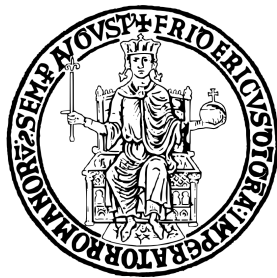


UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II



SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA
E DELLE TECNOLOGIE DELL'INFORMAZIONE

INFORMATICA

CLASSE L-31

TESI DI LAUREA TRIENNALE

Hypatia Metis per Metamath

Un traduttore da Metamath a Hypatia

Relatore

Fabio Mogavero

Candidato

Virginia Antonia Esposito

Matricola N86004987

Anno Accademico 2025–2026



HYPATIA

Tesi Triennali

Tesi Triennale

Hypatia Metis per Metamath

Un traduttore da Metamath a Hypatia

Virginia Antonia Esposito

Componente: Metis

Lavoro sviluppato nell'ambito del progetto Hypatia

hypatiafsa.github.io

github.com/hypatiafsa

Sintesi

Hypatia è un linguaggio formale e una suite di strumenti *open-source* in fase di sviluppo. Il progetto si pone l'obiettivo di colmare il *gap* semantico tra il testo scritto dall'utente e l'oggetto matematico effettivamente verificato, caratteristica di molti *theorem prover*, attraverso un'architettura basata su una torre esplicita di dialetti impilabili e sotto-traducibili tra loro.

Questo elaborato illustra la progettazione e l'implementazione di una componente del modulo Metis, uno dei toolkit di Hypatia. Nello specifico, viene presentato un traduttore, sviluppato in linguaggio Haskell, da Metamath al dialetto base di Hypatia, Sunya.

Il processo di elaborazione implementato si articola in fasi preliminari di *parsing*, decompressione e verifica del testo in ingresso, seguite dalla traduzione effettiva verso la sintassi di Hypatia.

Indice

Sintesi	i
1 Introduzione	1
1.1 I theorem prover	1
1.2 Ambiti di utilizzo	3
1.3 L'impatto dell'Intelligenza Artificiale e la crepa nella verifica	3
1.4 L'approccio di Hypatia	4
1.5 Metis e il traduttore da Metamath a Sunya	5
1.6 Esempi di traduttori tra ITP	5
1.6.1 Dedukti	5
1.6.2 OpenTheory	5
1.6.3 Tra Dedukti e OpenTheory	5
1.6.4 Tra OpenTheory e Metamath	6
1.6.5 La differenza con Hypatia	6
2 Contesto	7
2.1 Metamath	7
2.1.1 Creazione e scopo	7
2.1.2 Costrutti e Sintassi	7
2.1.3 Dimostrazioni, meccanismi di verifica e funzionamento	9
2.2 Hypatia	13
2.2.1 Scopo e miglioramenti	13
2.2.2 Costrutti e Sintassi	14
2.2.3 Dimostrazioni, meccanismi di verifica e funzionamento	24
2.3 Confronto tra Metamath e Hypatia	28
3 Problema	29
3.1 Il traduttore	29
3.1.1 Come affrontare la traduzione	29
3.1.2 Obiettivi e validazione del risultato	33
3.1.3 Esempi di dimostrazioni Metamath e Hypatia a confronto	34
4 Progettazione	37
4.1 Mappa concettuale dei requisiti	38
4.2 Diagramma delle necessità	43
4.3 Diagramma degli elementi	44
4.3.1 Sub-Elements Diagram: Il Processore e la Monade State	46
4.3.2 Sub-Elements Diagram: Il Verificatore	48
4.4 Diagramma dell'Architettura	50
5 Implementazione	51

5.1	Traduttore da linea di comando	51
5.2	Gestione delle inclusioni e risoluzione delle dipendenze	52
5.3	Il Parser	53
5.4	Il Processo di Traduzione e la Gestione dello Stato	54
5.5	Traduzione dei Costrutti diretti	58
5.6	Traduzione dei Commenti	59
5.7	Gestione degli Scope nella traduzione	59
5.8	Traduzione delle <i>Assertion</i>	60
5.9	Traduzione degli Assiomi	60
5.10	Il Decompressore delle Dimostrazioni	61
5.11	Traduzione della dimostrazione	61
5.12	Gestione delle variabili d'appoggio (Dummy Variables)	63
5.13	Inclusioni esterne e teoremi non risolti	63
5.14	Flusso di Esecuzione (Sequence Diagram)	64
6	Validazione e test	67
6.1	Test effettuati	67
6.1.1	Struttura e Operazioni Metamath	67
6.1.2	Decompressore Metamath	68
6.1.3	Parser Metamath	68
6.1.4	Verificatore Metamath	69
6.2	File di test Metamath	71
6.2.1	Gestione degli errori strutturali: <code>anatomy-bad1.mm</code>	71
6.2.2	Traduzione nominale: <code>anatomy.mm</code>	71
6.2.3	Risoluzione delle inclusioni: <code>demo0-includer.mm</code>	72
6.2.4	Stress test di unificazione: <code>big-unifier.mm</code>	73
6.2.5	Sistemi logici estesi: <code>hol.mm</code>	73
6.3	Test con il Verificatore	73
7	Conclusioni	75
	Bibliografia	76

Introduzione

1.1 I theorem prover

L'idea di delegare a uno strumento computazionale la verifica di dimostrazioni formali nasce quasi con la nascita stessa del computer. Oggi, il panorama di questi strumenti si divide in due grandi famiglie metodologiche: gli *Automated Theorem Prover* (ATP) e gli *Interactive Theorem Prover* (ITP) detti anche *proof assistant*. La differenza tra i due approcci è sostanziale e risiede nel ruolo assegnato all'utente umano.

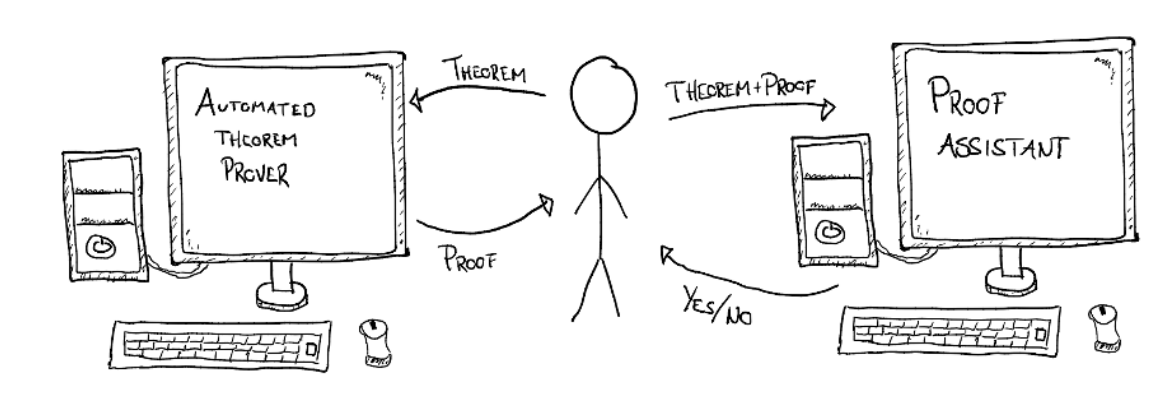


Figura 1.1: Differenza tra ATP e ITP[43].

Gli ATP offrono procedure per generare dimostrazioni dato un insieme di regole o assiomi. Già tra gli anni '50 e '60, venne presentato un primo programma in *The Logic Theory Machine* da Newell, Shaw e Simon[38] capace di provare la correttezza di alcuni teoremi presenti nel *Principia Mathematica*[46]. Il vero punto di svolta arrivò però nel 1965, quando Alan Robinson introdusse il principio di risoluzione per la logica del primo ordine[41]. Utilizzando il metodo dell'unificazione, sfruttando la sostituzione per rendere due espressioni identiche, trovare una dimostrazione si riduce al combinare regole per crearne di nuove e trovare la soluzione. Gli ATP moderni (come Vampire[31]) riescono a sintetizzare velocemente dimostrazioni complesse a partire da un set di assiomi prestabiliti.

Gli ITP, al contrario, sono dipendenti dall'utente. Possono assistere la costruzione del *proof object* tramite interazioni, tattiche e piccole automazioni, oppure elaborare una dimostrazione già fornita, mentre il verificatore ne certifica la correttezza. Questa tradizione prende forma a cavallo degli anni '70 con il sistema Automath ideato da Nicolaas de Bruijn[7, 12, 10, 8, 9, 11] e l'architettura LCF (*Logic for Computable Functions*) sviluppata da Robin Milner[22]. Da queste radici sono scaturite le fondamenta per le moderne famiglie di *proof assistant*, tra cui Mizar[37], il ramo HOL (che include HOL Light[26], HOL4[42] e Isabelle[39])[23], Coq (Rocq)[47], Matita[1], Metamath[34] e Lean[18, 35].

- **Automath (1967):** Iniziato da N.G. de Bruijn, i cui lavori fondamentali sul linguaggio sono stati documentati in diverse pubblicazioni tra il 1970 e il 1994. Questo progetto ha introdotto per la prima volta l'idea di trattare le dimostrazioni come oggetti formali di prima classe (*proofs-as-objects*) sfruttando l'isomorfismo di Curry-Howard, riducendo di fatto il controllo di una dimostrazione all'esecuzione di un algoritmo di *type-checking*. Automath ha definito il concetto di *Logical Framework* e il cosiddetto "criterio di De Bruijn", il quale impone che il sistema generi oggetti di prova che possano essere verificati in modo indipendente da un programma scritto dall'utente.
- **LCF (1972):** Sviluppato originariamente da Robin Milner, il sistema *Logic for Computable Functions* (Edinburgh LCF) è stato descritto nel 1979 in collaborazione con Michael J. Gordon e Christopher P. Wadsworth. In LCF, i teoremi sono rappresentati come un tipo di dato astratto (`thm`), in cui le uniche costanti sono gli assiomi e le uniche funzioni costruttrici sono le regole logiche di inferenza. Questo approccio basato su uno *small kernel* permette all'utente di programmare tattiche arbitrarie senza compromettere la validità del sistema.
- **Mizar (1973):** Sviluppato da Andrzej Trybulec e successivamente illustrato in una panoramica del 2009 da Adam Naumowicz e Artur Korniłowicz, Mizar si è distinto per la creazione di un linguaggio di input dichiarativo progettato per essere il più vicino possibile al naturale testo matematico umano. Questo sistema ha introdotto il *batch proof checking* e ha dato vita alla *Mizar Mathematical Library* (MML), il più grande e longevo archivio coerente di matematica formalizzata.
- **Famiglia HOL:** Questa famiglia rappresenta un'evoluzione diretta dell'architettura LCF, come tracciato storicamente da Mike Gordon nel 2000. Include **Isabelle**, nato nel 1986 e descritto come *generic theorem prover* da Lawrence C. Paulson nel 1994; **HOL Light**, presentato in un'introduzione tutoriale da John Harrison nel 1996; e **HOL4**, riassunto in una panoramica da Konrad Slind e Michael Norrish nel 2008.
- **Coq / Rocq (1989):** Basato sul *Calculus of Inductive Constructions*, esplorato e documentato nel 2004 da Yves Bertot e Pierre Castéran. Estende l'isomorfismo di Curry-Howard introducendo il paradigma *proofs-as-programs*, dal quale è possibile estrarre un programma funzionale eseguibile. L'elevata espressività del sistema permette la verifica di astrazioni avanzate e della propria meta-teoria, come dimostrato dai progetti Template-Coq (2018) e MetaCoq (2021).
- **Matita (2006¹):** *Interactive Theorem Prover* sviluppato presso l'Università di Bologna da Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi e Stefano Zacchiroli (a cui si è aggiunto in seguito Wilmer Ricciotti).
- **Metamath (1992²):** Creato originariamente da Norman Megill, definisce un linguaggio informatico progettato esplicitamente per la stesura e la verifica di prove

¹Lo sviluppo del sistema è iniziato nei primi anni 2000, con prime presentazioni pubbliche attorno al 2006, sebbene il *paper* di riferimento definitivo nella comunità accademica sia stato pubblicato nel 2011.

²Sebbene lo sviluppo teorico del linguaggio fosse iniziato attorno al 1990, le prime dimostrazioni formali all'interno del database principale `set.mm` recano la data del 1992. La prima stesura del manuale di riferimento risale al 1997, mentre una sua importante e definitiva revisione è stata pubblicata nel 2019 in collaborazione con David A. Wheeler.

matematiche.

- **Lean (2013³)**: Sviluppato originariamente da Leonardo de Moura. Nel 2021 si è evoluto nella versione Lean 4, co-sviluppata insieme a Sebastian Ullrich, la quale ha unificato le funzionalità di *theorem prover* con quelle di un vero e proprio linguaggio di programmazione.

1.2 Ambiti di utilizzo

L'impiego dei *theorem prover* ha ormai trascorso i confini della logica pura per diventare un'infrastruttura critica nell'ingegneria del software. Storicamente, questi sistemi sono stati impiegati per certificare teoremi matematici estremamente complessi, come il Teorema dei Quattro Colori [21] in Coq o la Congettura di Keplero (progetto Flyspeck)[25] in HOL Light e Isabelle.

Oggi, grazie alle loro garanzie di affidabilità, gli ITP vengono regolarmente impiegati in ambienti industriali e critici. Strumenti come PVS[17] e HOL Light sono utilizzati per validare, rispettivamente, complessi protocolli di volo per la NASA[36] e progetti di nuovi microchip per Intel[27]. Allo stesso modo, la dimostrazione automatica ha permesso la certificazione formale di sistemi operativi interi, come dimostra la verifica del microkernel seL4 condotta tramite Isabelle[30]. Questo ha trasformato il settore in un ambito di studio vastissimo e in continua evoluzione.

Esistono infatti scenari in cui una dimostrazione errata può avere conseguenze molto gravi, rendendo questi strumenti risorse molto utili. Il caso più emblematico è quello della crittografia, disciplina che affronta il problema della comunicazione sicura. Oggi la quasi totalità di azioni che eseguiamo su Internet è circondata dalla crittografia, che sia iscriversi a un social fino ad arrivare alle elezioni politiche.

Data l'importanza della sicurezza di queste informazioni possiamo renderci conto di come non basti più cercare di testarne la sicurezza provando solo a corromperla, ma è necessaria una verifica formale. Poiché molti algoritmi di sicurezza si riducono a calcoli matematici complessi (come avviene per l'algoritmo a chiave pubblica RSA e la fattorizzazione di grandi interi), vengono in aiuto i *theorem prover* per certificarne la sicurezza.

Alcuni *proof assistant* come *EasyCrypt*[5] sono nati proprio per questo e per fornire verifiche automatiche (*machine-checked*) delle prove di sicurezza. Nel corso dell'ultimo decennio, questi sistemi hanno validato sia primitive crittografiche di base sia protocolli estremamente complessi, come il TLS (alla base del protocollo HTTPS) e sistemi di voto elettronico come Helios (verificato da *EasyCrypt*[16]).

1.3 L'impatto dell'Intelligenza Artificiale e la crepa nella verifica

A cinquant'anni dalla loro concezione, le architetture basate su piccoli kernel facilmente verificabili, ideate da pionieri come de Bruijn e Milner, iniziano a mostrare i propri limiti strutturali di fronte alle sfide moderne. L'utilizzo dei *theorem prover* non è più riservato esclusivamente a una ristretta cerchia di specialisti del settore formale. Con la crescente integrazione dei sistemi di Intelligenza Artificiale, gli ITP vengono sempre più impiegati come oracoli o servizi esterni per la validazione automatica. Tuttavia, gli agenti di IA

³Il progetto *open-source* è stato avviato nel 2013 presso Microsoft Research, ma il *paper* che ne ha sancito la presentazione ufficiale alla comunità accademica ("The Lean Theorem Prover") è stato pubblicato solo nel 2015, in collaborazione con S. Kong, J. Avigad, F. Van Doorn e J. von Raumer.

tendono a ottimizzare ciecamente l'obiettivo richiesto, cadendo facilmente nel cosiddetto *reward hacking*: cercano e sfruttano debolezze, scappatoie o vulnerabilità nel sistema di verifica al solo scopo di ottenere un riscontro positivo[2, 33]. Questa dinamica ha reso le fondamenta della verifica molto meno certe rispetto al passato.

Questo nuovo paradigma di utilizzo ha portato alla luce le fragilità del metodo tradizionale, spingendo la ricerca verso meccanismi di auto-verifica e di certificazione degli stessi sistemi formali. Tra i progetti più rilevanti in questa direzione figurano:

- **MetaCoq (Template-Coq):** Un progetto mirato alla meta-programmazione certificata e alla formalizzazione del nucleo di Coq all'interno di Coq stesso, garantendo in questo modo la correttezza del sistema di base[3, 44].
- **Lean4Lean:** Lo sviluppo di un *typechecker* (verificatore) per Lean, scritto interamente nel linguaggio Lean stesso, per formalizzarne la correttezza[15].
- **Implementazione di Mizar in Rust:** La riscrittura del verificatore di Mizar nel linguaggio Rust, un'operazione che ha permesso di svelare bug silenti ed errori legati alla gestione della memoria nel sistema originale[14].
- **Meccanizzazione della semantica di HOL:** Ricerche incentrate sulla formalizzazione della semantica di HOL e delle sue estensioni conservative, in particolare per gestire in modo verificato meccanismi complessi come l'*ad-hoc overloading*[40, 20].

Il problema di fondo risiede nella distanza tra ciò che viene scritto e ciò che viene effettivamente validato. I moderni ITP operano in modo molto simile ai compilatori tradizionali: l'input fornito dall'utente attraversa una complessa "torre" di linguaggi intermedi e subisce diverse fasi di normalizzazione prima di raggiungere il kernel. Tuttavia, negli ITP i livelli intermedi rimangono spesso del tutto invisibili. Il sistema mostra soltanto lo stato finale, oscurando le trasformazioni che la formula ha subito lungo il percorso. Di conseguenza, si viene a creare una vera e propria "crepa nella verifica": l'oggetto matematico testato dal kernel è ormai strutturalmente diverso e distante dalla rappresentazione originaria pensata e fornita dall'utente.

1.4 L'approccio di Hypatia

Per minimizzare la distanza tra il testo scritto dall'utente e l'oggetto matematico formalmente verificato, l'obiettivo di Hypatia è quello di utilizzare una torre esplicita di dialetti sotto-traducibili tra loro. Invece di delegare la validazione a un singolo processo, il sistema impiegherà la gerarchia di sottolinguaggi intermedi in ogni passaggio di conversione per produrre un certificato verificabile in modo indipendente. Questo approccio riduce le incongruenze strutturali e distribuisce la responsabilità della verifica su una sequenza tracciabile di validazioni modulari.

Nello specifico:

- **Sunya:** Il nucleo, ispirato a Metamath. I costrutti sono lunghi, la prova facilmente verificabile passo per passo anche per l'utente.
- **Sifr:** Un livello sintattico costruito su Sunya che riduce la verbosità per l'utente.
- **Zero:** Il dialetto più avanzato, concepito per sviluppi matematici su larga scala, che offre un'esperienza dichiarativa vicina a quella di Mizar o Isar, costruito su Sifr.

L'intera infrastruttura a supporto di questo ecosistema è orchestrata dal *kernel* minimale **Aristarchus**, dal modulo di interfacciamento **Metis** e dal *language server* **Pythia**.

1.5 Metis e il traduttore da Metamath a Sunya

La parte in cui ci concentreremo in questo testo è il modulo Metis, che si occupa della normalizzazione e della traduzione tra i dialetti e dai linguaggi esterni.

In particolare la traduzione partirà dal linguaggio Metamath.

Questa traduzione rappresenta il primo ponte degli altri linguaggi verso Hypatia, come linguaggio di verifica e certificazione comune a sistemi già consolidati.

1.6 Esempi di traduttori tra ITP

Esistono già altri progetti che prevedono la traduzione tra ITP, come il framework **Dedukti** [4] e **OpenTheory** [29]. Questi strumenti nascono con l'obiettivo di favorire l'interoperabilità e la condivisione di librerie matematiche tra diversi dimostratori interattivi, affrontando il problema della frammentazione logica.

1.6.1 Dedukti

Dedukti è un framework logico progettato per fungere da linguaggio universale per le dimostrazioni. Dal punto di vista teorico, si basa sul $\lambda\Pi$ -calcolo modulo teoria. L'obiettivo principale di Dedukti è fornire un formato comune in cui le dimostrazioni sviluppate in vari Interactive Theorem Provers (ITP) possano essere espresse e verificate [6]. Sfruttando le regole di riscrittura, Dedukti è in grado di codificare le logiche e i fondamenti di sistemi eterogenei all'interno di un unico ambiente unificato.

