dello *scope* (composto da file \*.pl e engine associato) nel quale la query andrà eseguita: tale procedimento desideriamo che non sia fatto "manualmente" dallo sviluppatore, in più ci sembra poco sensato che un utente voglia eseguire una query appena scritta su uno *scope* diverso da quello a cui appartiene il file che sta guardando in quel momento.

Per questi motivi alla pressione del pulsante *solve*, (oppure alla pressione di "invio" proprio come per l'IDE) viene fatto in modo che l'esecuzione della query comprenda anche l'inserimento dello *scope*; per fare questo è stato modificato il metodo precedentemente usato (*executeQuery*) con *executeQueryWS* (With Scope) il quale, grazie ai metodi *getActiveProject* e *getActiveFile* che permettono di recuperare file Prolog e progetto correntemente aperti nell'editor, viene aggiunto automaticamente alla query immessa lo *scope* attuale e viene messa in esecuzione, risparmiandoci così il passaggio presente in precedenza che prevedeva prima l'inserimento di una query nella QueryList, e poi l'esecuzione di questa nello *scope* associato.

Con queste piccole modifiche siamo così riusciti ad ottenere, a livello di interazione utente – ambiente di sviluppo, un comportamento molto più vicino a quello che caratterizzava la programmazione tuProlog con il vecchio IDE.



### c. Ripristino della funzionalità della visualizzazione dei bindings

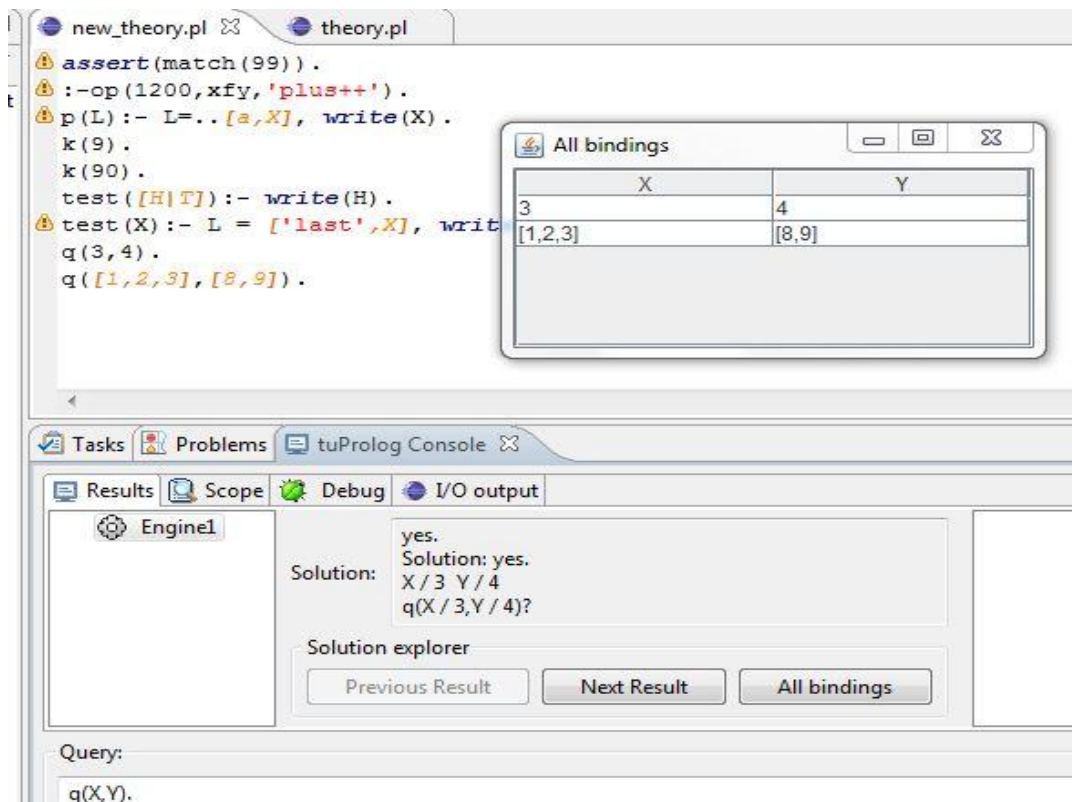
(Analizzato e implementato da Mordenti, relazione di Mordenti)

È stato ritenuto utile inoltre, ripristinare la funzionalità di visualizzazione dei bindings riguardanti le variabili specificate nella query tramite una finestra resa visibile alla pressione del pulsante “All Bindings”.