Il plug-in "MM to Dedukti"

Nell'ambito della traduzione verso Dedukti, un caso di studio fondamentale è la conversione della libreria di Metamath. Ledein e Butte [32] hanno esplorato la traduzione dell'universo matematico di Metamath verso Dedukti sviluppando il traduttore Mathilde. Questo sforzo mira a codificare i fondamenti e i teoremi dimostrati nell'ecosistema di Metamath all'interno del $\lambda\Pi$ -calcolo modulo, avvalendosi di due approcci complementari: una codifica profonda (*deep embedding*), aderente alla sintassi originale, e una codifica superficiale (*shallow embedding*), progettata per dedurre la tipizzazione dei simboli ed essere più leggibile per l'utente umano.

1.6.2 OpenTheory

OpenTheory è un progetto sviluppato per standardizzare e condividere librerie di teorie tra i proof assistant appartenenti alla famiglia HOL (Higher-Order Logic) [29]. Il sistema definisce un formato standard per i pacchetti di teorie, permettendo a sistemi diversi (come HOL Light, HOL4 e ProofPower) di esportare e importare teoremi in modo interoperabile. OpenTheory funge quindi da libreria standard condivisa, facilitando enormemente il riutilizzo del lavoro formale all'interno dell'ecosistema dei dimostratori basati su HOL.

1.6.3 Tra Dedukti e OpenTheory

La connessione tra l'ecosistema HOL e Dedukti è stata ampiamente studiata per permettere la condivisione delle librerie oltre i confini della singola logica di origine. Il lavoro di Thiré [45] utilizza Dedukti come framework intermediario per esportare dimostrazioni verso

altri ecosistemi. Nello specifico, avvalendosi della logica $STT\forall_{\beta\delta}$ implementata in Dedukti, è possibile tradurre ed esportare librerie matematiche verso il formato OpenTheory, rendendole di fatto fruibili ai proof assistant della famiglia HOL, oltre che ad ambienti come Coq e Matita.

1.6.4 Tra OpenTheory e Metamath

Un ulteriore ponte nel panorama della traduzione tra ITP riguarda la connessione tra la logica di ordine superiore (propria dei sistemi compatibili con OpenTheory, come HOL Light) e l'infrastruttura di Metamath. Carneiro [13] ha affrontato questo problema sviluppando una conversione formale delle dimostrazioni in formato OpenTheory verso Metamath. Poiché la famiglia HOL si basa sulla teoria dei tipi semplici e il database principale di Metamath fa comunemente uso della teoria degli insiemi ZFC, questa traduzione ha richiesto una complessa mappatura concettuale. Il progetto ha dimostrato la fattibilità del trasferimento di teoremi dall'alta automazione di HOL al linguaggio fondazionale di Metamath.

1.6.5 La differenza con Hypatia

Rispetto all'approccio basato sul λ -calcolo e sulla teoria dei tipi adottato da Dedukti, l'architettura di Hypatia risulta strutturalmente affine a quella di Metamath. Entrambi i sistemi condividono un'impostazione incentrata sulla grammatica (*grammar-first*), in cui la sintassi non è vincolata a una teoria dei tipi predeterminata. Di conseguenza, il processo di traduzione da Metamath verso il dialetto base Sunya (che si ispira proprio a questo linguaggio) mantiene un'elevata corrispondenza strutturale.

Alla luce di questa compatibilità, Metamath è stato selezionato come primo linguaggio sorgente. In più, combinando il nuovo traduttore con i sistemi di traduzione già pubblicati (come Mathilde e le conversioni basate su OpenTheory), diventa possibile importare in Hypatia un vasto *corpus* di dimostrazioni formali, collegando il sistema alla rete di file e librerie dell'ecosistema ITP.

Contesto

2.1 Metamath

2.1.1 Creazione e scopo

Metamath[34] è sia un linguaggio formale che un software ideato per archiviare, verificare e studiare dimostrazioni matematiche. Il suo scopo principale è rappresentare la matematica, in ogni suo campo, in modo chiaro e semplice. Il pubblico di riferimento originario è composto da tecnici, ingegneri e programmatori: figure professionali abituate alla logica matematica e che ne hanno già studiato la sua applicazione in modo pratico, ma che potrebbero trovare difficoltà nell'approcciarsi alla notazione matematica tradizionale tipica dei teoremi più avanzati.

Riesce a codificare qualsiasi tipo di teorema attraverso le prove (*proofs*), e le loro dimostrazioni, delle sequenze esplicite e lineari di assiomi e passi logici. Bellezza, semplicità e rigore sono i tre principi chiave che ne hanno guidato la creazione e lo sviluppo.

Ciò che lo ha reso innovativo rispetto ai *Theorem Provers* come Coq, HOL o Isabelle, risiede nel suo approccio alla dimostrazione: in questi sistemi, infatti, le proof sono scritte a tutti gli effetti come programmi che cercano o generano la prova, richiedendo una curva di apprendimento ripida e una forte conoscenza matematica pregressa.

Metamath, al contrario, risulta libero da queste restrizioni poiché agisce strettamente come *Proof Verifier*. Non possiede al suo interno alcuna conoscenza intrinseca o pre-programmata di simboli matematici, regole di deduzione o teoremi, piuttosto ne rende possibile la definizione degli stessi da zero, consentendone l'archiviazione in database strutturati (come `set.mm` o `hol.mm`) al fine di garantirne il riutilizzo.

Il sistema si limita a controllare la correttezza delle dimostrazioni fornitegli in input basandosi su un principio di pura sostituzione, verificando ed esponendo ogni singolo passaggio. Questa semplicità, unita a un set di costrutti estremamente ridotto, garantisce un'altissima stabilità nel tempo, minimizzando il rischio di fallimento o corruzione delle prove tra le diverse versioni del software.

2.1.2 Costrutti e Sintassi

I soli caratteri permessi in un file Metamath sono i 95 caratteri ASCII stampabili (i numeri e le lettere maiuscole e minuscole) e i 32 caratteri speciali:

`! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`

Più i caratteri non stampabili: lo spazio, la tabulazione (*tab*), il ritorno a capo (*carriage return*), l'avanzamento di riga (*line feed*) e l'avanzamento pagina (*form feed*).

La sintassi di Metamath è basata su una sequenza di token separati da spazi. Ogni istruzione (o *statement*) inizia generalmente con un'etichetta identificativa, una *label* (opzionale per alcuni costrutti), seguita dal tipo di costrutto e dai suoi parametri, per poi terminare obbligatoriamente con il token di chiusura `$.`.

I costrutti principali del linguaggio sono i seguenti:

- **`$( e $)`** – *Commenti*: Racchiudono il testo utile al programmatore o all'utente a cui è destinato, ma non viene incluso ai fini della validazione. Possono essere inseriti in qualsiasi punto del file, purché tra due token validi
(es. `$( Questo è un commento.$)`);
- **`$[ e $]`** – *Inclusione file*: Permettono di importare il contenuto di altri file sorgente
(es. `$[ miu.mm $]`);
- **`${ e $}`** – *Blocchi di scoping*: Definiscono un ambito locale (*scope*). Le variabili, le ipotesi e le restrizioni dichiarate all'interno di un blocco cessano di esistere al di fuori di esso, garantendo la modularità delle dimostrazioni. I costrutti `$a` e `$p` sono invece globali al momento della definizione
(es. `$_1 $_2 $_2 $_3 $_4 $_4 $_3 $_1`);
- **`$c`** – *Costanti*: Dichiarano i simboli immutabili del linguaggio (come operatori logici, $\rightarrow$ o parentesi)
(es. `$c -> ( ) |- wff $.`);
- **`$v`** – *Variabili*: Dichiarano le variabili utilizzabili come segnaposto (*placeholder*) nelle formule
(es. `$v P Q R $.`);
- **`$d`** – *Variabili disgiunte* (*Disjoint variables*): Specifica che due o più variabili non possono essere sostituite con espressioni che contengono variabili in comune
(es. `$d P Q $.`);
- **`$f`** – *Ipotesi di tipo*: (*floating*) definisce il "tipo" o la categoria sintattica di una variabile
(es. `label $f wff P $.` indica che `P` è una formula ben formata (*well-formed formula*));
- **`$e`** – *Ipotesi logica*: (*essential*) dichiara una premessa logica che deve essere vera affinché un teorema possa essere applicato
(es. `label $e |- ( P -> Q ) $.`);
- **`$a`** – *Asserzione assiomatica*: (*axiom*) Dichiara una regola o un assioma accettato come vero senza alcuna dimostrazione. Determina le basi del sistema formale in uso
(es. `label $a |- Q $.`);
- **`$p`** – *Asserzione dimostrabile*: (*provable*) Dichiara un teorema o un lemma. A differenza del costrutto `$a`, richiede obbligatoriamente una dimostrazione. La sintassi richiede il token `$=` per separare l'enunciato del teorema dalla sua dimostrazione
(es. `label $p |- Q $= ... $.`);

Ora che abbiamo definito i costrutti fondamentali di Metamath, possiamo passare al concetto di *Frame*. Ogni assioma o asserzione dimostrabile in Metamath è associato a un

Frame.

Con questo termine si intende il gruppo di ipotesi e restrizioni di disgiunzione associate alle variabili che compaiono nell'enunciato. Un frame gode di diverse proprietà fondamentali:

1. Ogni variabile contenuta in un assioma, in un teorema o in un'ipotesi logica deve essere dichiarata in un'apposita ipotesi di tipo.
2. Due ipotesi di tipo non possono dichiarare la stessa variabile.
3. L'ipotesi di tipo in cui si introduce una variabile deve precedere qualsiasi ipotesi logica o asserzione che fa uso di tale variabile.

Associato a un frame troviamo l'insieme delle *Mandatory hypotheses* (ipotesi obbligatorie), ordinate secondo la sequenza in cui appaiono. Quest'ordine, dato che le dimostrazioni sfruttano la notazione polacca inversa o RPN (*Reverse Polish Notation*), viene definito **RPN order**.

Nei teoremi più complessi si rende spesso necessario l'uso di variabili d'appoggio, note come *Dummy variables*. Per gestire questo aspetto si definisce il **Frame esteso**, che amplia il frame standard includendo le condizioni di disgiunzione e le ipotesi relative alle variabili non obbligatorie. Ogni frame standard è implicitamente un frame esteso, ma non vale il viceversa.

2.1.3 Dimostrazioni, meccanismi di verifica e funzionamento

In Metamath, una dimostrazione è costituita da una sequenza di label separate da spazi. Ogni label all'interno di una prova fa riferimento al rispettivo *statement* formale e funge da istruzione per il verificatore, guidandolo nella costruzione progressiva dell'espressione matematica. Affinché la dimostrazione sia considerata corretta, il risultato finale di questa sequenza di operazioni deve coincidere esattamente con l'enunciato (la tesi) dell'asserzione che si intende provare.

Le label ammesse in una dimostrazione devono necessariamente corrispondere a un'ipotesi (di tipo o logica) oppure a un'altra asserzione (assiomatica o dimostrabile) già definita e valida nel contesto corrente. L'ordine in cui queste label vengono elencate definisce la sequenza di operazioni da eseguire su una pila (*stack*), seguendo la logica della notazione polacca inversa (*RPN*).

Per comprendere il funzionamento di un calcolatore basato su RPN, si considerino i seguenti esempi aritmetici:

- La sequenza 2, 3, + viene valutata come $2 + 3$, producendo **5**;
- La sequenza 2, 3, 5, *, + calcola prima $3 \times 5 = 15$, poi somma il 2, risultando in 2, 15, + e restituendo infine **17**.

Il meccanismo di verifica procede in questo modo: quando il programma incontra una label associata a un'ipotesi, inserisce l'espressione corrispondente in cima allo stack (operazione di *push*). Quando, invece, incontra la label di un'asserzione, estrae (tramite *pop*) gli ultimi elementi aggiunti allo stack per farli corrispondere alle ipotesi obbligatorie (*mandatory hypotheses*) dell'asserzione richiamata.

Questo processo determina le sostituzioni esatte da effettuare sulle variabili; una volta applicate, il verificatore rimuove definitivamente le ipotesi utilizzate dalla pila e vi inserisce

il nuovo risultato generato dall'asserzione con le dovute sostituzioni. Al termine della lettura di tutte le label della prova, per certificare la validità del teorema, lo stack deve contenere un unico elemento, identico alla tesi dell'asserzione di partenza.

Di seguito inserisco un esempio di prova:

```
$c ( ) -> wff $.
$v p q r s $.
wp $f wff p $.
wq $f wff q $.
wr $f wff r $.
ws $f wff s $.
w2 $a wff ( p -> q ) $.
wnew $p wff ( s -> ( r -> p ) ) $= ws wr wp w2 w2 $.
```

Figura 2.1: Frammento di codice Metamath.

Nella Figura 2.1 sono visibili le dichiarazioni di costanti, variabili, ipotesi di tipo e un assioma. All'interno della dimostrazione dell'asserzione finale troviamo la seguente sequenza di label:

```
ws wr wp w2 w2
```

Questa dimostrazione si sviluppa in uno stack di questo tipo:

Passo 1: Il verificatore legge la label **ws**, che corrisponde a un'ipotesi di tipo, e la inserisce nello stack.

Stack location	Contents
1	wff s

Figura 2.2: Aggiunta dell'ipotesi **ws** allo stack.

Passi 2 e 3: Successivamente, vengono lette le label **wr** e **wp**. Essendo anch'esse delle ipotesi, vengono impilate sequenzialmente sopra le precedenti.

Stack location	Contents
3	wff p
2	wff r
1	wff s

Figura 2.3: Aggiunta delle ipotesi **wr** e **wp** allo stack.

Passo 4: Appare la label **w2**. Per la prima volta ci troviamo di fronte ad un'asserzione. Il verificatore quindi legge le ipotesi obbligatorie corrispondenti a **w2**, in questo caso **wp** e **wq**, e le confronta con gli ultimi due elementi nello stack, controlla se i typecode coincidono, ed esegue infine una sostituzione. Questo processo si chiama Unificazione.

In caso il verificatore non si trovi in condizioni da poter effettuare la sostituzione darà errore, e la prova viene considerata errata. Se invece l'unificazione è andata a buon fine, l'asserzione sostituita viene pushata nello stack, e le ipotesi poppate.

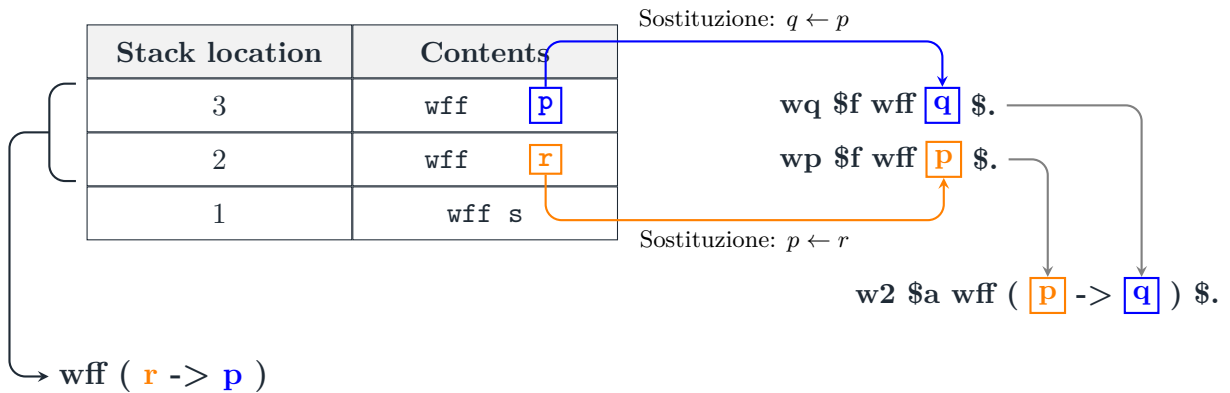


Figura 2.4: Processo di unificazione: le variabili delle ipotesi vengono istanziate con i valori estratti dallo stack.

Passo 5: Appare di nuovo $w2$. Ripetiamo lo stesso meccanismo.

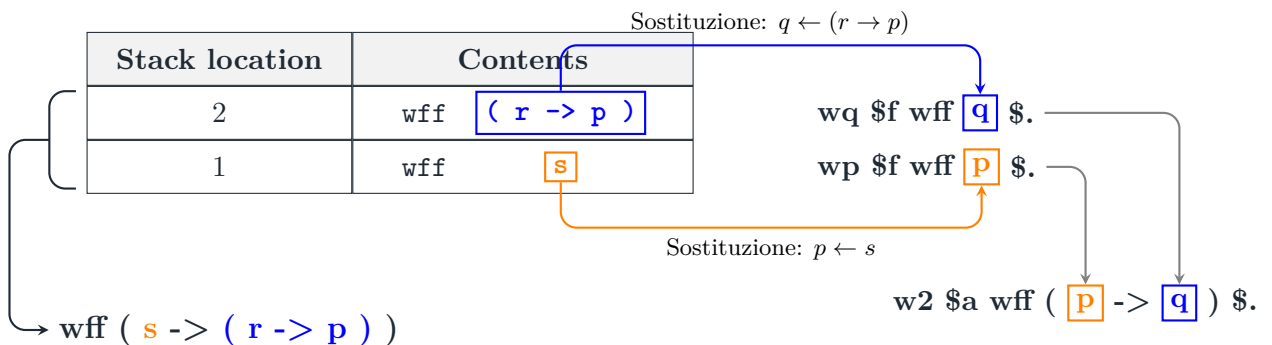


Figura 2.5: Confronto del typecode e creazione dell'asserzione sostituita (Secondo passaggio).

Alla fine lo stack contiene un solo elemento, identico alla tesi che volevamo dimostrare, quindi la prova è corretta.

Stack location	Contents
1	wff (s -> (r -> p))

Figura 2.6: Stack a fine dimostrazione: la prova è corretta poiché rimane un unico elemento coincidente con la tesi.

Prove Compresse (Compressed Proofs)

Nei database di grandi dimensioni (come `set.mm`), le dimostrazioni vengono archiviate in un formato compresso per ottimizzare lo spazio di memorizzazione.

Una prova compressa si distingue per la presenza di una parentesi tonda aperta posizionata subito dopo il token $\$=$. La sua struttura sintattica generale è la seguente:

$\$= (\text{label}_1 \text{label}_2 \dots \text{label}_n) \text{stringa_di_lettere } \$.$

La prova è divisa in due sezioni principali:

1. **Dizionario delle label:** Una sequenza di etichette racchiuse tra parentesi tonde, separate da spazi.
2. **Codifica della dimostrazione:** Una sequenza continua di lettere maiuscole (dalla A alla Z) che mappa la dimostrazione vera e propria, senza necessità di spaziature intermedie. (L'uso dei caratteri non spaziati riduce la dimensione del file di circa il 20%).

Ogni lettera maiuscola (o gruppo di lettere) nella seconda sezione viene convertita in un numero intero. Il sistema utilizza una codifica posizionale mista:

- Le lettere dalla **A** alla **T** rappresentano la cifra meno significativa (in base 20) e fungono da terminatore del numero. Assumono valori da 1 a 20.
- Le lettere dalla **U** alla **Y** rappresentano le cifre più significative (in base 5) e fungono da prefissi moltiplicatori.
- Ad esempio: $A = 1$, $T = 20$, $UA = 21$, fino a $YYT = 620$.

Mappatura degli interi Una volta decodificato l'intero k , Metamath lo associa a uno *statement* o a una sotto-prova secondo una mappatura. Siano m il numero di ipotesi obbligatorie (*mandatory hypotheses*) del teorema da dimostrare e n il numero di label elencate tra le parentesi tonde:

- **Se $1 \leq k \leq m$:** L'intero si riferisce all'ipotesi obbligatoria k -esima (seguendo l'ordine RPN). Queste label non vengono dichiarate tra parentesi poiché il programma le conosce già implicitamente.
- **Se $m < k \leq m + n$:** L'intero punta alla label che si trova nella posizione $(k - m)$ all'interno delle parentesi tonde.
- **Se $k > m + n$:** L'intero punta a una sotto-dimostrazione precedentemente salvata in memoria tramite il **Tag Z** (vedi sotto).

Ottimizzazione tramite il Tag Z Nelle dimostrazioni matematiche, ed in particolare nella costruzione di formule ben formate (*wff*), capita frequentemente che intere sequenze di passi logici siano identiche e ripetute. Per evitare questa ridondanza, Metamath introduce la lettera **Z**.

Quando incontra una **Z** posizionata subito dopo un numero (es. **BZ**), salva l'intera sotto-prova costruita fino a quel momento nello stack. La prima sotto-prova salvata con la **Z** assumerà il valore di richiamo $m + n + 1$, la seconda $m + n + 2$ e così via. Quando in seguito il programma incontrerà uno di questi interi superiori a $m + n$, non farà altro che sostituirlo con l'intero blocco di passi logici precedentemente memorizzato.

Infine, come nelle prove RPN standard, un passo logico non ancora completato può essere indicato con il simbolo **?** all'interno della sequenza di lettere.

2.2 Hypatia

2.2.1 Scopo e miglioramenti

Hypatia è sia un linguaggio formale che una suite di strumenti (*toolkit*) *open-source* attualmente in fase di sviluppo. Questo ecosistema permette la definizione, la verifica, la trasformazione e la certificazione di sistemi formali codificati all'interno del linguaggio stesso.

L'attuale architettura software è progettata attorno a un *kernel* di verifica minimale denominato **Aristarchus**. A supporto di questo nucleo, tre macro-attività sono (o saranno) implementate attraverso i rispettivi *toolkit*:

- **Metis**: si occupa delle operazioni di normalizzazione, compilazione e traduzione;
- **Anaxagoras**: è dedicato all'analisi degli artefatti di dimostrazione;
- **Pythia**: fornisce il supporto *Language Server* per l'integrazione con gli ambienti di sviluppo (IDE).

Il linguaggio formale è organizzato in una famiglia di dialetti, che vengono specificati all'inizio di un documento Hypatia affinché i relativi costrutti vengano interpretati di conseguenza. I tre dialetti base attuali prendono il nome di **Sunya**, **Sifr** e **Zero**.

Di seguito vengono illustrate le differenze:

- **Sunya**: Costituisce il nucleo conciso e costruttivo alla base della famiglia. Adotta un approccio incentrato sulla grammatica (*grammar-first*) ispirato a Metamath, in cui il sistema formale viene specificato tramite simboli, variabili, asserzioni e inferenze esplicite, senza assumere una fondazione assiomatica di base. In Sunya, infrastrutture linguistiche come alfabeti, marcatori di intervallo, delimitatori, lessemi e schemi sono dichiarazioni di prima classe ispezionabili direttamente dal verificatore. Il dialetto separa il linguaggio *host* (Hypatia stesso), il *meta-linguaggio* (il testo governato dalle regole del frammento) e il *linguaggio oggetto* (le stringhe citate su cui si ragiona). Il meccanismo di prova è un analogo puramente strutturale del *Modus Ponens*, in cui le sostituzioni sono giustificate dalla mera identità sintattica dei termini e non da un'uguaglianza semantica ambientale. Non prevedendo principi impliciti come il terzo escluso o l'assioma della scelta, Sunya opera in modo strettamente costruttivo: la correttezza coincide con la derivabilità passo per passo.
- **Sifr**: Rappresenta uno strato puramente sintattico costruito al di sopra di Sunya, ideato per ridurre la verbosità evitabile preservando i medesimi confini di fiducia (*trust boundary*). Introduce comodità ergonomiche come assegnazioni esplicite, definizioni di frasi nominate, forme di asserzione più ricche e costrutti di script di prova che legano un'affermazione (*claim*) alle evidenze per giustificarla. Queste estensioni non aggiungono potere espressivo: ogni costrutto Sifr deve obbligatoriamente subire una trasformazione deterministica (*desugaring*) in Sunya, processo che diviene parte integrante della semantica certificata.
- **Zero**: È il dialetto di livello più alto, attualmente ancora in fase di progettazione attiva. È destinato a sviluppi formali su larga scala, offrendo un vernacolo matematico flessibile e compositivo, vicino allo stile dei sistemi dichiarativi come Mizar e Isabelle/Isar. Include costrutti per la gestione di grammatiche regolari, *context-free* e

context-sensitive, permettendo di descrivere sintassi complesse direttamente all'interno del linguaggio. Zero funge da punto d'incontro: fornisce ai formalizzatori umani una struttura altamente leggibile per lo sviluppo, mentre agli sviluppatori espone chiaramente gli obblighi di traduzione e i target per i certificati nei dialetti sottostanti.

Come spiegato, ogni dialetto si basa su quello precedente a se stesso e devono poter essere ritradotti verso il basso.

L'utente sarà quindi libero di muoversi tra le **Lingua** con i dialetti come meglio crede per il lavoro che sta svolgendo, mantenendo la semantica coerente.

La grammatica di Sunya è profondamente ispirata a quella di Metamath, il quale ha ampiamente dimostrato quanto un sistema possa essere espressivo specificando la matematica unicamente tramite simboli, variabili, asserzioni e inferenze esplicite, senza la necessità di assumere una fondazione assiomatica preesistente.

Tuttavia, rispetto a Metamath, Hypatia compie un passo in più, ponendo la filosofia incentrata sulla grammatica, e priva di conoscenza intrinseca, alla base del linguaggio stesso. Sfruttando il principio stesso dell'Informatica Teorica offre infatti all'utente la possibilità di strutturare un linguaggio formale dichiarandone esplicitamente l'alfabeto, le parole, le variabili, le regole di formazione e le relazioni di derivabilità.

Seguendo proprio l'esempio dei linguaggi di programmazione come spiegato nell'Introduzione, il miglioramento di Hypatia sta proprio in questo: a partire dall'oggetto fornito, viene srotolato in sottolinguaggi, ognuno dei quali viene verificato, e alla fine viene formata una certificazione che annulla il gap presente negli altri ITP tra l'oggetto fornito e le sottorappresentazioni invisibili che vengono verificate.

In questo testo ci concentreremo su una delle funzionalità di Metis: il traduttore da Metamath ad Hypatia, in particolare nel dialetto Sunya che andremo ad approfondire più avanti.

2.2.2 Costrutti e Sintassi

Struttura del documento

Un documento può essere visto come una sequenza di 0 o più costrutti **Hypatia** o **Lingua**, che definiscono l'ambiente di lavoro:

$$\text{DOCUMENT}(\text{DIALECT}, \text{MODE}) := \\ [\text{HYPATIA}(\text{MODE}) \mid \text{LINGUA}(\text{DIALECT}, \text{MODE})]^*$$

- **\$Hypatia** (o **\$Hp**): *Selezione del dialetto*. Definisce quale dialetto utilizzare (Sunya, Sifr, o Zero) e, opzionalmente tramite il tag **\$mode**, la modalità di interpretazione per la risoluzione delle ambiguità grammaticali (*Unica*, *Omnes*, *Valida* o *Aliqua*). Il dialetto scelto influenzerà il modo in cui il Parser interpreta il Document innestato. Sintassi:

$$\text{HYPATIA}(\text{MODE}) := \\ [\$\text{Hypatia} \mid \$\text{Hp}] \text{DIALECT} [[\$\text{modo} \mid \$\text{md}] \text{MODE}']? \$: \\ \text{DOCUMENT}(\text{DIALECT}, \text{MODE}') \$.$$

(es. **\$Hypatia Sunya \$modo Unica \$: ... \$.**);

- **\$Lingua** (o **\$Ln**): *Definizione di frammento*. Apre la definizione di un frammento di linguaggio (ovvero un sistema formale o un suo sottosistema) assegnandogli un nome. Sintassi:

$$\begin{aligned} LINGUA(DIALECT, MODE) := \\ [\$Lingua \mid \$Ln \mid NAME \$: \\ LANGUAGE(DIALECT, MODE) \$. \end{aligned}$$

(es. **\$Lingua** MIU **\$:** ... **\$.**);

- **\$Language**: Sequenza possibilmente vuota di costrutti Hypatia, Lingua o Construct. Sintassi:

$$\begin{aligned} LANGUAGE(DIALECT, MODE) := \\ [HYPATIA(MODE) \mid LINGUA(DIALECT, MODE) \mid \\ CONSTRUCT(DIALECT, MODE)]^* \end{aligned}$$

Come in Metamath, ogni costrutto è tipicamente introdotto da una parola chiave preceduta dal simbolo del dollaro (es. **\$Hypatia**) e viene chiuso dal token terminatore **\$.**. Una caratteristica tipica del linguaggio Hypatia è la **doppia nomenclatura**: ogni costrutto possiede sia un nome esteso e descrittivo in latino (per favorire la leggibilità e la chiarezza semantica), sia una forma canonica standardizzata composta esattamente da due lettere (per favorire la concisione durante il *parsing*). Le due forme sono semanticamente equivalenti (es. **\$Principium** equivale a **\$Pn**).

Token definiti in Hypatia

Questi sono alcuni dei token che vengono definiti e utilizzati dal linguaggio Hypatia:

- **\$.** – Chiude il costrutto corrente;
- **\$:** – Apre il corpo di un costrutto nominato;
- **\$-** – Separa una premessa da una conclusione all'interno di uno *Schema*;
- **\${** e **\$}** – Delimitano uno *Schema* annidato, permettendo la definizione di asserzioni composte;
- **\$=** – Introduce il valore di un'assegnazione per i lessemi mutabili (*mutable lexemes*);
- **\$,** – Separa assegnazioni consecutive all'interno di una singola istruzione di assegnazione;
- **\$@** – Qualifica un riferimento nominato (*named reference*) specificandone il percorso attraverso i frammenti di linguaggio che lo includono (risoluzione dei *namespace*);
- **\$ex** – Introduce una sezione dedicata alle evidenze e alle dimostrazioni (*proof section*);
- **\$per** (o **\$pr**) – Collega un passo di dimostrazione a un elemento precedentemente nominato (come un assioma o un teorema);
- **\$cum** (o **\$cm**) – Introduce la lista di assegnazioni utilizzate per istanziare un passo di dimostrazione;
- **\$modo** (o **\$md**) – Introduce una specifica modalità di interpretazione esplicita (es. *Unica, Omnes, Valida, Aliqua*).

Rappresentazione interna e superficiale

Definito il contesto come insieme di entità semanticamente attive fino a quel punto nel testo, contiene gli elementi che possono rendere la scrittura di un file hypatia, interpretabile in più modi come il dialetto, lo scoping, e le dichiarazioni.

In una lingua in dialetto Sunya vengono definite le variabili come:

\$Mutabilia x x_y y\$.

E gli spazi come:

\$Intervalla \sp _ \$.

Il parser non riuscirebbe a dare un'unica interpretazione di un costrutto che utilizza `x_y` in quanto potrebbe essere visto sia come `x y` seprate, sia come unica variabile.

Vengono distinte due rappresentazioni in Hypatia:

- **Superficiale:** Cosa si può vedere, scritto dall'utente, include gli spazi, i commenti e la forme scelte come il nome abbreviato o in latino per i costrutti, le forme di escape ecc.. Il primo strato in cui un documento Hypatia viene letto.
- **Interna:** Rende esplicita cosa la forma superficiale può voler rappresentare all'interno del contesto creato. Necessaria per gestire le ambiguità create dalla forma superficiale.

Ad ogni frase nella rappresentazione superficiale, dato il contesto, corrispondono una o più interpretazioni interne. Al contrario ad ogni rappresentazione interna corrispondono un insieme di rappresentazioni superficiali.

- **Il meta-linguaggio:** il testo situato al di fuori degli inclusori.
Il meta-testo è una sequenza di **SYMBOLS** dichiarati nell'alfabeto del frammento. La rappresentazione superficiale del meta-testo corrisponde al contenuto testuale che, una volta interpretate tutte le forme di alias e di *escaping*, ammette una segmentazione completa in simboli dichiarati. La medesima stringa superficiale potrebbe ammettere molteplici segmentazioni valide.
- **Il linguaggio oggetto:** il testo racchiuso all'interno di una coppia di delimitatori dichiarati nel costrutto **\$Inclusores**.

Il testo oggetto (*Object Text*) è formalizzato dalla seguente struttura:

*OPENING-DELIMITER OBJECT-CONTENT(CLOSING-DELIMITER)
CLOSING-DELIMITER*

I delimitatori di apertura e chiusura devono appartenere alla medesima coppia dichiarata. Essi svolgono una funzione puramente sintattica di confinamento per il *parser* di Hypatia e non appartengono al contenuto semantico rappresentato.

Con **OBJECT-CONTENT** si intende una sequenza (eventualmente vuota) di **SYMBOLS** dichiarati nell'alfabeto dichiarato nel frammento. L'elaborazione del contenuto oggetto rispetta regole specifiche:

- **Spaziature:** Gli spazi bianchi (*whitespace*) propri della sintassi Hypatia vengono acquisiti e interpretati come parte del contenuto oggetto solo se il relativo simbolo è dichiarato nell'alfabeto.

- **Escaping dei delimitatori:** Un'occorrenza diretta del delimitatore di chiusura terminerebbe prematuramente il blocco di testo. Per includere tale simbolo come puro contenuto del linguaggio oggetto, lo si deve far precedere immediatamente da un *backslash* (\). Il *backslash* sopprime la funzione strutturale di chiusura, non viene mantenuto nel contenuto rappresentato, e permette al simbolo chiuso di essere valutato rispetto all'alfabeto.
- **Marcatori di escape generici:** Più in generale, un *backslash* che non viene consumato da un alias, da un doppio backslash o dall'escaping di chiusura, funge da marcatore per sopprimere l'interpretazione contesto-dipendente del carattere immediatamente successivo.
- **Non-determinismo:** Qualora il testo superficiale presenti ambiguità sintattiche (ad esempio, un delimitatore di apertura associato a più coppie, o conflitti tra alias ed *escaping*), il *parser* non scarta le alternative, ma conserva in memoria tutte le interpretazioni compatibili.

La Struttura del Testo nei Frammenti (Fragment Text)

La porzione testuale di un frammento è governata dal costrutto **TEXT**. Un frammento di testo può essere vuoto; in tal caso, viene rappresentato esplicitamente dal terminale speciale del linguaggio *host*: **\$Vacuum** (o nella sua forma contratta **\$Vc**). Questo costrutto denota l'assenza formale di contenuto.

Quando il **TEXT** non è vuoto, esso è composto da porzioni in meta-linguaggio (**META-TEXT**) e porzioni in linguaggio oggetto (**OBJECT-TEXT**), separate da intervalli dichiarati (**INTERVAL**) qualora necessario.

Le regole di derivazione grammaticale che ne definiscono la composizione sono le seguenti:

```
TEXT :=
    VACUUM | META-TEXT TEXT-AFTER-META | OBJECT-TEXT TEXT-AFTER-OBJECT
TEXT-AFTER-META :=
    [ INTERVAL* [ INTERVAL META-TEXT TEXT-AFTER-META
                  | OBJECT-TEXT TEXT-AFTER-OBJECT ] ]?
TEXT-AFTER-OBJECT :=
    [ INTERVAL* [ META-TEXT TEXT-AFTER-META
                  | OBJECT-TEXT TEXT-AFTER-OBJECT ] ]?
```

Da questa formalizzazione emergono regole rigorose per la scomposizione e l'interpretazione del testo:

- **Il ruolo degli Intervalli:** Gli intervalli (**INTERVAL**) fungono da separatori strutturali e si collocano esclusivamente tra componenti testuali adiacenti. Essi non fanno parte del contenuto semantico (né del meta-testo né del testo oggetto), ma definiscono unicamente l'impalcatura sintattica. L'inserimento di almeno un intervallo è obbligatorio tra due blocchi **META-TEXT** consecutivi, mentre è opzionale negli altri casi. Più intervalli consecutivi mantengono il medesimo effetto separatore di uno singolo, senza introdurre blocchi di testo vuoti.
- **Spaziatura e Terminali Host:** Gli spazi bianchi (*whitespace*) che separano un **TEXT** dalla sintassi *host* circostante mantengono il loro ruolo di separatori a livello *host*, a prescindere dal fatto che quello stesso carattere sia stato dichiarato o meno come

intervallo o delimitatore nell'alfabeto. I terminali attivi di Hypatia mantengono la precedenza assoluta, fungendo da argini testuali per il frammento.

- **Precedenza ed Escaping nel Meta-Testo:** Al di fuori di un OBJECT-TEXT, un'occorrenza diretta e non soggetta a *escape* di un carattere riconosciuto come INTERVAL o OPENING-DELIMITER assume un ruolo puramente strutturale e non può in alcun caso contribuire al META-TEXT. Per forzare l'inclusione di tali caratteri come contenuto testuale meta-linguistico, è necessario anteporre un *backslash* (\). Questo marcatore sopprime la funzione strutturale del carattere (senza essere a sua volta incluso nel testo finale), permettendone la valutazione come simbolo puro, a condizione che esso appartenga all'alfabeto dichiarato.
- **Risoluzione delle ambiguità:** Analogamente alla gestione del linguaggio oggetto, qualora una sequenza preceduta da *backslash* ammetta sia un'interpretazione come *alias* di un simbolo, sia come *escape* strutturale di un intervallo o delimitatore, il *parser* conserva entrambe le letture. Più in generale, qualsiasi occorrenza superficiale (come delimitatori sovrapposti) che ammetta molteplici riconoscimenti strutturali compatibili genera una biforcazione dell'albero di analisi, preservando ogni interpretazione valida per le successive fasi di validazione semantica.

Interpretazione del documento

In ogni momento di analisi, due sono le caratteristiche che determinano il modo in cui un documento viene parsato e interpretato:

- **Dialetto:** determina la sintassi disponibile in quel contesto e la relativa semantica.
 - Sunya
 - Sifr
 - Zero
- **Modalità:** determina l'interpretazione di un contenuto.
 - **Unica:** richiede che esista esattamente *una sola* interpretazione ammissibile, e che questa sia formalmente valida. Qualsiasi ambiguità irrisolta porta al fallimento della verifica;
 - **Omnes:** richiede che l'insieme di interpretazioni non sia vuoto e che *tutte* le interpretazioni generate dal parser risultino valide;
 - **Valida:** scarta le interpretazioni non valide e richiede che quelle rimaste non siano 0 e che siano corrette per gli obblighi semantici successivi;
 - **Aliqua:** richiede che *almeno una* delle interpretazioni ammissibili risulti valida (il sistema tollera l'esistenza di alternative errate purché ne esista una corretta).

Commenti

In Hypatia, tutto il testo non presente in dei costrutti viene trattato com *blob*, ovvero insieme di caratteri che vengono ignorati.

Tuttavia se nel blob fosse scritto qualcosa di riconducibile al linguaggio Hypatia questo sarebbe interpretato.

Esistono quindi dei simboli di escape per creare commenti Hypatia, in cui la semantica viene disabilitata:

- `--`: commento a linea singola.
- `/- e -/`: commento multilinea.

Meccanismi di escape e alias

In Hypatia, la corretta interpretazione del documento richiede una chiara distinzione tra i token strutturali del linguaggio (linguaggio host) e i simboli che compongono il testo dei frammenti formali (meta-linguaggio e linguaggio oggetto). Per gestire i caratteri non stampabili e risolvere le ambiguità sintattiche, il sistema definisce regole precise di *alias* e *escape*.

- **Alias per i simboli (Whitespace)**: Per rappresentare all'interno del frammento alcuni simboli invisibili, come gli spazi, che altrimenti verrebbero scartati dal parser del linguaggio host, si utilizzano degli alias predefiniti introdotti dal carattere backslash (`\`).

I principali alias di spaziatura includono: `\sp` (*spatium*, spazio), `\th` (*tabulatio horizontalis*, tabulazione), `\ln` (*linea nova*, a capo), e `\nh` (*nihil*, nessun carattere).

Sintassi:

`\[/ALIAS]`

(es. `$Alphabetum M\spI $.` dichiara un singolo simbolo atomico "M I" che contiene uno spazio al suo interno).

- **Escape di livello Host (Doppio Dollaro)**: I terminali di Hypatia (come `$.`, `$ex`, `$cum`) sono attivi a seconda del contesto. Se si desidera includere la sequenza di caratteri esatta di uno di questi terminali all'interno del testo del frammento senza che venga eseguito come comando, è necessario anteporre un ulteriore simbolo del dollaro (`$`) per neutralizzarne il significato strutturale.

Sintassi:

`$$[TERMINALE-ATTIVO]`

(es. `$$.` rappresenta la stringa letterale "\$.", impedendo al parser di interpretarla come chiusura del costrutto; `$$ex` rappresenta la stringa letterale "\$ex").

- **Escape del Backslash (Testo letterale)**: Se si desidera scrivere una sequenza di testo che coincida casualmente con un alias riconosciuto, il backslash iniziale deve essere neutralizzato raddoppiandolo (`\\`), che viene interpretato come un singolo backslash letterale.

Sintassi:

`\\[/TESTO]`

(es. `\\sp` rappresenta testualmente i tre caratteri "`\sp`" e non viene convertito in uno spazio; `\\sp` rappresenta un backslash letterale seguito da uno spazio reale).

- **Escape nei Commenti e nell'Object Text:** Il backslash viene impiegato anche come soppressore di ruolo strutturale all'interno dei testi oggetto o nei testi informali (*blob*).
 - *Nei commenti:* Anteporre un backslash ai delimitatori di commento impedisce l'apertura o la chiusura degli stessi (es. \- viene letto come semplice testo e non inizializza un commento a linea singola; lo stesso vale per \/- e \-/).
 - *Nell'Object Text:* Un backslash posizionato immediatamente prima della stringa di chiusura di un *Inclusore* ne annulla l'effetto chiusurale, permettendo di utilizzare liberamente quel carattere all'interno della formula o del linguaggio oggetto.