Per implementare tale funzionalità per il nostro plug-in è stato necessario aggiungere due ArrayList alla classe *PrologEngine.java*: una per collezionare la lista delle variabili della query che fanno match con clausole presenti, l'altra memorizza tutti i termini che fanno bind della teoria corrispondenti a tali variabili; dichiarate *Info* e *TermList* rispettivamente.

Le quali venivano aggiornate aggiungendo ad esse variabili/termini all'interno dei metodi *query* e *next* in cui si ottengono le soluzioni consultando i campi dell'oggetto *SolveInfo* che caratterizza ogni risultato associato ad una certa query. Sempre in tale classe introdurremo due metodi (*getSolveInfo* / *getListTerm*) con i quali sarà possibile da parte della vista della console andare a reperire tali informazioni per poi comporre la tabella dei bindings.

Quindi nella classe associata alla “tuPrologConsole” view (*ConsoleView.java*) tali metodi saranno utilizzati ad ogni “refresh” della vista per poter far sì che il listener associato al pulsante “All Bindings” costruisca la tabella con i bindings associati all'attuale query in esecuzione.



#### i. Refactoring classe `PrologQueryFactory.java`

(Analizzato e implementato da Mordenti, relazione di Mordenti)

Dopo aver implementato la gestione dell'operatore `assert` della libreria di base, che nel plugin precedente non era gestita, è stato riscontrato un malfunzionamento nella visualizzazione dei risultati e conseguentemente dei binding delle variabili: abbiamo notato che, come era stata precedentemente implementata il metodo per l'esecuzione delle query la teoria correntemente presente nell'editor veniva preso e "appeso" alla teoria già presente all'interno del motore Prolog tramite il metodo `addTheory`. Questo procedimento portava a soluzioni inesatte dove le variabili facevano bind con un numero sempre maggiore di termini anche se, di fatto, la teoria non cambiasse mai generando quindi bind multipli tra stesse coppie variabile-termini.

È stato dunque necessario andare a modificare il metodo `ExecuteQuery` nella parte relativa all'inserimento di una certa teoria al motore corrente, confrontando l'ultima teoria corretta considerata per una precedente esecuzione con la teoria digitata nell'editor e andando a rimpiazzare la precedente teoria se e solo se la teoria è stata modificata.

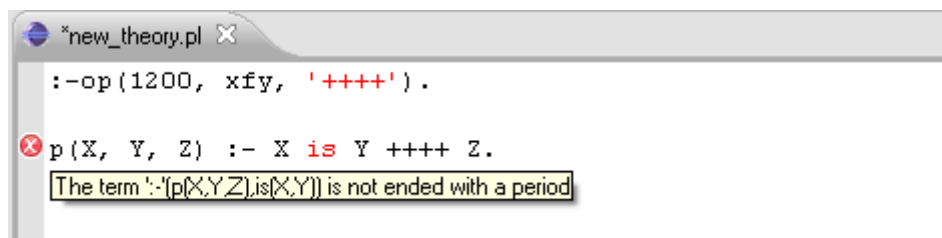
Nel caso in cui siano diverse le due teorie confrontate allora tramite *setTheory*<sup>2</sup> si inserisce la nuova teoria sulla quale verrà eseguita la query.

### 3. Notifica degli errori sintattici durante la scrittura

#### a. Supporto agli operatori dinamici

*(Analizzato e implementato da Fabbri, relazione di Fabbri)*

Un primo problema con cui ci siamo scontrati è che il parser, così come richiamato dal plugin esistente non riconosce gli operatori dinamici (lo screenshot sotto riportato rappresenta una tipica situazione di errore). Questo succede perché gli operatori dinamici sono gestiti dall'engine tuProlog tramite un apposito componente (OperatorManager) a cui gli operatori dinamici vengono aggiunti da un ulteriore componente che visita la teoria e non dal Parser stesso. Il problema è relativo solamente agli operatori dinamici definiti nella teoria corrente, infatti come si può vedere dallo screenshot l'operatore "is" viene correttamente riconosciuto mentre "++++" viene segnalato anche se correttamente dichiarato.