Di seguito sono elencati i costrutti fondamentali del dialetto base **Sunya**, suddivisi per ruolo architetturale all'interno del sistema.

Infrastruttura Alfabetica

- **\$Alphabetum** (o **\$Ab**): Dichiarare i simboli atomici permessi nel frammento o estende quelli già dichiarati. Come già detto in precedenza nell'esempio l'alfabeto può contenere più interpretazioni del testo.

Sintassi:

$$ALPHABETUM := [\$Alphabetum \mid \$Ab] [SYMBOL]^* \$.$$

(es. `$Alphabetum M I U x y z $.`);

I *SYMBOL* vengono dichiarati separati da uno spazio, implicito alla struttura del costrutto. Simboli dichiarati vicini, diventano uno stesso simbolo. Per dichiarare un simbolo contenente lo spazio, va utilizzato il meccanismo di escape `\sp`.

(es. `$Alphabetum M\spI $.` dichiara il simbolo 'M I')

- **\$Intervalla** (o **\$It**): Dichiarare i simboli (come lo spazio o la tabulazione) che fungono da separatori tra le parole del meta-linguaggio o estende quelli già dichiarati in precedenza. Vengono letti come per l'*ALPHABETUM*.

Sintassi:

$$INTERVALLA := [\$Intervalla \mid \$It] [SYMBOL]^* \$.$$

(es. `$Intervalla \sp \th \tv $.`)

- **\$Inclusores** (o **\$Ic**): Dichiarare o estende l'insieme di coppie di delimitatori utilizzati per incapsulare e separare il linguaggio oggetto dal meta-linguaggio. Sono letti da sinistra verso destra in coppie ordinate, sempre separati dallo spazio.

Sintassi:

$$INCLUSORES := [\$Inclusores \mid \$Ic] [SYMBOL SYMBOL]^* \$.$$

(es. `$Inclusores " " $.`);

Infrastruttura Lessicale

- **\$Lexicon** (o **\$Lx**): Dichiarare o estende i lessemi immutabili (le costanti logiche e strutturali del frammento) separati da spazi.

Sintassi:

$$LEXICON := [\$Lexicon \mid \$Lx] [META-TEXT]^* \$.$$

(es. $\$Lexicon \ M \ I \ U \ \$.$)

- **\$Mutabilia** (o **\$Mt**): Dichiarare o estendere i lessemi mutabili, ovvero le variabili che fungeranno da *placeholder* e che potranno essere istanziate durante le dimostrazioni. Sintassi:

$$MUTABILIA := [\$Mutabilia \mid \$Mt] [META-TEXT]^* \$.$$

(es. $\$Mutabilia \ x \ y \ z \ \$.$)

Strutture Espressive: Frasi, Schemi e Assegnazioni

- **Phrase (Frase)**: Rappresenta un tipo di sintassi dipendente dal contesto che denota un particolare sottoinsieme del **TEXT**. Nelle *Phrase*, le componenti in meta-linguaggio (*META-TEXT*) devono ammettere una scomposizione lessicale completa in lessemi immutabili (definite nel *Lexicon*) o mutabili (definite nelle *Mutabilia*). All'interno delle porzioni in linguaggio oggetto (*OBJECT-TEXT*), i lessemi immutabili non vengono riconosciuti, mentre i lessemi mutabili devono essere riconosciuti (ove possibile) come *placeholder*, a meno che il loro riconoscimento non sia esplicitamente soppresso da un simbolo di escape. Un **TEXT** corrisponde ad una *phrase* se è possibile stabilirne questa definizione. **VACUUM** rappresenta la *Phrase* vuota.
- **Assertion (Asserzione)**: Un'asserzione è un'unità logica che può presentarsi in due forme distinte: come singola **PHRASE**, oppure come uno schema logico annidato delimitato esplicitamente dai token $\$\{$ e $\$\}$.
- **Schema**: È il costrutto che cattura il pattern inferenziale di Hypatia ("poiché... allora..."). Esso collega un'asserzione di *premessa* a un'asserzione di *conclusione* tramite il token separatore **\$-** (nella sua forma base: **Premessa \$- Conclusione**).

Interfacce Mutabili: Free, Act e Mut Per gestire le variabili (i lessemi mutabili) all'interno di queste strutture, il motore logico di Hypatia definisce tre insiemi specifici, responsabili di tracciare l'identità e il ruolo di propagazione di ogni variabile:

- **Mutabilia (mut)**: Rappresenta l'insieme totale di tutti i *Mutabilia* presenti in una rappresentazione interna (sia usati nel meta-linguaggio che come *placeholder* nel testo oggetto).
- **Free (free)**: Rappresenta l'insieme delle variabili "libere", ovvero quelle esposte strutturalmente dal costrutto verso il suo confine esterno.
- **Active (act)**: Rappresenta l'interfaccia mutabile "attiva", ovvero l'insieme delle variabili concretamente disponibili per subire un'istanziamento.

Per una rappresentazione interna di un'ASSERTION che porta una rappresentazione interna di *Phrase* allora:

- $\text{mut}(\text{Assertion}) = \text{mut}(\text{Phrase})$
- $\text{free}(\text{Assertion}) = \text{mut}(\text{Phrase})$
- $\text{act}(\text{Assertion}) = \text{mut}(\text{Phrase})$

Per una rappresentazione interna di uno **SCHEMA** **INNESTATO** che porta una rappresentazione interna di *Schema* allora:

- $\text{mut}(\text{Schema innestato}) = \text{mut}(\text{Schema})$
- $\text{free}(\text{Schema innestato}) = \text{free}(\text{Schema})$
- $\text{act}(\text{Schema innestato}) = \text{act}(\text{Schema})$

Quando si valuta uno **SCHEMA** nella forma base $P \$- C$ (dove P è la rappresentazione della premessa e C quella della conclusione), il calcolo delle interfacce diventa:

- $\text{mut}(\text{Schema}) = \text{mut}(P) \cup \text{mut}(C)$. Registra tutte le variabili mutabili presenti nell'intero schema (unione di premessa e conclusione).
- $\text{free}(\text{Schema}) = \text{free}(C) \setminus \text{mut}(P)$. Le variabili libere dello schema sono unicamente quelle esposte dalla sua conclusione che *non* sono state già fornite e vincolate dalla premessa.
- $\text{act}(\text{Schema}) = \text{free}(P) \cup \text{act}(C)$. L'interfaccia attiva per l'istanziamento è formata dall'interfaccia libera della premessa, unita all'attività che si propaga in modo ricorsivo attraverso la conclusione.

In virtù di questa definizione ricorsiva, uno schema annidato può arrivare ad avere un'interfaccia **free** vuota (senza variabili libere esposte verso l'esterno) pur mantenendo un'interfaccia **active** non vuota per le istanziazioni interne.

Validità e Sicurezza (Safety) Il calcolo compositivo di questi insiemi è alla base del concetto di **Validità** formale dei costrutti.

- Una rappresentazione interna di una **PHRASE** è considerata *safe* se e solo se il suo insieme $\text{mut} = \emptyset$. La validità di una rappresentazione interna di una **PHRASE** coincide esattamente con la sua condizione di *safety*. La validità è quindi una condizione applicata alla singola rappresentazione interna, non una restrizione sulle rappresentazioni "grezze" ammesse in generale dalla categoria sintattica. Di conseguenza, sia le rappresentazioni valide che quelle invalide continuano a far parte dell'insieme generale delle rappresentazioni interne, mentre soltanto quelle valide costituiscono l'insieme delle rappresentazioni valide.
- Una rappresentazione interna di uno **SCHEMA** è considerato *safe* se e solo se il suo insieme $\text{free} = \emptyset$. Ciò implica che ogni singola variabile esposta nella conclusione debba essere obbligatoriamente coperta da una variabile presente nella premessa ($\text{free}(C) \subseteq \text{mut}(P)$). Questo meccanismo sintattico previene in modo assoluto la comparsa di variabili ingiustificate o "fantasma" all'esito di un processo di deduzione logica. La validità di una rappresentazione interna di uno **SCHEMA** coincide esattamente con la sua condizione di *safety*. La validità è quindi una condizione applicata alla singola rappresentazione interna, non una restrizione sulle rappresentazioni "grezze" ammesse in generale dalla categoria sintattica.

Assegnazioni (Assignments) Un'espressione di assegnazione mappa uno o più lessemi mutabili al medesimo frammento testuale sostitutivo tramite il token $\$=$. Se sono presenti più assegnazioni in catena, queste vengono separate dal token $\$,$. Sintassi:

$$ASSIGNMENTS := ASSIGNMENT [\$, ASSIGNMENT]^*$$

$$ASSIGNMENT := [MUTABLE-LEXEME]+ \$= TEXT$$

(es. $x \ y \ \$= M \ \$, z \ \$= \$Vacuum$)

Le assegnazioni elencate in una singola espressione vengono applicate dal motore in modo strettamente **simultaneo** sull'intera superficie del costrutto (sia nella premessa che nella conclusione, preservando intatto il separatore $\$-$ ed eventuali altri elementi di contorno). I frammenti testuali sostitutivi rimpiazzano sintatticamente ogni occorrenza del *placeholder* associato. Poiché la sostituzione è concorrente, un valore di assegnazione non può mai essere alterato o influenzato da un'altra assegnazione presente nella medesima istruzione. Al termine della procedura di sostituzione testuale di superficie, il frammento risultante viene sottoposto da zero a una nuova interpretazione strutturale da parte del *parser*.

Infrastruttura Teorica (Afferzioni e Regole)

L'infrastruttura logica di Hypatia si basa sulle **Phrases** (asserzioni isolate) e sugli **Schemas** (strutture inferenziali che legano una o più premesse a una conclusione tramite il *token* separatore $\$-$). I costrutti si classificano in base al proprio contenuto e alla necessità di includere una prova formale.

La dichiarazione di uno di questi elementi introduce un nuovo identificatore all'interno del frammento, associandovi gli attributi richiesti dallo specifico costrutto, quali la modalità di interpretazione, l'espressione di riferimento (sia essa una *Phrase* o uno *Schema*) e l'eventuale dimostrazione.

Di seguito vengono descritte le tipologie di costrutti:

- **\$Principium** (o **\$Pn**): *Dichiarazione di Principio*. Dichiarare una Frase accettata come vera senza necessità di dimostrazione.

Sintassi:

$$PRINCIPIUM := [\$Principium \mid \$Pn] NAME \$: PHRASE \$.$$

(es. $\$Principium \text{ MI-Principle } \$: \text{ MI } \$.$);

- **\$Regula** (o **\$Rg**): *Dichiarazione di Regola*. Dichiarare uno Schema (regola di inferenza) accettato senza prova.

Sintassi:

$$REGULA := [\$Regula \mid \$Rg] NAME \$: SCHEMA \$.$$

(es. $\$Regula \text{ I-Rule } \$: \text{ xI } \$- \text{ xIU } \$.$);

- **\$Productum** (o **\$Pd**): *Dichiarazione di Prodotto*. Dichiarare un teorema strutturato come Frase. A differenza del Principium, richiede obbligatoriamente una o più dimostrazioni esplicite fornite all'interno di un blocco introdotto dal token **\$ex**.

Sintassi:

$$\begin{aligned} PRODUCTUM &:= [\$Productum \mid \$Pd] NAME \$: \\ &\quad PHRASE \$ex [PHRASE-PROOF]+ \$ \\ PHRASE-PROOF &:= [\$Probatione \mid \$Pb] PHRASE-PROOF-SCRIPT \$ \end{aligned}$$

(es. $\$Productum$ Teorema $\$$: MIU $\$ex \dots \$$.);

- **$\$Derivatio$ (o $\$Dr$):** *Dichiarazione di Derivazione.* Dichiarata un teorema strutturato come Schema inferenziale dimostrabile. Anch'esso richiede una o più prove esplicite fornite nel blocco $\$ex$.

Sintassi:

$$\begin{aligned} DERIVATIO &:= [\$Derivatio \mid \$Dr] NAME \$ \\ SCHEMA \ \$ex \ [SCHEMA-PROOF]_+ \$ \\ SCHEMA-PROOF &:= [\$Probatione \mid \$Pb] SCHEMA-PROOF-SCRIPT \$ \end{aligned}$$

(es. $\$Derivatio$ Teorema $\$$: xUIIIy $\$-$ xy $\$ex \dots \$$.)

Per richiamare uno di questi costrutti all'interno di un frammento, la sintassi prevede la seguente notazione:

$$REFERENCE := NAME [\$@ NAME]^*$$

All'interno di un singolo frammento, la dichiarazione associa un nome a un unico elemento. Viceversa, elementi omonimi dichiarati in frammenti separati costituiscono entità logiche distinte. Un costrutto diventa accessibile al termine della sua dichiarazione e rimane logicamente vincolato al frammento di origine. Qualora venga definito all'interno di un blocco $\$Lingua$ innestato, esso diventa visibile al blocco $\$Lingua$ contenitore immediatamente dopo la chiusura del frammento annidato.

Una *Reference* si definisce *unqualified* se è costituito unicamente dal nome del costrutto. In tal caso, la risoluzione del nome avviene valutando esclusivamente le dichiarazioni del frammento corrente; in assenza di corrispondenze, il riferimento genera un errore.

Una *Reference* è invece *qualified* se contiene almeno un *token* $\$@$, seguito dal nome del frammento in cui risiede la dichiarazione del costrutto *target*. Tale espressione risulta valida solo se identifica univocamente un singolo elemento: l'assenza di riscontri o la presenza di più risultati compatibili al riferimento fornito, comportano il fallimento della valutazione.

Infine, una catena di qualificazione completa (*full qualification*) è definita dalla sequenza logica di tutti i frammenti che incapsulano la dichiarazione, ordinati dal livello più interno a quello più esterno.

2.2.3 Dimostrazioni, meccanismi di verifica e funzionamento

In Hypatia, l'infrastruttura delle dimostrazioni è fondamentale per giustificare i costrutti teorici che non sono accettati come assiomi, ovvero i prodotti ($\$Productum$) e le derivazioni ($\$Derivatio$). Ogni dimostrazione è introdotta dal token $\$ex$ (sezione delle evidenze) e racchiusa all'interno del costrutto $\$Probatione$ (o $\$Pb$).

Il linguaggio distingue due tipologie principali di script di dimostrazione:

- **Phrase-proof scripts:** giustificano una singola *Phrase* dichiarata da un $\$Productum$. La prova si considera completata con successo quando l'espressione generata dall'ultimo elemento di prova coincide sintatticamente con la *Phrase* dichiarata.
- **Schema-proof scripts:** giustificano uno *Schema* dichiarato da una $\$Derivatio$. A differenza delle prove sulle frasi, queste mantengono uno **stack delle premesse** (inizialmente vuoto) e un **target residuo** (inizialmente uguale allo schema dichiarato dall'utente).

Proof Objects e Risultati Terminali

A differenza di altri verificatori lineari, in Hypatia l'interpretazione di una prova dipende dal contesto e procede in due fasi distinte:

1. **Costruzione del Proof Object:** Lo script determina un *proof object*, ovvero un albero radicato costruito livello per livello a partire dagli stati di prova correnti. Il passaggio da uno stato all'altro (tramite la funzione Succ^F) è governato dalla modalità attiva (*Unica, Omnes, Valida, Aliqua*), che può generare rami unici, multipli o rami interrotti.
2. **Valutazione dei Terminal Results:** I risultati terminali estratti dai rami completi (dopo aver svuotato l'eventuale stack delle premesse in ordine inverso) vengono confrontati con la rappresentazione interna del corpo della dichiarazione per confermarne la correttezza.

Inoltre, affinché una prova sia considerata conclusa con successo, essa deve rispettare la **non-circularità**: ogni dipendenza diretta da un prodotto o derivazione (attraverso un riferimento) deve essere giustificabile ricorsivamente, senza mai creare cicli infiniti o dipendere dall'elemento stesso che si sta cercando di dimostrare.

Contesto di prova e passi fondamentali nel dialetto Sunya

Una dimostrazione è composta da una sequenza ordinata di passi terminati dal token $\$$., ognuno dei quali genera un *proof item* aggiunto al contesto locale.

Negli *schema proof*, ogni *proof item* stabilito possiede metadati specifici chiamati **Mutabilia Rigide** (indicati con $\text{rig}(i)$ per un item i). La rigidità è una proprietà dell'elemento di prova, non solo della sua rappresentazione interna, per cui item distinti che trasportano la stessa rappresentazione di SCHEMA possono avere insiemi rigidi differenti. Se i trasporta una rappresentazione interna PHRASE r , allora $\text{rig}(i) = \text{mut}(r)$. Se trasporta uno SCHEMA r , allora $\text{free}(r) \subseteq \text{rig}(i) \subseteq \text{mut}(r)$. Di conseguenza, ogni mutabile di una frase stabilita sotto determinate premesse diventa rigido in quel *proof item*, mentre uno schema può conservare mutabilia rigide non libere ereditate dal modo in cui è stato stabilito. I passi di dimostrazione elementari determinano e propagano la rigidità come segue:

- **\$Assumo** (o **\$Am**): Introduce nel contesto di prova una frase o uno schema giustificati tramite il riferimento a un elemento nominato già disponibile (come un **\$Principium**, una **\$Regula** o un **\$Productum** validato).

Questo passo assegna a ogni *proof item* che stabilisce un insieme rigido vuoto ($\text{rig}(i) = \emptyset$). Ciò è coerente poiché le frasi e gli schemi nominati sono intrinsecamente *safe*.

Sintassi per le frasi:

$$\begin{aligned} \text{ASSUMO-PHRASE} &:= [\$Assumo \mid \$Am] \text{ PHRASE} \\ &\quad [\$per \mid \$pr] \\ &\quad [\$Principium \mid \$Pn \mid \$Productum \mid \$Pd] \text{ REFERENCE } \$ \end{aligned}$$

Sintassi per gli schemi:

$$\begin{aligned} \text{ASSUMO-SCHEMA} &:= [\$Assumo \mid \$Am] \text{ SCHEMA} \\ &\quad [\$per \mid \$pr] \\ &\quad [\$Regulam \mid \$Rg \mid \$Derivationem \mid \$Dr] \text{ REFERENCE } \$ \end{aligned}$$

(es. $\$Assumo$ MI $\$per$ $\$Principium$ MI-Principle $\$.$);

- **$\$Obtineo$** (o **$\$Ot$**): Istanza lo schema trasportato dall'elemento di prova precedente i_A , tramite assegnazioni introdotte da **$\$cum$** . Negli *schema proof*, la condizione sul dominio delle assegnazioni viene raffinata: le sostituzioni possono avvenire solo se $\text{dom}(\text{asg}(q)) \subseteq \text{act}(r) \setminus \text{rig}(i_A)$. Se un'istanziatura ammissibile q produce una nuova rappresentazione di **SCHEMA** r' , l'item stabilito avrà un insieme rigido $\text{rig}(i) = \text{free}(r') \cup (\text{rig}(i_A) \cap \text{mut}(r')) \cup \text{dep}_q(T, r')$. Il primo componente rende rigida l'interfaccia libera del risultato, il secondo preserva la rigidità ereditata dallo schema istanziato (ogniqualevolta i relativi mutabilia restino presenti), e il terzo traccia le nuove dipendenze dallo stack delle premesse T introdotte dai valori di assegnazione.

Sintassi:

$$\begin{aligned} \text{OBTINEO} &:= [\$Obtineo \mid \$Ot] \text{SCHEMA} \\ &\quad [\$cum \mid \$cm] \text{ASSIGNMENTS } \$ \end{aligned}$$

(es. $\$Obtineo$ MI $\$-$ MIU $\$cum$ x $\$=$ M $\$.$);

- **$\$Consequor$** (o **$\$Cn$**): Rappresenta l'analogo formale del *modus ponens* all'interno degli script di prova. Deriva una nuova frase o un nuovo schema basandosi sui due elementi di prova immediatamente precedenti. Siano i_A il penultimo item e i_S l'ultimo, e sia r_B la rappresentazione della conclusione dello **SCHEMA** selezionato da i_S . Se r_B è una **PHRASE**, il nuovo item stabilito avrà $\text{rig}(i) = \text{mut}(r_B)$. Se r_B è uno **SCHEMA**, la sua rigidità sarà $\text{rig}(i) = \text{free}(r_B) \cup ((\text{rig}(i_A) \cup \text{rig}(i_S)) \cap \text{mut}(r_B))$. Dunque, la rigidità già presente in entrambi gli item consumati si propaga esattamente quando il mutabile corrispondente occorre nello schema stabilito.

Sintassi:

$$\text{CONSEQUOR} := [\$Consequor \mid \$Cn] [\text{PHRASE} \mid \text{SCHEMA}] \$$$

(es. $\$Consequor$ MIU $\$.$);

- **$\$Reitero$** (o **$\$Rt$**): Ripropone una frase o uno schema già stabiliti nel contesto di prova corrente. Questo passo preserva esattamente l'insieme rigido trasportato dal *proof item* che viene ristabilito. La semantica generale di **$\$Reitero$** tiene già conto del fatto che *proof item* differenti possano corrispondere alla stessa rappresentazione interna ma avere metadati diversi, determinando stati successivi differenti a meno che gli stati risultanti non coincidano.

Sintassi:

$$\text{REITERO} := [\$Reitero \mid \$Rt] [\text{PHRASE} \mid \text{SCHEMA}] \$$$

(es. $\$Reitero$ MI $\$.$);

- **$\$Praemitto$** (o **$\$Pm$**): Passo utilizzato esclusivamente negli *schema-proof*, espone una premessa aggiungendola allo stack. Quando stabilisce una rappresentazione **PHRASE** r , assegna al nuovo item l'insieme rigido $\text{rig}(i) = \text{mut}(r)$. Quando stabilisce una rappresentazione **SCHEMA** r , assegna al nuovo item l'insieme rigido $\text{rig}(i) = \text{free}(r)$.

Sintassi:

$$\text{PRAEMITTO} := [\$Praemitto \mid \$Pm] [\text{Mutable-PHRASE} \mid \text{SCHEMA}] \$$$

(es. $\$Praemitto$ xUIIIy $\$.$);

Dipendenze delle assegnazioni ($\text{dep}_q(T, r')$) Quando il passo $\$0\text{btineo}$ applica una sostituzione (che indichiamo con q) per generare un nuovo SCHEMA r' , il motore logico ha bisogno di tracciare l'origine del testo appena elaborato. L'insieme $\text{dep}_q(T, r')$ serve proprio a catturare questa "memoria temporanea" sulla provenienza dei caratteri.

Una variabile (o *lessema mutabile*) viene inserita in questo insieme delle dipendenze se e solo se soddisfa contemporaneamente due condizioni:

1. È già presente nello stack delle premesse attuali (ovvero appartiene all'insieme $\text{mut}(T)$).
2. Almeno una sua occorrenza all'interno del nuovo schema r' esiste *proprio perché* è stata immessa materialmente (in tutto o in parte) dal valore dell'assegnazione q .

Da questa definizione deriva logicamente che l'insieme delle dipendenze conterrà sempre e solo variabili condivise sia dalle premesse che dal nuovo schema (formalmente, $\text{dep}_q(T, r') \subseteq \text{mut}(T) \cap \text{mut}(r')$).

La potenziale fusione del testo: se la stringa appena inserita si attacca ai caratteri adiacenti formando una variabile diversa da quella di partenza, il sistema registrerà la dipendenza basandosi esclusivamente sulla variabile finale riconosciuta nel nuovo contesto. Il significato isolato che il valore di assegnazione aveva prima dell'inserimento viene ignorato. Infine, questa informazione transitoria sull'origine testuale viene cancellata in modo definitivo non appena il sistema termina di calcolare l'insieme rigido del nuovo *proof item*.

Meccanismi di verifica: Purezza e Pseudo-prove

L'architettura di verifica di Hypatia implementa il tracciamento della validità logica. Una dimostrazione, una volta completata, può assumere due stati definitivi: **genuina** (*genuine*) o **pseudo-prova** (*pseudo-proof*).

Per gestire ipotesi, esplorazioni logiche o teoremi non ancora verificati, Hypatia mette a disposizione il costrutto **\$Divino**:

- **\$Divino** (o **\$Dv**): Permette di congetturare una frase o uno schema senza che questi debbano derivare formalmente dagli elementi precedenti nel contesto. Un passo **\$Divino** che stabilisce una rappresentazione interna PHRASE r assegna al nuovo *proof item* un insieme rigido $\text{rig}(i) = \text{mut}(r)$. Quando stabilisce una rappresentazione interna SCHEMA r , l'insieme rigido assegnato è $\text{free}(r)$.

Sintassi:

$$\text{DIVINO} := [\$Divino \mid \$Dv] [PHRASE \mid SCHEMA] \$.$$

(es. **\$Divino** MI **\$-** MIU **\$.**);

La presenza di un passo **\$Divino** (o il riferimento a un altro costrutto precedentemente basato su pseudo-prove) "macchia" in modo irreversibile la prova, declassandola a pseudo-prova. Al termine dell'analisi di un documento, quest'ultimo è considerato **puro** esclusivamente se nessun prodotto o derivazione al suo interno risulta "macchiato".

2.3 Confronto tra Metamath e Hypatia

Come analizzato nei paragrafi precedenti, pur condividendo la stessa radice filosofica basata sulla grammatica (*grammar-first*) e sul ruolo del sistema come *Proof Verifier* puro, Metamath e Hypatia presentano sostanziali differenze architetturali.

Al fine di facilitare la comprensione del processo di traduzione operato dal modulo Metis, la Tabella 2.1 propone una mappatura riassuntiva delle similitudini e differenze tra i due sistemi formali.

Tabella 2.1: Corrispondenze e differenze chiave tra Metamath e Hypatia (Sunya).

Caratteristica	Metamath	Hypatia
Archiviazione e Librerie	Orientato all'archiviazione compatta e presenza di database come set.mm.	Non ancora implementato un meccanismo di archiviazione o un sistema di librerie.
Meccanismo di toolkit	Non prevede un insieme di toolkit per facilitare il passaggio a Metamath.	Prevede più toolkits.
Distinzione sintattica	Non prevede un meccanismo di distinzione sintattica.	Distingue tra Linguaggio host, meta e oggetto, tramite frasi quoted o unquoted.
Approccio alle variabili	Metavariabili	Segnaposto sostituibili

Problema

3.1 Il traduttore

3.1.1 Come affrontare la traduzione

Nel capitolo precedente abbiamo analizzato le differenze principali tra i due linguaggi. Per implementare il traduttore è quindi necessario stabilire una mappatura che metta in relazione sia i costrutti sia la semantica che li caratterizza.

La Tabella 3.1 illustra il confronto diretto tra i costrutti di Metamath e i loro equivalenti in Hypatia:

Tabella 3.1: Confronto tra i costrutti Metamath e Hypatia.

Caratteristica / Costrutto	Metamath	Hypatia (Dialecto Sunya)
Definizione dell'alfabeto	Non esiste.	\$Alphabetum.
Definizione dei separatori	Impliciti (non modificabili, gli spazi bianchi fungono da delimitatori per i token).	Espliciti (\$Intervalla).
Definizione degli inclusori	Non esiste.	\$Inclusores.
Costanti immutabili	\$c	\$Lexicon (o \$Lx).
Variabili	\$v	\$Mutabilia (o \$Mt).
Ipotesi Logica	\$e (<i>essential hypothesis</i>)	Tradotta come parte della premessa in uno <i>Schema</i> (corrispondente a una <i>mandatory hypothesis</i>) o introdotta in una prova tramite \$Praemitto.
Ipotesi di tipo	\$f (<i>floating hypothesis</i>)	Come sopra: parte della premessa in uno <i>Schema</i> o introdotta in prova tramite \$Praemitto.

Caratteristica / Costrutto	Metamath	Hypatia (Dialecto Sunya)
Assiomi (Senza dimostrazione)	\$a	\$Principium (per assiomi senza ipotesi) e \$Regula (per assiomi con ipotesi).
Teoremi (Richiedono dimostrazione)	\$p separato dalla prova tramite \$=	\$Productum (per teoremi senza ipotesi) e \$Derivatio (per teoremi con ipotesi).
Gestione della dimostrazione	Stringa di <i>label</i> , validazione basata sullo <i>Stack</i> (RPN).	Sequenza di passi logici tramite \$Obtineo, \$Consequor o \$Reitero; verifica strutturale basata sul <i>Modus Ponens</i> .

Essendo il dialetto Sunya ispirato a Metamath, gran parte dei costrutti presenta un'evidente corrispondenza logica. Tuttavia, Hypatia consente di specificare in modo più dettagliato la struttura sintattica del linguaggio: richiede infatti la dichiarazione dell'alfabeto, degli intervalli e degli inclusori, distinguendo il linguaggio *host*, il metalinguaggio e il linguaggio oggetto. Tali caratteristiche, pur imponendo maggiori responsabilità di specifica, offrono gli strumenti necessari per colmare le differenze rispetto al linguaggio sorgente.

Una conversione puramente sintattica non è pertanto sufficiente. Diversi meccanismi tipici di Metamath richiedono un'elaborazione algoritmica dedicata per essere convertiti in Hypatia.

Alphabetum, Lexicon e Mutabilia

In Metamath le costanti (\$c) e le variabili (\$v) definiscono l'intero insieme dei simboli utilizzabili nel database. In Hypatia, la dichiarazione di tali elementi si divide in tre componenti:

- **\$Alphabetum:** Definisce l'insieme dei caratteri o delle sottostringhe che compongono i simboli dichiarati nel lessico e nelle variabili.
- **\$Lexicon:** Corrisponde alla dichiarazione Metamath \$c e contiene i simboli immutabili. Ogni simbolo qui definito deve essere composto esclusivamente da elementi presenti nell'alfabeto.
- **\$Mutabilia:** Corrisponde alla dichiarazione \$v e definisce i simboli associati alle variabili. Anche in questo caso, i simboli devono derivare dagli elementi dell'alfabeto.

Questa specifica impone un'elaborazione in fase di traduzione: il modulo deve scomporre in singoli caratteri le variabili e le costanti, inserendo i risultati all'interno dell'alfabeto.

Gestione delle variabili e ipotesi Mandatory

In Hypatia non è presente il concetto nativo di ipotesi *mandatory*. In Metamath, una variabile è considerata *mandatory* se compare direttamente nell'enunciato dell'asserzione o in una qualsiasi ipotesi logica (**\$e**) attiva nello *scope*. Tutte le ipotesi logiche attive nello *scope* sono implicitamente *mandatory*, mentre le ipotesi di tipo (**\$f**) acquisiscono tale stato solo se la variabile a cui assegnano il tipo risulta a sua volta *mandatory*. Questa definizione si riflette nella traduzione in Hypatia, influenzando la distinzione strutturale di regole e teoremi.

Regole e Teoremi

Metamath non distingue formalmente la tipologia delle dimostrazioni, raggruppandole sotto il costrutto generico di *proof*. Hypatia richiede invece una distinzione esplicita tra *Derivatio* e *Productum*. Un principio analogo si applica alle regole senza dimostrazione: Metamath le definisce unitariamente come assiomi (**\$a**), mentre Hypatia le divide in *Regula* e *Principium*. La classificazione dipende dalla presenza di ipotesi *mandatory* associate all'elemento. Poiché Hypatia non permette di definire le ipotesi come costrutti separati, queste vengono integrate nello *Schema*, assumendo la forma strutturale `ipotesi1 $- {ipotesi2 $- {... $- tesi}}`. Ne consegue che una *Regula* o una *Derivatio* implementano uno *Schema* (e includono pertanto delle premesse), mentre un *Principium* o un *Productum* implementano una *Phrase* (senza premesse).

Dimostrazioni

In Metamath una dimostrazione è costituita da una sequenza di etichette (*label*) che operano su uno *stack* RPN. In Hypatia, la prova si articola in passi logici distinti, i quali vengono generati dal traduttore per ottenere il medesimo risultato semantico:

- **\$Praemitto**: Introduce le espressioni associate alle ipotesi del teorema, le quali figurano già nello *Schema* della tesi.
- **\$Assumo**: Dichiarare uno *Schema* o una *Phrase* a partire da una regola o un teorema esterno. Viene invocato quando nella dimostrazione Metamath si incontrano *label* di asserzioni.
- **\$Obtineo**: Esegue le sostituzioni logiche. È impiegato per emulare le operazioni implicite di sostituzione applicate da Metamath agli elementi dello *stack* RPN.
- **\$Reitero**: Consente di richiamare teoremi o regole precedentemente introdotti nella dimostrazione, qualora non risultino più attivi nello *scope* corrente per la valutazione del costrutto successivo.
- **\$Consequor**: Aggiunge al contesto della dimostrazione la *Phrase* o lo *Schema* risultante dall'applicazione dei passi precedenti. Viene utilizzato per derivare la conseguenza ottenuta da uno *Schema* applicandolo alla premessa immediatamente precedente.
- **\$Divino**: Introduce uno *Schema* o una *Phrase* assunti per veri senza convalida esplicita, declassando la *proof* a *pseudo-proof*. Risulta funzionale quando le *label* sono mancanti o errate nel file originale, permettendo di proseguire l'elaborazione.

Ulteriori elementi gestiti dal traduttore per assicurare una corretta elaborazione:

- **Commenti:** I commenti originali vengono preservati per consentirne la lettura nel documento generato. Offrendo Hypatia differenti approcci per l'isolamento del testo non formale, la modalità di traduzione deve essere posta come parametro di configurazione per l'utente.
- **Dummy Variables:** In Metamath, le *dummy variables* fungono da variabili d'appoggio all'interno della dimostrazione senza comparire nell'enunciato del teorema. Il traduttore deve mantenerne le ipotesi al fine della corretta dimostrazione, e includerle all'interno della dimostrazione in cui vengono utilizzate. Poiché in Hypatia la loro introduzione richiede l'utilizzo dei relativi *\$Praemitto* (e relativo *Schema* di tesi) per le ipotesi, esse vengono trattate come le altre ipotesi *mandatory* dei teoremi che ne fanno uso.
- **Variabili disgiunte (*Disjoint*):** Il vincolo di disgiunzione è assente in Hypatia e necessita di un controllo specifico. Tale implementazione dovrebbe operare analizzando la dichiarazione come linguaggio oggetto e validando restrizioni sulle sottostringhe durante i processi di sostituzione.
- **Inclusione di file:** Mancando un meccanismo nativo in Hypatia per l'inclusione di file esterni, il traduttore deve applicare una logica di espansione delle inclusioni Metamath. I file richiamati devono essere tradotti in *Lingue* innestate; le relative variabili devono essere sollevate al livello gerarchico superiore per garantirne la corretta visibilità.
- **Reference:** In caso di *Lingue* innestate il traduttore deve implementare il meccanismo di *Reference* per richiamare una regola o teorema nella *Lingua* esterna, quindi in caso di file inclusi i teoremi dei file interni vanno memorizzati insieme al nome del file dove sono stati dichiarati (che dà il nome alla *Lingua*).
- **Inclusioni non autorizzate:** Qualora l'utente ometta il *flag -i* (che autorizza l'inclusione esplicita) o indichi una profondità insufficiente, il processo di elaborazione non viene interrotto. La traduzione prosegue cercando di bypassare la validazione formale del file mancante, dove è possibile. In caso di sole label mancanti assume per corretta la prova associata utilizzando il *Divino* e marchiando la *proof* come *pseudo-proof*. Se invece non sono presenti variabili o costanti utilizzate dalla prova del file inclusore (come nel caso di *demo0-include.mm*) la traduzione risulta impossibile, in quanto i simboli non risultano interpretabili.
- **Prove compresse:** Il dialetto Sunya non supporta la compressione delle dimostrazioni. Il traduttore dovrà integrare pertanto un algoritmo di decompressione che espanda la stringa codificata al formato testuale completo, procedendo poi alla traduzione standard.
- **Scope:** I blocchi di *scope*, utilizzati in Metamath per limitare la visibilità di ipotesi e variabili, non hanno un corrispettivo diretto in Hypatia. La traduzione ne dovrà emulare il comportamento a livello logico: Hypatia incapsula le premesse necessarie all'interno dello *Schema* generato per ogni singola regola o teorema. Le variabili, agendo come puri *placeholder* testuali e non conservando attributi di stato tra gli utilizzi, non richiedono invece una dichiarazione isolata.

3.1.2 Obiettivi e validazione del risultato

L'obiettivo principale di questo componente è definire una soluzione algoritmica in grado di traslare fedelmente la logica di Metamath nel contesto di Hypatia.

L'output generato deve rispettare la grammatica del dialetto Sunya ed essere dimostrabile dal verificatore Aristarchus, preservando l'esatta semantica logica del database di partenza. Il codice finale deve inoltre mantenere un'elevata leggibilità: le scelte di traduzione non devono stravolgere il contesto, favorendo la lettura per l'utente. Per lo stesso motivo, il sistema deve rendere visibile la catena di eventuali errori riscontrati nei file, motivando così l'inserimento delle correzioni.

A livello strutturale, ogni documento generato inizierà con il costrutto `$Lingua` accompagnato dal percorso del file tradotto (comportamento replicato anche per i file inclusi). Le dimostrazioni tradotte dovranno risultare formalmente consistenti: il traduttore avrà il compito di esplicitare ogni singolo passo logico, richiamare correttamente le asserzioni e le ipotesi precedentemente assunte (tramite `$Assumo`), e istanziarle al momento del bisogno. Il codice finale, infine, dovrà garantire il rispetto del principio di *safety* per tutti gli schemi generati.

3.1.3 Esempi di dimostrazioni Metamath e Hypatia a confronto

Di seguito verrà illustrato un esempio di traduzione, prendendo come riferimento la codifica del sistema formale MIU di Douglas Hofstadter[28].

Nel codice 3.1 è riportata la struttura originale estratta dal file `miu.mm`. In questo ambiente, l'alfabeto e i costrutti logici vengono definiti tramite costanti (`$c`), variabili (`$v`), ipotesi di tipo (`$f`), ipotesi logiche (`$e`) ed assiomi (`$a`). La dimostrazione del teorema (`theorem1`) è scritta nella versione estesa, in notazione RPN standard (*Reverse Polish Notation*).

```

1  $( First, we declare the constant symbols of the language. $)
2  $c M I U |- wff $.
3
4  $( Next, we declare some variables. $)
5  $v x y $.
6  wx $f wff x $.
7  wy $f wff y $.
8
9  we $a wff $.
10 wM $a wff x M $.
11 wI $a wff x I $.
12 wU $a wff x U $.
13
14 wxy $a wff x y $.
15
16 $( Assert the axiom. $)
17 ax $a |- M I $.
18
19 $( Assert the rules. $)
20 ${
21   Ia $e |- x I $.
22   I_ $a |- x I U $.
23 $}
24
25 ${
26   IIa $e |- M x $.
27   II $a |- M x x $.
28 $}
29
30 ${
31   IIIa $e |- x I I I y $.
32   III $a |- x U y $.
33 $}
34
35 ${
36   IVa $e |- x U U y $.
37   IV $a |- x y $.
38 $}
39
40 $( Now we prove the theorem MUIIU. $)
41 theorem1 $p |- M U I I U $=
42   we wM wU wI we wI wU we wU wI wU we wM we wI wU we wM wI wI wI we wI wI we
43   wI ax II II I_ III II IV $.

```

Listing 3.1: Frammento del database `miu.mm` originale in Metamath.

Il codice 3.2 mostra invece come potrebbe essere rappresentata questa dimostrazione nel dialetto Sunya di Hypatia. La traduzione è puramente a scopo illustrativo, non corrisponde alla traduzione del teorema raffigurato in Metamath, ma rappresenta invece un modo in cui si potrebbe ottenere quel risultato in una dimostrazione Hypatia. Come previsto dai requisiti del traduttore Metis, l'infrastruttura linguistica viene esplicitata semanticamente attraverso i costrutti `$Alphabetum`, `$Lexicon` e `$Mutabilia`. L'assioma viene tradotto in un `$Principium`, mentre le restrizioni logiche diventano `$Regula`. Infine ogni singolo passo della dimostrazione viene reso esplicito tramite la concatenazione di `$Assumo`, `$Obtineo`, `$Reitero` e `$Consequor`.

```

1  $Hypatia Sunya $:
2
3  $Lingua MIU $:
4
5      -- The alphabet of the MIU formal system consists of five symbols.
6  $Alphabetum M I U x y $.
7
8      -- The lexicon consists of all meaningful atomic words in the language.
9  $Lexicon M I U $.
10
11     -- The metavariables "x" and "y" represent arbitrary sequences.
12 $Mutabilia x y $.
13
14     -- This is the first sentence to which the system assigns a meaning.
15 $Principium MI $: MI $.
16
17     -- The following rules describe how new sentences can be derived.
18 $Regula I   $: xI   $- xIU $.
19 $Regula M   $: Mx   $- Mxx $.
20 $Regula III $: xIIy $- xUy $.
21 $Regula UU  $: xUy  $- xy  $.
22
23     -- The following productum, required for the MUIIU Productum, is presented without
24     -- its proof for brevity.
25 $Productum MUIUUIU $:
26
27     MUIUUIU
28 $ex
29
30     $Probatione
31
32     ...
33
34     $.
35
36     -- We can now derive is the sequence "MUIIU".
37 $Productum MUIIU $:
38
39     MUIIU
40
41 $ex
42
43     $Probatione
44
45     $Assumo      xUy  $- xy      $per $Regulam UU $.