Per risolvere il problema abbiamo introdotto all'interno del componente di parsing del plugin una funzionalità per esplorare i termini generati dal parser alla ricerca delle definizioni di operatori dinamici `: -op(...)` all'interno della teoria. La ricerca viene ripetuta ricorsivamente anche sulle direttive `load_library`, tramite cui è possibile caricare una libreria prolog, e `consult`, attraverso cui è possibile caricare una teoria contenuta in un altro file, in modo da rendere il più possibile completa la segnalazione di errori sintattici.

#### b. Segnalazione degli errori sintattici successivi al primo

---

<sup>2</sup> metodo simile al precedente *addTheory* ma con la differenza che la teoria che si va ad aggiungere all'engine non va ad appendersi a quella già presente ma la sostituisce del tutto (precedentemente la teoria presente viene rimossa tramite il metodo *clearTheory* della classe Prolog)

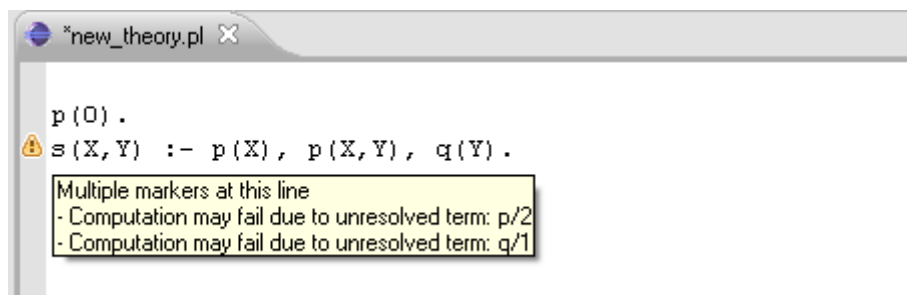
*(Analizzato e implementato da Prati, relazione di Prati)*

Questa funzionalità è stata ottenuta introducendo la memoria per le righe finora “parsate” dal parser tuProlog. Questo permette quindi di avere la conoscenza del punto in cui si è arrivati a parsare e quindi ogni volta che il parser tuProlog si ferma e riporta un errore, questo viene visualizzato e un nuovo parser si occuperà di controllare una nuova teoria che parte dalla linea successiva all’errore trovato. In questo modo possiamo avere un’indicazione dei vari errori del documento in modo da avere subito un’indicazione della correttezza globale dello stesso.

### c. Warning sui termini in body non dichiarati in teoria

*(Analizzato e implementato da Fabbri, relazione di Fabbri)*

Abbiamo esteso il controllo della teoria corrente con una funzionalità di checking in cui vengono notificati tramite warning i termini (Struct) nel body del fatto che non trovano una corrispondenza nella teoria. Ad esempio nell’immagine qui sotto si può notare che sul secondo fatto la computazione sicuramente fallirebbe in quanto l’engine tuProlog non riuscirebbe a unificare i termini  $p(X,Y)$  e  $q(Y)$ .



Questa funzionalità di checking viene avviata successivamente al parsing, se questo ha avuto esito positivo, e procede esplorando l’AST alla ricerca dei termini in body e poi verificando se questi sono presenti in teoria. La funzionalità è stata implementata nella classe “TheoryChecker” e fa uso della modifica al parser relativa al mapping dei termini sul testo sopra citata.

#### 4. Token Coloring Dinamico per operatori e funtori

*(Analizzato e implementato da Prati, relazione di Prati)*

Un utile supporto dell'editor fornito da Eclipse è quello della colorazione della sintassi. Abbiamo quindi pensato di effettuare una colorazione personalizzata sulla base di quelli che sono operatori definiti dall'utente, oltre alla colorazione (già presente nel plugin) degli operatori di libreria.

Un primo problema è stato quello di avere una segnalazione ogni volta che l'utente aggiunge un operatore dinamico. Questo è stato ottenuto aggiungendo al PrologParser una funzione di notifica di eventi "Nuovi Operatori". Sono stati quindi introdotti sia l'evento sia l'interfaccia dei listener, in modo da essere notificati, una volta registrati presso l'Operator Manager dell'engine, dell'aggiunta di un nuovo operatore.