```

```
46      $Obtineo    MUIUUUIU $- MUIIU $cum x $= MUI $, y $= IU $.
47      $Assumo    MUIUUUIU          $per $Productum MUIUUUIU $.
48      $Reitero    MUIUUUIU $- MUIIU $.
49      $Consequor          MUIIU $.
50
51      $.
52      $.
53
54      $.
```

Listing 3.2: Traduzione del sistema MIU nel dialetto Sunya di Hypatia.

Progettazione

Per la progettazione dell'architettura del traduttore è stata adottata la metodologia *Functional Declarative Design (FDD)* descritta nel testo *Functional Design and Architecture: Examples in Haskell* di Alexander Granin [24].

Il processo di sviluppo iterativo *top-down* proposto dal FDD prevede una transizione graduale dalla raccolta dei requisiti (tramite mappe mentali) fino alla stesura formale dell'architettura del software basata su livelli.

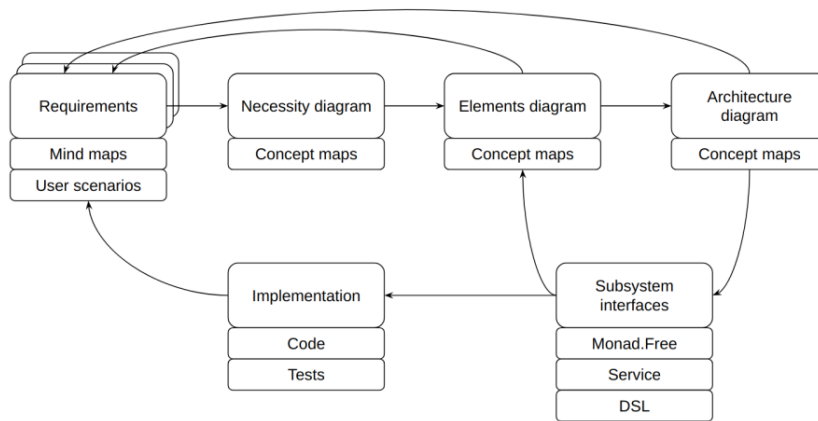


Figura 4.1: Processo iterativo del design architetturale nel Functional Declarative Design.

L'obiettivo della progettazione è definire l'architettura del modulo Metis strutturandola per soddisfare i seguenti requisiti funzionali e non funzionali:

- **Parsing e Conservazione:** eseguire il *parsing* del file sorgente Metamath preservandone i commenti originali e l'alberatura sintattica, gestendo in modo esplicito le inclusioni multiple;
- **Decompressione e Verifica:** validare la consistenza del contesto formale ed eseguire la decompressione delle prove (*Proof*) quando archiviate in formato compresso;
- **Gestione delle Risorse:** ottimizzare il carico computazionale minimizzando le iterazioni sugli elementi già verificati, permettendo l'elaborazione di file di grandi dimensioni limitando il rischio di *stack overflow*;
- **Astrazione e Purezza:** rispettare il paradigma funzionale confinando gli effetti collaterali (I/O) all'esterno della logica di dominio.

4.1 Mappa concettuale dei requisiti

Seguendo la teoria della progettazione di progetti in linguaggi formali fornita nel testo, verranno presentate alcune raffigurazioni grafiche dei requisiti.

In queste mappe concettuali sono raffigurate ad alto livello gli obiettivi e le necessità del progetto, che verranno sviluppate successivamente.

Nella mappa concettuale 4.2 viene raffigurato il primo strato dei requisiti del traduttore. Sarà formato da una serie di componenti: un parser esclusivo per gli include, un parser generale del file, un decompressore, il verificatore e il modulo di traduzione effettivo.

Oltre a questo dovrà rispettare dei requisiti non funzionali che permettono di far funzionare il traduttore anche su file di grandi dimensioni cercando di mantenere alta la leggibilità.

Dovrà restituire come output un elemento doppio contenente sia l'albero di traduzione di tutte le lingue dei file, sia un'albero di errori, che poi verranno gestiti da un pretty printer per stampare a schermo.

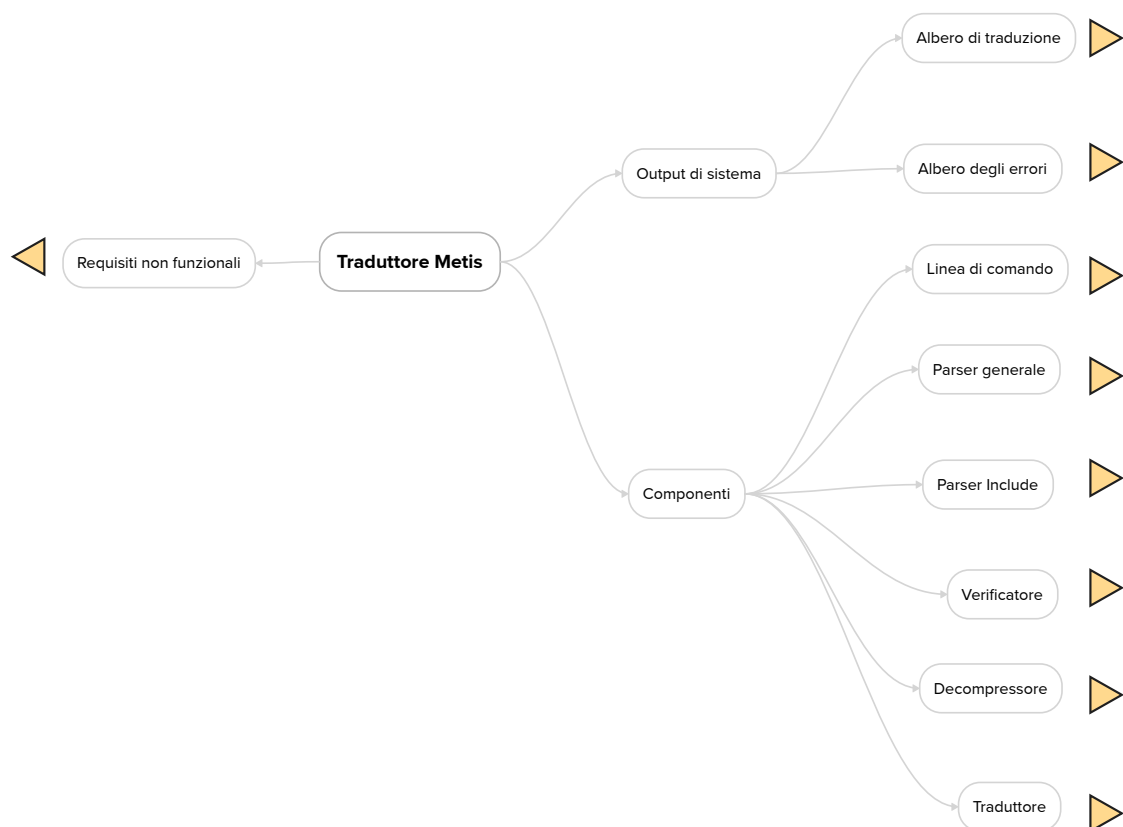


Figura 4.2: Mappa concettuale dei requisiti primo livello.

La mappa 4.3 viene rappresentato il secondo strato di requisiti dell'Output di sistema. Il traduttore dovrà lavorare a prescindere dalla verifica del file di partenza, così da dare sempre in output anche solo una traduzione parziale. Quindi l'albero di traduzione dovrà essere costruito man mano ed essere sempre disponibile.

Se verranno riscontrati degli errori, verranno aggiunti all'albero degli errori, e poi mostrati all'utente prima della traduzione eseguita, così parmettergli di notare gli errori in anticipo, e magari aggiustare subito il file.

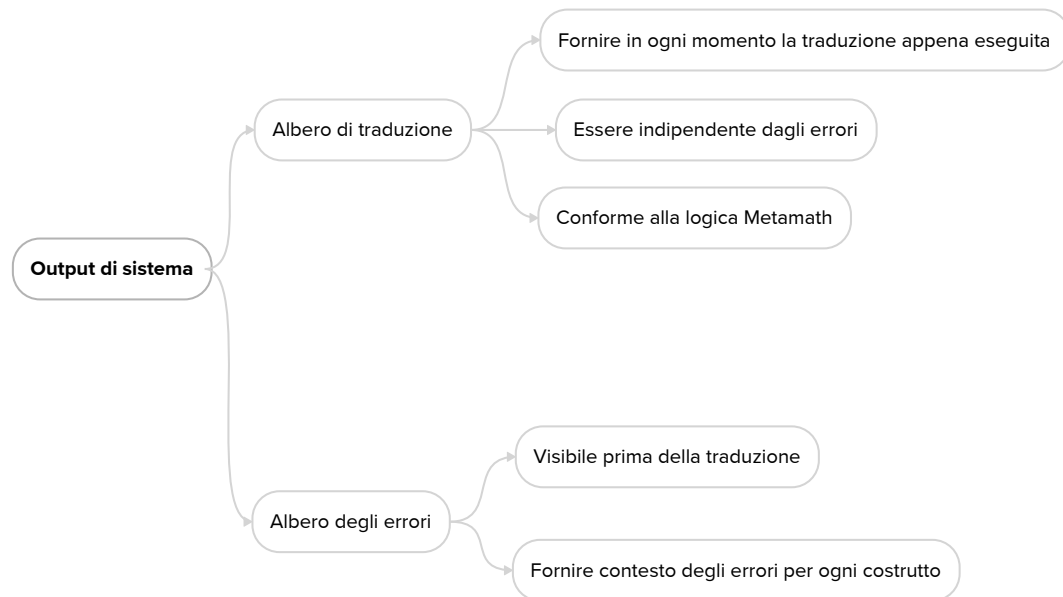


Figura 4.3: Mappa dei requisiti secondo livello Output di sistema.

In 4.4 vengono mostrati invece i requisiti non funzionali che deve rispettare il sistema. Come detto prima, deve mantenere bassa la complessità, cercare di leggere il file quante meno volte possibile, così da non ripetere sempre le stesse operazioni. Inoltre, essendo il codice scritto in Haskell deve mantenere per quanto possibile la purezza delle funzioni, se non per le operazioni di IO che devono essere tenute quanto più in disparte dal resto. Deve risultare semplice per l'utente che decide di utilizzare il programma, e deve risultare facilmente espandibile agli altri linguaggi che verranno tradotti.



Figura 4.4: Mappa dei requisiti secondo livello Requisiti non funzionali.

Nella mappa 4.5 vengono infine descritte le componenti.

- **Parser generale:** Non deve comportarsi come un parser normale, dovrebbe conservare i commenti così da farli comparire nella traduzione e non far perdere contesto all'utente. Deve restituire il file che poi verrà dato in input al verificatore per controllare e tradurre il file Metamath.
- **Parser include:** Deve fungere da micro parser per trovare gli include nel file e trovare una soluzione per il problema del diamante, che verrà affrontato in successivamente.
- **Verificatore:** Il verificatore deve controllare il file Metamath secondo la sua logica, controllare gli scope e la correttezza delle dimostrazioni del file.
- **Decompressore:** Deve implementare l'algoritmo di decompressione di Metamath per passarlo al verificatore, restituirà la stessa struttura di cui sono composte le dimostrazioni estese.
- **Linea di comando:** La linea di comando deve avere varie opzioni per la traduzione da dare all'utente.
- **Traduttore:** Deve implementare la traduzione effettiva del documento Metamath, e costruire man mano l'albero di traduzione. Deve evitare le ambiguità di Hypatia, se possibile e deve riuscire a coprire tutti i casi di costrutti o funzionalità che si possono trovare in un file Metamath.



Figura 4.5: Mappa dei requisiti secondo livello Componenti.

4.2 Diagramma delle necessità

Verrà di seguito mostrato un diagramma che illustra la prima visione ad alto livello dei vari componenti del tool Metis e le relative comunicazioni.

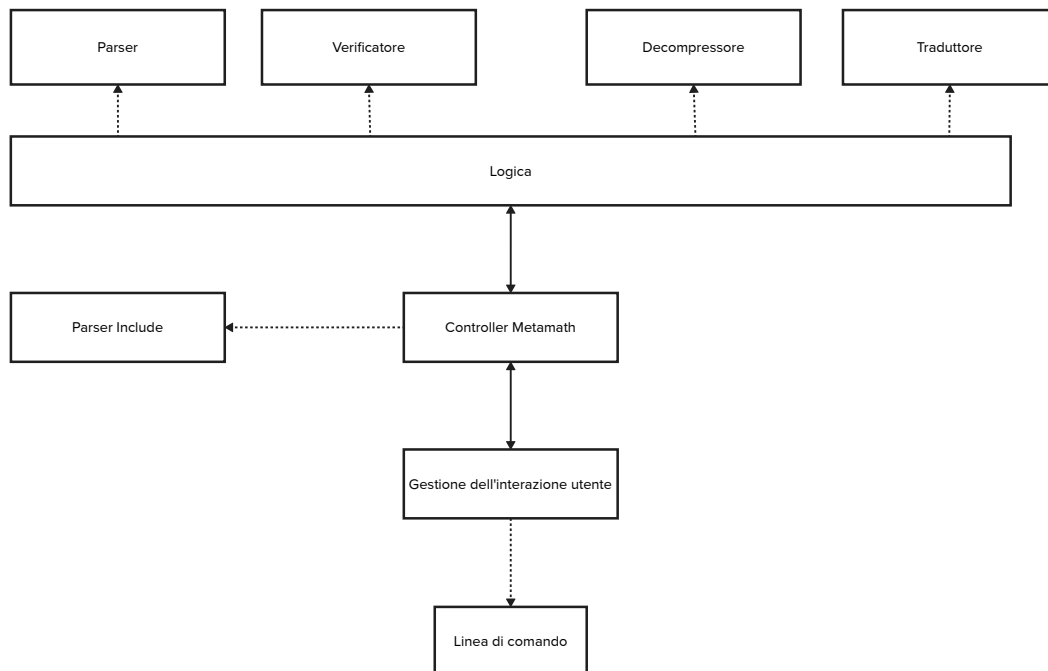


Figura 4.6: Diagramma delle necessità del progetto.

Un diagramma delle necessità (Necessity Diagram) rappresenta le macro-componenti essenziali del programma, deducibili dalle mappe concettuali elaborate durante la fase di *brainstorming* dei requisiti. I dettagli architetturali e le interazioni di livello inferiore verranno affrontati nelle sezioni successive.

In questa fase preliminare, si stabilisce che l'utente interagirà con il sistema attraverso la linea di comando, fornendo in input uno o più file da processare e specificando le relative opzioni di esecuzione. Poiché la lettura dal file system (tramite funzioni come `readFile`) comporta effetti collaterali e appartiene alla monade `IO` (impura), tale operazione deve essere confinata all'interno di un modulo di controllo centrale.

Il "Controller" gestisce l'interazione con l'utente e il caricamento dei file, per poi delegare l'elaborazione dei dati ai moduli composti da funzioni pure. Nello specifico, i parser vengono invocati per primi al fine di generare la struttura dati formale (AST), la quale verrà successivamente fornita in input ai moduli di verifica, decompressione e traduzione, operanti in contesti puramente funzionali.

Sono presenti due tipi di frecce che collegano i vari componenti del programma:

- Freccie continue e bidirezionali: rappresentano la comunicazione diretta tra le componenti.
- Freccie tratteggiate: rappresentano le dipendenze dei moduli.

4.3 Diagramma degli elementi

Il passo successivo alla definizione delle necessità è la stesura del diagramma degli elementi (Elements Diagram). Questo diagramma rappresenta una mappa concettuale semi-strutturata che illustra i moduli fisici del sistema, le librerie esterne, i dati e i concetti di dominio. Le relazioni tra questi nodi sono libere e servono a fissare i risultati del *brainstorming* prima di vincolarli in un'architettura rigida.

Fedelmente alla convenzione del Functional Declarative Design (FDD), ogni elemento all'interno del diagramma è etichettato con due acronimi che ne descrivono il ruolo e la collocazione architetturale:

- **Tipo di elemento:** *Subsystem* (SS), *Library* (L), *Concept* (C), *Data* (D), *Model* (M).
- **Layer (Strato):** *Application Layer* (AL), *Interoperability Layer* (IL), *Domain Model Layer* (DM), *View/Presentation Layer* (VL).

Dall'analisi del codice sorgente del tool Metis, sono stati individuati i seguenti elementi chiave:

- **Elementi di IO (AL e IL):** Il modulo `Translator CLI` (SS, AL) gestisce l'input iniziale, mentre l'`Include Manager` (SS, IL) si occupa di risolvere le inclusioni ricorsive appoggiandosi al *File System* (C).
- **Librerie Esterne (L):** Il parser si avvale della libreria combinatoria `Megaparsec` (L).
- **Dati e Modelli (D, M):** Le strutture che transitano nel sistema sono l'`AST Metamath` (D, DM), l'`AST Hypatia` (D, DM) e la struttura di raccolta anomalie `Error Tree` (D, DM). Il nucleo centrale mantiene inoltre il `Context StateHYP` (M, DM).
- **Sottosistemi di Dominio (SS, DM):** Il cuore logico è orchestrato dal `Processor (MMtoHYP)` che delega compiti specifici al `Metamath Parser`, al `Decompressor`, al `Verifier` e al `Translator`.
- **Presentazione (VL):** Il modulo `Pretty Printer` (SS, VL) formatta gli AST in un output testuale leggibile (D, VL).

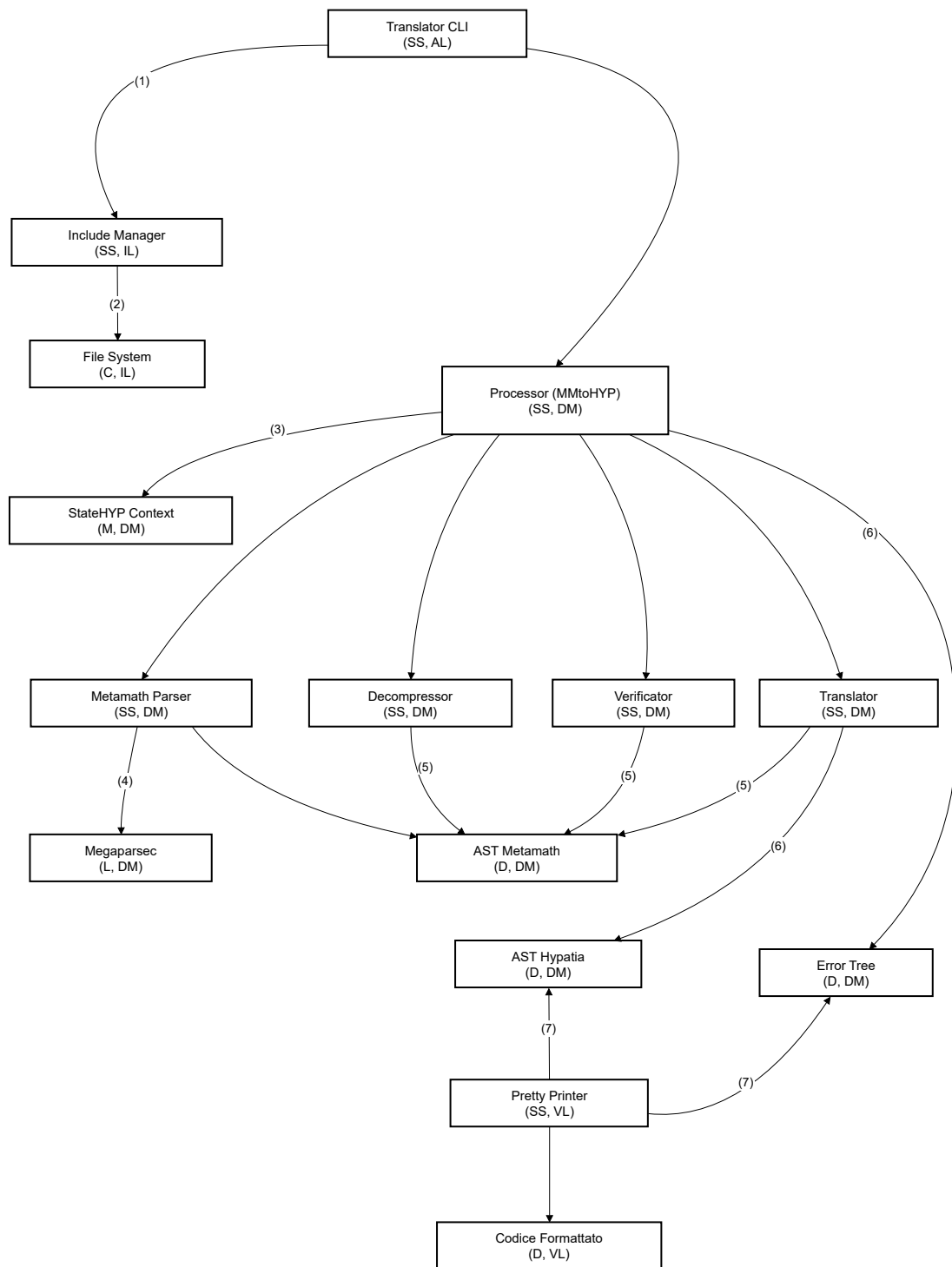


Figura 4.7: Diagramma degli elementi del Traduttore Metis, con annotazioni di tipo e layer secondo il FDD.