Una volta ottenuto la possibilità di avere una notifica per ogni operatore aggiunto, è stato introdotto un nuovo scanner, il "DynOpScanner" che non fa altro che sovrascrivere il DefaultScanner, con la peculiarità di avere nel costruttore un riferimento alla lista dei nuovi operatori da aggiungere alla colorazione.

Infine l'editor (che è registrato presso l'operator manager dell'engine come OperatorEventListener) si occuperà di riconfigurare il "Reconciler" ogni volta che un operatore verrà aggiunto, in modo da poterlo colorare.

Per poter avere una lista sempre aggiornata (soprattutto in relazione agli operatori rimossi dalla teoria) è stato necessario notificare attraverso l'evento la lista completa ogni volta che un operatore viene trovato; questo fa sì che anche in caso di rimozione di un operatore vi sia una coerente de-colorizzazione nell'editor.

Una possibile modifica futura è quella di incrementare le performance in quanto utilizzando la documentazione java, e quindi la struttura standard di riconfigurazione, si hanno performance non elevate in fatto di key-typing (ripetizione caratteri lenta).

#### 5. Outline Views

##### a. Rimozione view superflue e aggiunta della AST View

*(Analizzato e implementato da Mordenti, relazione di Mordenti)*

Riguardando le funzionalità delle viste già presenti nella parte destra della perspective del plugin originario: innanzitutto "Theory" ricopre un ruolo poco chiaro, sembra che faccia vedere nell'engine attuale quali sono i file Prolog in esso contenuti; funzione poco utile visto che possiamo avere informazioni simili solo guardando il Package Explorer di Eclipse e per questo tale vista è stata rimossa.

Abbiamo poi mosso la nostra attenzione riguardo alla vista "Method", questa permette di visualizzare in maniera dinamica i fatti della teoria che vengono man mano inseriti. Funzionalità che fornisce poche informazioni a quanto è già scritto nell'editor di testo del file Prolog: per questo abbiamo pensato fosse più utile integrare ad esso la possibilità di ricavarsi dinamicamente un AST della teoria

composta, eliminando quindi tale vista con una vista per la visualizzazione dell'abstract syntax tree, sempre mantenendo dinamico il funzionamento.

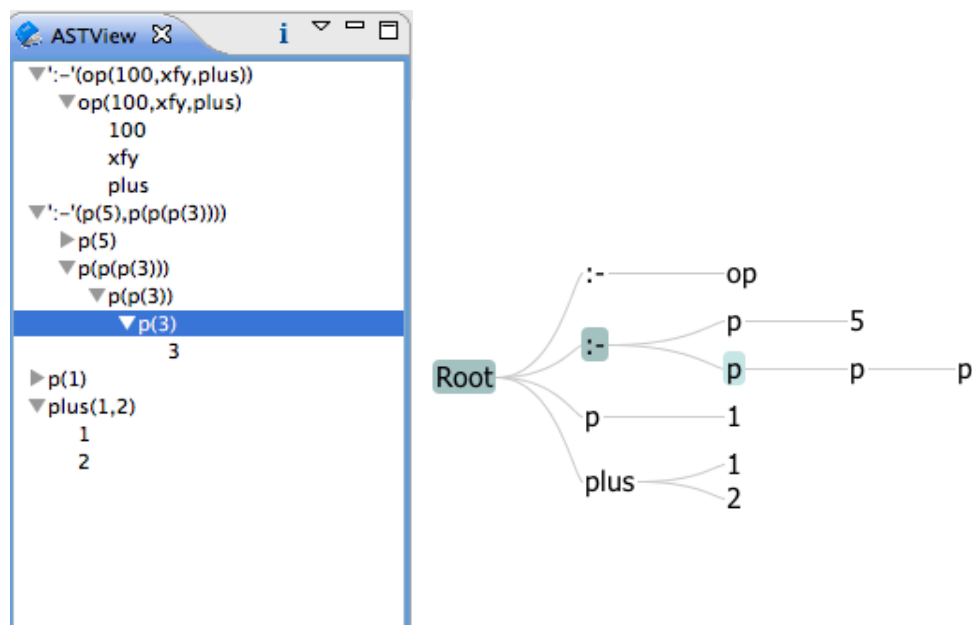
## b. AST View

*(Analizzato e implementato da Prati, relazione di Prati)*

Abbiamo realizzato una vista, inclusa nella perspective, che permette di visualizzare immediatamente la struttura della nostra teoria sotto forma di albero. L'albero è interattivo del tipo "menù a scaletta" e permette di visualizzare le gerarchie dei fatti presenti nella teoria.