4.3.1 Sub-Elements Diagram: Il Processore e la Monade State

Una delle difficoltà principali nell'adozione del paradigma funzionale è la gestione del contesto mutabile. Nel *Functional Declarative Design*, tale problematica viene risolta incapsulando i dati all'interno della Monade **State**, che simula l'alterazione delle variabili passando lo stato come argomento implicito tra le azioni monadiche in background, preservando la purezza della computazione.

Il modulo `MMtoHYP.hs` rappresenta il nucleo orchestratore del traduttore Metis. Al fine di evitare l'utilizzo di variabili globali o effetti collaterali impuri, l'intera elaborazione dei file Metamath avviene all'interno del contesto monadico `State StateHYP ()`.

Il seguente *Sub-Elements Diagram* espone l'architettura logica del processore centrale, illustrando come i *Concepts* (C) modifichino il *Model* (M) collaborando con i sottosistemi adiacenti:

1. Il **Processor** (SS, DM) delega l'elaborazione alla **State Evaluation Logic** (C, DM) (funzione `processMetamath`). Questo concetto inizializza l'orchestrazione e valuta la monade **State**, estraendo infine il documento prodotto e l'albero degli errori.
2. Il cuore del modello di dominio è lo **StateHYP Context** (M, DM). Esso mantiene immutabilmente lo stato dello stack di Metamath, i file inclusi, i costrutti tradotti e l'albero delle anomalie in corso di generazione.
3. L'iterazione sulle istruzioni è governata dalla **Context Update Logic** (C, DM) (funzione `updateContext`). Questa logica consuma iterativamente i nodi dell'AST Metamath (D, DM).
4. Durante l'aggiornamento, il processore si avvale della **Symbol Validation** (C, DM) per verificare formalmente se le variabili e le costanti (come `$v` e `$c`) siano già state dichiarate nel contesto.
5. Quando si incontra una dichiarazione dimostrabile (come il costrutto **Provable**), la logica di aggiornamento si interfaccia con i sottosistemi esterni: invoca il **Decompressor** (SS, DM) e il **Verifier** (SS, DM) per validare formalmente la correttezza della prova.
6. In seguito alla verifica, delega al **Translator** (SS, DM) la generazione delle strutture finali nell'AST **Hypatia** (D, DM), le quali vengono riaccodate nello stato (`stHypCtx`).
7. Qualsiasi fallimento, dipendenza mancante o errore matematico non genera un'eccezione bloccante, ma viene accumulato sotto forma di **IssueOptions** (D, DM) all'interno dello stato, popolando dinamicamente la struttura dell'**Error Tree** finale.

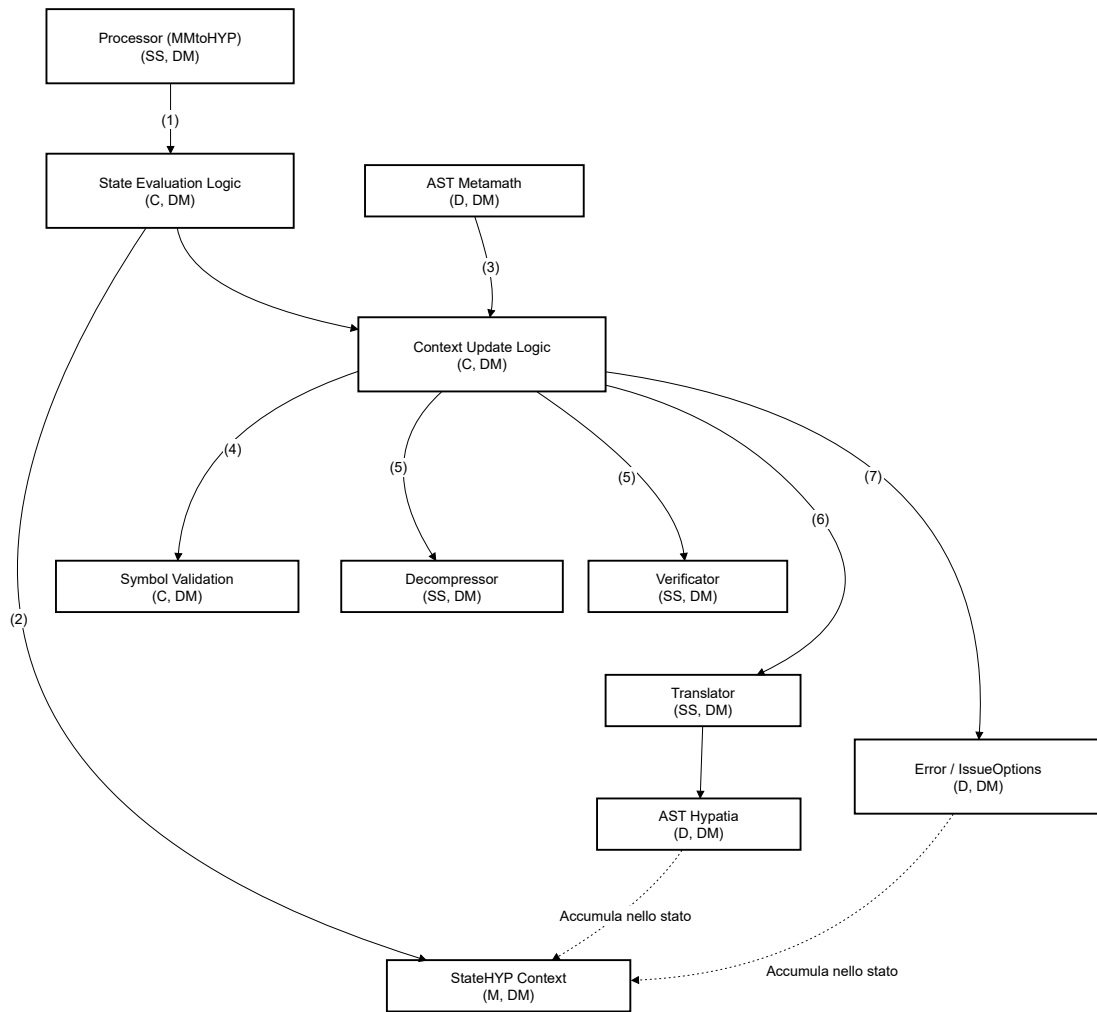


Figura 4.8: Sub-Elements Diagram del Processore (MMtoHYP). L'orchestrazione dei costrutti si realizza propagando un contesto immutabile (StateHYP) senza effetti collaterali impuri.

4.3.2 Sub-Elements Diagram: Il Verificatore

Di seguito vengono descritte le interazioni interne del Verificatore, indicizzate nel diagramma:

1. Il **Verifier** (SS, DM) delega l'elaborazione alla **Verification Orchestration** (C, DM), il concetto logico (implementato dalla funzione **verify**) che governa l'intero ciclo di validazione di un'asserzione Metamath.
2. L'orchestrazione inizializza e altera iterativamente lo **StepState Context** (M, DM), il modello di stato immutabile che traccia lo stack, gli errori e la prova in corso di generazione.
3. L'evoluzione dello stato è demandata alla **Step Application Logic** (C, DM) (funzione **applyStep**), che applica singolarmente ogni passo della dimostrazione.
4. Durante l'applicazione dei passi, il sistema si appoggia alla **Substitution Logic** (C, DM) per far combaciare i token matematici, generando dinamicamente una **Substitution Map** (M, DM).
5. Parallelamente, la **Schema Generation** (C, DM) valuta le ipotesi per costruire formalmente i **Sunya Schemas** (M, DM).
6. Al termine della validazione dei passi, la **Proof AST Generation** (C, DM) si occupa di generare i nodi dell'albero dimostrativo per Hypatia.
7. Per convertire i token matematici grezzi in costrutti testuali corretti, la generazione dell'AST richiede l'ausilio del **Translator** (SS, DM), dal quale importa la **Lexical Translation** (C, DM) (ovvero le funzioni pure **toPhrase** e **toWord**).
8. L'intero processo culmina con la produzione del **VerifyResult** (D, DM), la struttura dati finale restituita al processore centrale.

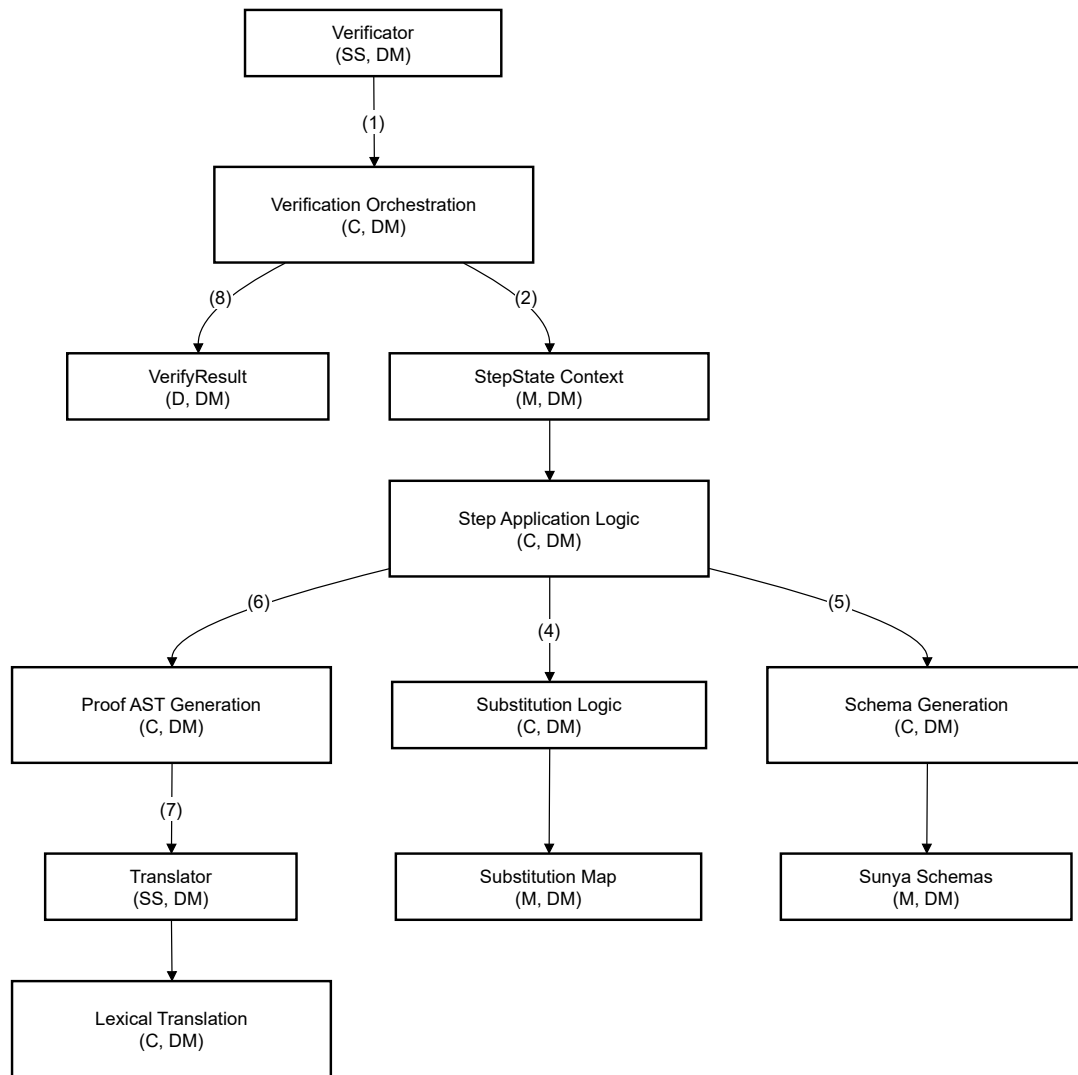


Figura 4.9: Sub-Elements Diagram del Verificatore. Le singole funzioni logiche sono modellate come Concepts (C) che operano su Models (M) e collaborano con il Translator.

4.4 Diagramma dell'Architettura

A conclusione del processo di progettazione, si giunge alla stesura del Diagramma dell'Architettura formale.

L'architettura definitiva del traduttore Metis, mettendo in luce l'infrastruttura completa dei moduli di dominio:

- **Domain Model Layer (Verde):** Costituisce il blocco logico monolitico centrale. Al vertice si situano i quattro sottomoduli di manipolazione e calcolo (*Metamath Parser*, *Decompressor*, *Verifier* e *Translator*). Immediatamente sotto di essi, a fare da substrato per il mantenimento dei dati formali, figurano il *Context Manager* (*Metamath.hs*) per lo stack, le *AST Definitions* per i tipi, e l'*Error Builder* (*Error.hs*). Tutti questi elementi poggiano strutturalmente sul *Logic Control Core* (MMtoHYP), l'orchestratore che sostiene l'intero dominio. Alla base del blocco verde sono dichiarate le astrazioni matematiche che rendono possibile la computazione pura: *Monad.State*, *Monad.Either* e *Monad.Parsec*.
- **Interoperability Layer (Viola):** Rappresentato visivamente come un asse orizzontale di interscambio (*Controller*). Contiene l'*Include Manager* che mette in comunicazione il dominio puro con il mondo esterno, operando sul *File System* tramite *Monad.IO* e catturando le anomalie con *Monad.Either*.
- **Application Layer (Azzurro):** Collocato lateralmente per orchestrare l'ingresso. Il blocco applicativo superiore (*Configuration*) preleva opzioni in forma pura, mentre il blocco inferiore dell'inizializzazione (*trnRun*) delega il flusso al Controller, appoggiando interamente sulla *Monad.IO*.
- **Presentation Layer (Giallo):** Collocato in uscita, preposto alla formattazione. Il *Pretty Printer* trasforma le strutture AST in stringhe testuali formattate operando in un contesto di astrazione *Puro*.

Le relazioni bidirezionali indicano che la comunicazione tra strato impuro e strato puro avviene esclusivamente attraverso le interfacce esposte dal Core, garantendo l'isolamento dei side-effect e il rispetto stringente delle *best practices* funzionali.

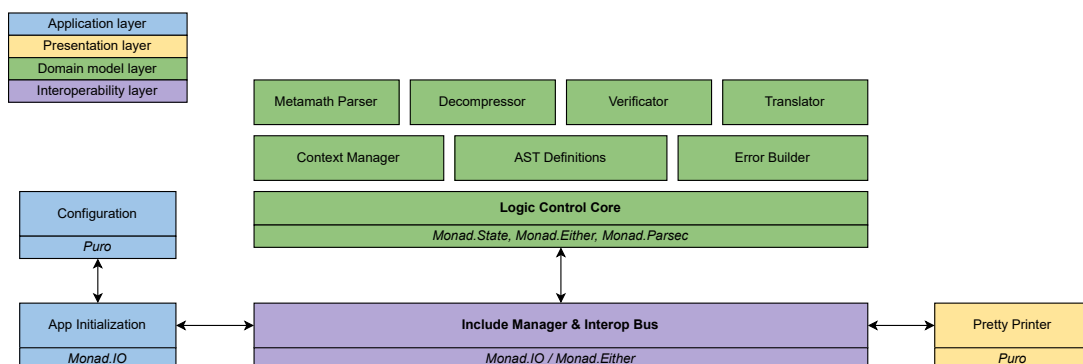


Figura 4.10: Diagramma dell'Architettura dettagliata del Traduttore Metis, modellato con l'ausilio dei "burger blocks" e la stratificazione cromatica del FDD.

Implementazione

5.1 Traduttore da linea di comando

Il *toolkit* mette a disposizione uno strumento da linea di comando per l'esecuzione del processo di traduzione, invocabile tramite la seguente istruzione:

```
hypatia processor translator <fileNames>
```

dove con `<fileNames>` si intendono i percorsi (*path*) dei file sorgente che si desidera tradurre.

Per configurare e specializzare il comportamento del traduttore, l'utente ha a disposizione diversi *flag* opzionali da inserire come argomenti:

- **--translateFrom <name>** (o **-t <name>**): Specifica il nome del linguaggio sorgente. Il valore di *default* è Metamath, che attualmente rappresenta anche l'unica opzione supportata.
- **--hypatiaComment** (o **-c**): Impone che eventuali commenti incontrati nel sorgente vengano tradotti come commenti nativi di Hypatia, ovvero come stringhe di testo libere dagli *escape* strutturali del linguaggio originale.
- **--languageSourceComment** (o **-s**): Specifica che i commenti debbano essere tradotti preservando la sintassi del linguaggio di provenienza (in questo caso Metamath), mantenendo quindi i delimitatori `$` (e `$`).
- **--includeFiles** (o **-i**): Indica al traduttore di analizzare ed elaborare anche gli eventuali file esterni inclusi all'interno del documento principale.
- **--depth <number>** (o **-d <number>**): Definisce la profondità massima consentita per lo "srotolamento" delle inclusioni. Di *default*, l'esplorazione non ha limiti e tutti i file annidati vengono analizzati.
- **--specificAlphabet** (o **-a**): Dichiara la volontà di generare un alfabeto *ad hoc* estratto dinamicamente dal file analizzato, anziché utilizzare l'alfabeto generale predefinito di Metamath.

Flusso di esecuzione

Il ciclo di vita della traduzione si articola in diverse fasi sequenziali. Inizialmente, il path del file sorgente viene passato come argomento dalla linea di comando (`<fileNames>`). Qualora sia presente il flag `-i`, la gestione delle direttive di inclusione viene demandata a un *parser* dedicato, che procede alla lettura ricorsiva dei file dipendenti.

Il programma costruisce quindi la struttura dati interna (l'albero sintattico) rappresentativa del file Metamath. Procedendo progressivamente con l'analisi di questa struttura logica e applicando le regole di mappatura definite nel capitolo precedente, il traduttore genera e restituisce in output il documento Hypatia finale.

5.2 Gestione delle inclusioni e risoluzione delle dipendenze

Nel linguaggio Metamath, la direttiva di inclusione permette a un file di importare il contenuto di altri file sorgente. Per gestire correttamente questo meccanismo all'interno del traduttore, è fondamentale prevenire il cosiddetto "problema del diamante" (raffigurato nell'immagine 5.1) e le inclusioni ricorsive, evitando al contempo l'analisi ripetuta e ridondante di uno stesso file.

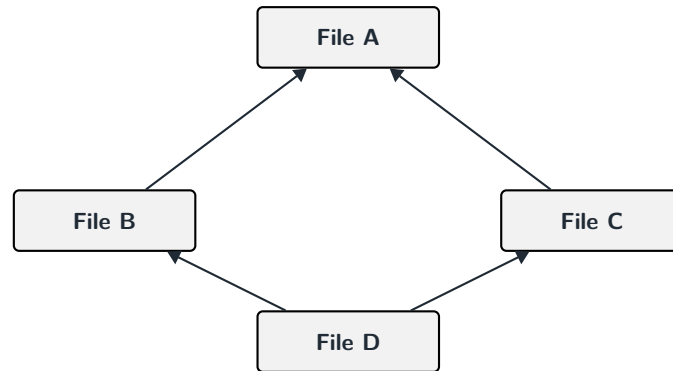


Figura 5.1: Rappresentazione del problema del diamante nell'inclusione tra file.

Per risolvere questa problematica, l'implementazione si avvale di una struttura dati a mappa (*Map*) che funge da registro di memorizzazione. Questa mappa associa il nome (o, per meglio dire, il *path*) di ogni file a una coppia di valori:

- Un valore booleano, che funge da *flag* di stato per indicare se il file è già stato tradotto, o ha cominciato già a esserlo;
- La struttura dati formale (l'albero sintattico) del documento Metamath associato a quel file.

Il flusso di esecuzione utilizza questa mappa come memoria di contesto che viene propagata durante la lettura:

- Quando viene incontrata una direttiva di inclusione, se il nome del file non è stato ancora analizzato (stato **False**), il sistema sospende temporaneamente l'analisi del file principale. Aggiorna il booleano, passa quindi alla traduzione dell'inclusione e ne salva il risultato, per poi ritornare all'esecuzione principale.
- Se, invece, il file risulta già presente nella mappa con il *flag* impostato a **true**, significa che il documento e il suo contesto sono già stati inclusi e tradotti in precedenza. In questo caso, il *parser* passa semplicemente oltre, ignorando l'istruzione ed evitando cicli infiniti o ridondanze.