Questo è stato realizzato aggiungendo una vista alla perspective che, ad ogni modifica del documento, va ad estrarre dai termini presenti nella teoria il relativo albero e lo visualizza in tempo reale nella vista.

E' stato inoltre introdotto un pulsante che, grazie all'ausilio di una libreria esterna, permette di graficare quello che è l'albero e di navigarlo in maniera interattiva. L'albero che si può vedere in figura non è completo proprio per la sua interattività, che permette di espanderlo nei punti desiderati senza comprometterne la visibilità, in quanto una visione di insieme potrebbe risultare difficile da visualizzare sullo schermo.



## Sviluppi futuri

Allo stato attuale non tutte le caratteristiche che abbiamo pensato sono implementate o sviluppate come volevamo sul plugin, lavori futuri potrebbero essere utili in diverse direzioni. Innanzitutto sarebbe da rivedere completamente il meccanismo con cui vengono effettuate le query (classi `PrologEngine`, `PrologQuery`, `PrologQueryScope`, `PrologQueryResult`,

PrologQueryFactory) per eliminare il problema per cui i termini all'interno di assert vengono aggiunti duplicati sulla teoria dinamica e per introdurre un meccanismo concorrente di esecuzione delle query utile nel caso di computazioni laboriose e necessario nel caso di computazioni in deadlock o che non terminano. Un'ulteriore feature che tornerebbe molto utile agli utenti, potrebbe essere un content assist dinamico che propone all'utilizzatore operatori, primitive e rispettiva arità.

Un ultimo ma più complicato punto su cui si potrebbe lavorare sarebbe quello descritto al punto 1.a in modo da rendere il plugin eclipse compatibile per differenti engine prolog in modo da coprire le diverse preferenze degli utenti e poter effettuare analisi sulle performance.

## Conclusioni

Il progetto a causa delle difficoltà iniziali, e di impegni derivati dalla concomitanza con il secondo ciclo di lezioni si è protratto per 4 mesi circa. Le nostre attività si sono suddivise in una prima fase di analisi e test su piattaforme esistenti, sulla quale abbiamo lavorato in gruppo per circa 45 ore a testa e una seconda fase di estensione del plugin eclipse per tuProlog esistente, sulla quale abbiamo lavorato individualmente impiegando circa 55 ore, per un totale di circa 100 ore a testa. Il plugin, allo stato attuale, risulta funzionante, sufficientemente stabile, e presenta le seguenti caratteristiche:

- a. Possibilità di gestire progetti e teorie prolog in maniera semplice ed immediata grazie al file-manager integrato dell'IDE Eclipse.
- b. Scrittura della teoria assistita dal controllo di correttezza della sintassi con segnalazione real-time degli errori sintattici (anche su più righe) e warning per i termini in body non dichiarati.
- c. Colorazione dinamica di funtori e operatori definiti nella teoria e nelle librerie caricate.
- d. Possibilità di gestire diversi engine tuProlog, caricare selettivamente teorie e librerie ed eseguire query su di essi.
- e. Esecuzione delle query in maniera semplice, visualizzazione del risultato della query, dell'output e delle informazioni di debug, comandi per la navigazione della soluzione e la visualizzazione dei binding delle variabili.
- f. Visualizzazione dell'Abstract Syntax Tree generato dal parser sulla teoria corrente in vista integrata a albero espandibile o in finestra ad albero navigabile interattivamente.
- g. Memorizzazione e visualizzazione delle ultime query eseguite.

Lo sviluppo e l'estensione dell'IDE attuale è stato guidato dall'esperienza di programmazione prolog che abbiamo acquisito nell'ambito di questo corso. Ci siamo concentrati in particolare sulla semplicità di sviluppo di programmazione prolog, cercando di superare i problemi e le limitazioni dell'attuale IDE java, grazie anche alla completezza di funzionalità della piattaforma Eclipse. Ci auguriamo quindi che il nostro lavoro potrà essere utile agli studenti di tuProlog delle prossime generazioni.