Separazione tra logica pura e Monade IO

Un aspetto cruciale dell'implementazione in linguaggio Haskell riguarda la gestione delle operazioni di I/O (Input/Output). L'azione `readFile`, necessaria per estrarre il testo dei file da importare, genera effetti collaterali e appartiene alla **monade IO**.

Per rispettare i paradigmi della programmazione funzionale, che impongono una netta separazione tra il codice puro e quello impuro, l'interazione con il *file system* deve essere confinata alla superficie dell'applicativo (la CLI che interagisce con l'utente).

A tale scopo, è stato necessario implementare un *mini-parser* di pre-processamento. Questo componente scorre preliminarmente il documento originario al solo scopo di individuare le direttive d'inclusione, permettendo al programma di risolvere tutte le operazioni IO e caricare in memoria le stringhe di testo. Qui, se dichiarata una profondità specifica, questa viene decrementata a ogni strato di implementazione, fino al raggiungere lo 0. Solo successivamente, l'intero contesto purificato dagli effetti collaterali viene passato al *parser* principale per l'effettiva traduzione dell'albero sintattico.

5.3 Il Parser

Per lo sviluppo del modulo di analisi sintattica è stata impiegata la libreria *Megaparsec* in linguaggio Haskell. Tale scelta è stata motivata dalla sua architettura robusta e dall'ampia documentazione disponibile, caratteristiche che hanno agevolato significativamente l'implementazione delle logiche di *parsing*.

Una particolarità architetturale del *parser* sviluppato per i documenti Metamath riguarda la gestione dei commenti. Contrariamente alle implementazioni standard, che tendono a ignorare e scartare i commenti durante la fase di analisi lessicale, questo strumento è stato progettato per conservarli e integrarli all'interno dell'albero sintattico astratto (AST). In questo modo, le note e le spiegazioni presenti nel file originale possono essere fedelmente trasposte nel documento Hypatia generato.

Il risultato dell'analisi sintattica del file di input è la generazione di una struttura dati tipizzata, composta da una lista di costrutti formali (**Statement**) dotati dei rispettivi attributi e dalle relative dimostrazioni (**Proof**).

Di seguito è riportata la definizione dei tipi di dato in Haskell che costituiscono l'impalcatura logica del documento tradotto:

```
type Label = String
type Symbol = String

type Var = Symbol
type Const = Symbol

data Proof
  = ExtendedProof [Label]
  | CompressedProof [Label] String
  | NoProof
  deriving (Eq, Show)

data Statement
  = Constant [Symbol]
  | Variable [Symbol]
  | Disjoint [Var]
  | Floating Label Const Var
  | Essential Label Const [Symbol]
  | Scope [Statement]
  | Axiom Label Const [Symbol]
```

```
| Provable Label Const [Symbol] Proof
| Comment String
| Include Symbol
deriving (Eq, Show)
```

Ogni costruttore del tipo **Statement** modella uno specifico token strutturale del linguaggio Metamath:

- **Constant [Symbol]**: Dichiarare i simboli immutabili del sistema
(`$c math-symbol ... $.`).
- **Variable [Symbol]**: Dichiarare le metavariable del sistema
(`$v math-symbol ... $.`).
- **Disjoint [Var]**: Specifica le restrizioni di disgiunzione, impedendo che determinate variabili condividano sotto-espressioni
(`$d var var ... $.`).
- **Floating**: Modella un'ipotesi di tipo associata a una specifica variabile
(`label $f typecode variable $.`).
- **Essential**: Modella un'ipotesi logica o una premessa obbligatoria
(`label $e typecode math-symbol ... $.`).
- **Scope [Statement]**: Rappresenta un blocco di visibilità locale in cui circoscrivere variabili e ipotesi
(`$( ... $)`).
- **Axiom**: Definisce un'asserzione assiomatica, accettata come vera senza necessità di prova
(`label $a typecode math-symbol ... $.`).
- **Provable**: Definisce un teorema o lemma dimostrabile, associandolo all'oggetto che ne modella la dimostrazione
(`label $p typecode math-symbol $= proof $.`).
- **Comment String**: Conserva il contenuto testuale dei commenti originali incontrati nel file sorgente
(`$( ... $)`).
- **Include Symbol**: Gestisce le direttive di importazione, indicando il percorso di file esterni da srotolare
(`$( ... $)`).

5.4 Il Processo di Traduzione e la Gestione dello Stato

Terminata la fase di *parsing*, prende avvio il processo di traduzione vero e proprio. Il traduttore è stato progettato per cercare di arginare gli errori: sebbene l'applicativo integri una logica di verifica, il sistema non interrompe l'esecuzione al primo errore sintattico o semantico riscontrato. Al contrario, memorizza le anomalie in una struttura dedicata (un albero degli errori) e tenta di restituire una traduzione parziale basata sui frammenti di codice disponibili.

Per governare il flusso di esecuzione e mantenere traccia delle informazioni di contesto, l'implementazione si avvale della **Monade State**. Il tipo di dato che modella lo stato dell'applicativo durante la traduzione, denominato **StateHYP**, è definito come segue:

```
data StateHYP = StateHYP
  { stCtx :: MMContext
  , stIncluded :: M.Map String (Bool, String)
  , stHypCtx :: Seq LanguageConstruct
  , stSunyaMap :: M.Map String (ConstructSunya, Bool)
  , stErrors :: [IssueOptions]
  , stCurrentPath :: String
  , stEmergencyStop :: Bool
  , stDeclaredConstants :: S.Set String
  , stDeclaredVariables :: S.Set String
  }
```

Ogni campo all'interno di questa struttura dati ricopre un ruolo ben preciso durante l'avanzamento della traduzione:

- **stCtx**: rappresenta il contesto nativo Metamath.
- **stIncluded**: è una struttura a mappa (*Map*) utilizzata per la memorizzazione dei file inclusi, meccanismo essenziale per prevenire l'elaborazione ciclica e infinita delle dipendenze incrociate.
- **stHypCtx**: funge da contesto per Hypatia, comportandosi come un accumulatore sequenziale (*Seq*) in cui vengono salvati i costrutti formali man mano che il traduttore li genera.
- **stSunyaMap**: una mappa dedicata esclusivamente alle *label* dei costrutti Sunya tradotti. Si rivela essenziale durante la generazione delle prove, in particolare per risolvere i riferimenti incrociati richiesti dal passo logico **\$Reitero**.
- **stErrors**: un albero utilizzato per il tracciamento progressivo e non bloccante degli errori (sintattici o di unificazione) incontrati durante la traduzione. *IssueOptions* si articola in Errori, Warnings o *ErrorTree* interni per i file inclusi.
- **stCurrentPath**: memorizza il percorso (*path*) del file attualmente sotto analisi.
- **stEmergencyStop**: un *flag* di emergenza che viene attivato per bloccare l'esecuzione solo in caso di errori critici o irreversibili (come il mancato ritrovamento di un file sorgente essenziale).
- **stDeclaredConstants**: un insieme (*Set*) univoco che memorizza tutte le costanti matematiche e strutturali dichiarate nel sistema.
- **stDeclaredVariables**: un insieme (*Set*) univoco che memorizza tutte le variabili mutabili scoperte.

Durante l'attraversamento progressivo della lista dei costrutti Metamath (generata dal parser), ogni elemento viene analizzato, validato e integrato in questo stato.

Di seguito sono riportate le altre strutture dati utilizzate per la verifica e la traduzione del documento:

```
data MMContext = MMContext
  { stack :: Stack,
```

```

    globalLabels :: M.Map String LabelOptions
  }

data LabelOptions
  = LabelFloating String String
  | LabelEssential [String] String
  | LabelAssertion Assertion

```

Il contesto Metamath contiene lo *stack* della verifica e una mappa globale delle *label* scoperte fino al punto di esecuzione corrente, così da controllarne l'univocità.

```

newtype Stack = Stack
  { mmFrame :: [Frame]
  }

```

Lo stack non è altro che una lista di Frame.

```

data Frame = Frame
  { frameConstants :: S.Set String,
    frameVariables :: S.Set String,
    frameDisjoints :: S.Set (String, String),
    frameFloatings :: [(String, String, String)],
    frameFLabels :: M.Map String String,
    frameEssentials :: [(String, [String], String)],
    frameELabels :: M.Map String String
  }

```

- **frameConstants:** Un insieme che memorizza in modo univoco i simboli costanti (**\$c**) registrati all'interno di questo specifico blocco di visibilità.
- **frameVariables:** Un insieme dedicato al tracciamento delle variabili (**\$v**) dichiarate localmente. L'uso della struttura dati *Set* previene intrinsecamente le dichiarazioni duplicate all'interno del medesimo *frame*.
- **frameDisjoints:** Un insieme di coppie di stringhe che codifica le restrizioni di disgiunzione (**\$d**). Ogni coppia indica al validatore che le due variabili coinvolte sono indipendenti e non possono condividere variabili libere durante la fase di istanziazione.
- **frameFloatings:** Una lista di triple testuali che archivia le ipotesi di tipo (**\$f**) nel formato (*variabile*, *typecode*, *label*). La struttura a lista preserva l'ordine cronologico di apparizione, un requisito indispensabile per calcolare la corretta sequenza delle variabili obbligatorie (*mandatory variables*) nella firma dei teoremi Metamath.
- **frameFLabels:** Una mappa associativa (*Map*) progettata per l'accesso con complessità $O(\log n)$ alle ipotesi *floating* a differenza dell'utilizzo diretto della lista (con complessità $O(n)$). Utilizzando il nome della variabile come chiave e la *label* come valore, questa struttura applica e garantisce strutturalmente il vincolo formale di Metamath secondo cui una variabile può possedere un'unica dichiarazione di tipo attiva per blocco.
- **frameEssentials:** Una lista che colleziona le premesse logiche (**\$e**) introdotte nello *scope*, salvate nel formato (*typecode*, [*simboli_formula*], *label*). Conservare

fedelmente l'ordine top-down di dichiarazione è vitale, in quanto definisce la sequenza esatta con cui il risolutore RPN (*Reverse Polish Notation*) preleverà tali premesse dallo *stack* durante la convalida delle dimostrazioni.

- **frameELabels**: Una mappa di servizio per l'indicizzazione e il recupero rapido delle ipotesi *essential*. Il sistema concatena l'intera formula logica formattandola in un'unica stringa senza spazi, utilizzandola come chiave di ricerca per risalire istantaneamente all'etichetta (*label*) del costrutto.

Un aspetto cruciale in questa fase riguarda la gestione dello *scoping* per l'infrastruttura lessicale. In Metamath, quando un file importa variabili o costanti tramite un'inclusione, queste assumono validità globale e a cascata per tutto il file chiamante, dal punto esatto dell'inclusione in poi. Tuttavia, replicare questa logica sequenziale in Hypatia genererebbe errori di visibilità tra i frammenti. Per garantire la coerenza semantica della traduzione, si è optato per una strategia di *hoisting*: tutte le costanti e le variabili scoperte durante l'analisi vengono "sollevate" e raggruppate nei costrutti `$Lexicon` e `$Mutabilia` dichiarati in testa al costrutto `$Lingua` più esterno. In questo modo, l'infrastruttura alfabetica risulta sempre accessibile anche ai sotto-frammenti innestati creati per gestire le inclusioni.

5.5 Traduzione dei Costrutti diretti

Durante l'attraversamento progressivo della lista generata dal *parser*, i costrutti vengono immediatamente validati e, se possibile, tradotti nel corrispettivo Hypatia. Essi vengono quindi aggiunti all'accumulatore `stHypCtx` e alla mappa `stSunyaMap`, rivelandosi essenziali per la successiva traduzione delle dimostrazioni.

Questo approccio rende possibile un'elaborazione fluida a singolo scorrimento (*single-pass*). Di seguito sono descritte le tecniche di verifica e traduzione per ciascun costrutto di base:

- **Constant [Symbol]**: Il sistema verifica che la dichiarazione avvenga esclusivamente nel blocco più esterno (livello 0) e che i simboli dichiarati non risultino già attivi come variabili o costanti. Superati i controlli, i simboli vengono aggiunti all'insieme globale `stDeclaredConstants`.
- **Variable [Symbol]**: Viene controllato che i simboli non siano già stati dichiarati come costanti o come variabili attualmente attive nello *stack*. In caso di esito positivo, vengono inseriti in `stDeclaredVariables`.
- **Floating**: Il traduttore verifica l'univocità della *label* e si accerta che sia la variabile sia la costante associate siano effettivamente dichiarate e attive nel blocco corrente. Successivamente, la dichiarazione viene inserita tra le ipotesi *floating* del *frame* attivo.
- **Essential**: Similarmente alle ipotesi di tipo, si verifica l'univocità della *label* e la corretta dichiarazione dei simboli matematici coinvolti nella formula. L'ipotesi viene poi aggiunta all'elenco delle *essential* nello *stack*.

Coerentemente al meccanismo di *hoisting*, le variabili e le costanti vengono raccolte e aggiunte, insieme all'`$Alphabetum`, esclusivamente in testa all'albero di traduzione al termine dell'analisi. Le ipotesi *floating* ed *essential*, non possedendo una corrispondenza isolata diretta nel linguaggio Hypatia, vengono invece incapsulate dinamicamente all'interno dello *Schema* del teorema in cui si rendono necessarie.

5.6 Traduzione dei Commenti

La gestione dei commenti in fase di traduzione è determinata dai *flag* di configurazione passati dall'utente tramite CLI (`-c` oppure `-s`). Se viene richiesto di preservare la sintassi originale, il contenuto viene mantenuto all'interno dei delimitatori della sintassi di Metamath (`$ (` e `$)`). Qualora si richieda invece la conversione verso la sintassi di Hypatia, l'algoritmo analizza la struttura del testo: se individua la presenza del simbolo del dollaro (`$`) all'interno del blocco, il traduttore applica i delimitatori di isolamento multiriga (`/-` e `-/`) o a singola linea (`--`) per evitare collisioni lessicali con i costrutti del linguaggio *host*. In assenza del carattere `$`, il commento viene trascritto come blocco testuale libero.

5.7 Gestione degli Scope nella traduzione

Il meccanismo dei blocchi di visibilità locale in Metamath, delimitati dai token `${` e `$}`, richiede un'attenta gestione per conciliare il processo di validazione logica con l'architettura formale di Hypatia. La conversione si articola su due livelli:

- **Validazione logica (*Push/Pop*):** Quando il *parser* incontra un costrutto di apertura dello *scope*, il traduttore salva lo stato corrente allocando un nuovo *frame* vuoto in cima allo *stack* di contesto. Tutte le dichiarazioni locali (ipotesi e restrizioni di disgiunzione) vengono memorizzate all'interno di questo livello. Al raggiungimento del token di chiusura, il *frame* viene rimosso, deallocando le ipotesi temporanee per garantire che non interferiscano con la dimostrazione dei teoremi successivi.
- **Traduzione per Hypatia:** Sebbene la presenza di blocchi annidati in Metamath possa suggerire la necessità di generare costrutti `$Lingua` innestati, il traduttore adotta una strategia di appiattimento dello *scope*. In Metamath, i blocchi limitano la visibilità di variabili e ipotesi a specifiche porzioni di codice. Sfruttando le differenze architetturali tra i due sistemi, questa necessità di isolamento viene risolta su due fronti. Da un lato, la gestione delle variabili fa leva su una precisa differenza semantica: mentre in Metamath esse agiscono come metavariable vincolate al contesto, in Hypatia operano come *placeholder* testuali. Dall'altro, la generazione degli *Schemi* (assiomi e teoremi) viene effettuata estraendo esclusivamente le ipotesi attive nel *frame* corrente dello *stack* di valutazione in quel preciso istante. Poiché le ipotesi locali vengono incapsulate come premesse unicamente nei teoremi che rientrano nel loro campo di visibilità, la semantica dello *scoping* originale viene preservata. Di conseguenza, l'applicazione del meccanismo di *hoisting* — che dichiara l'infrastruttura lessicale a livello globale — mantiene inalterata la validità della traduzione e la coerenza formale del documento generato.

5.8 Traduzione delle *Assertion*

Per la traduzione di Assiomi o Dimostrazioni viene utilizzata la struttura:

```
data Assertion = Assertion
  { assDisjoints :: [(String, String)],
    assFloatings :: [(String, String, String)],
    assEssentials :: [(String, String)],
    assResult :: (String, [String])
  }
```

Il tipo di dato `Assertion` incapsula l'intera interfaccia logica di un assioma o di un teorema al momento della sua dichiarazione. Esso raccoglie non solo l'enunciato finale, ma anche il sottoinsieme strettamente necessario delle ipotesi e delle restrizioni attive nello *stack* in quell'istante, filtrando gli elementi ridondanti per isolare le sole componenti obbligatorie (*mandatory*). Di seguito è dettagliato il ruolo di ciascun attributo:

- **assDisjoints:** Una lista di coppie di stringhe che modella le restrizioni di disgiunzione obbligatorie. Durante la costruzione dell'asserzione, il sistema estrae l'insieme totale delle restrizioni (*\$d*) attive lungo tutti i *frame* aperti. Da queste, conserva esclusivamente quelle in cui *entrambe* le variabili coinvolte appartengono all'insieme delle variabili obbligatorie (ovvero le variabili che compaiono materialmente nell'enunciato finale o nelle premesse logiche).
- **assFloatings:** Una lista di triple strutturate come (*variabile*, *typecode*, *label*) che rappresenta le dichiarazioni di tipo necessarie per la validità dell'asserzione. Il sistema interroga lo *stack* ed estrae unicamente le ipotesi *floating* relative alle variabili obbligatorie calcolate in precedenza. La lista viene preservata nello specifico ordine gerarchico top-down, fondamentale per calcolare la corretta firma di un teorema durante la successiva fase di decompressione.
- **assEssentials:** Una lista di coppie nel formato (*[simboli_formula]*, *label*) che archivia tutte le premesse logiche (*essential hypotheses*) attualmente attive lungo i *frame* del contesto. Il primo elemento della tupla ricongiunge il *typecode* con il resto dei simboli della formula. Nella successiva conversione in Hypatia, questi elementi andranno a formare l'antecedente logico dello *Schema* (*\$Regula* o *\$Derivatio*).
- **assResult:** Una tupla strutturata come (*typecode*, *[simboli]*) che codifica l'enunciato finale (la tesi) dell'assioma o del teorema. Essa separa strutturalmente il codice di tipo dal resto della stringa e costituirà la conclusione logica dell'asserzione durante il processo di traduzione.

5.9 Traduzione degli Assiomi

Per tradurre un assioma Metamath *\$a* bisogna distinguere i due tipi di costrutto Hypatia in cui possono ricadere:

- **Regula:** Un assioma la cui tesi è formata da uno *Schema*
- **Principium:** Un assioma la cui tesi è formata da una *Phrase*

Come verrà fatto anche per i teoremi, il fattore discriminante è determinato analizzando le ipotesi attive nel contesto al momento della sua dichiarazione. L'algoritmo di traduzione (`translateAxiom`) distingue due scenari architetturali:

- **Assenza di premesse:** Se l'assioma non dipende da alcuna variabile (nessuna ipotesi *floating*) e non richiede premesse logiche (*essential*), esso viene tradotto come `$Principium`.
- **Presenza di premesse:** Se l'assioma è vincolato da ipotesi attive, la sua interfaccia deve essere strutturata logicamente. Il traduttore genera quindi uno *Schema* che pone le ipotesi come premesse e l'enunciato come conclusione, traducendolo come `$Regula`.

5.10 Il Decompressore delle Dimostrazioni

A differenza del normale processo di validazione eseguito dagli interpreti nativi di Metamath, il traduttore Metis ha l'assoluta necessità di decomprimere ogni prova prima di poterla eseguire. Questa esigenza è dettata da una differenza architetturale fondamentale: il dialetto base Sunya di Hypatia, essendo un nucleo formale puramente esplicito e testuale, non possiede un costrutto nativo equivalente per supportare logiche di compressione interne delle dimostrazioni.

Per questo motivo, è stato implementato un apposito modulo decompressore. L'algoritmo riproduce fedelmente la decodifica posizionale e su base intera illustrata nel capitolo dedicato all'architettura di Metamath.

Ricevendo in input il dizionario delle *label* (le etichette racchiuse tra parentesi tonde) e il blocco testuale compresso (la sequenza continua di lettere maiuscole), il decompressore calcola gli interi, espande gli eventuali blocchi salvati in memoria tramite il *Tag Z* e restituisce in *output* la lista completa, sequenziale ed espansa di tutte le *label* necessarie per ricostruire la cronologia dello *stack* della dimostrazione.

5.11 Traduzione della dimostrazione

Una volta ultimata la decompressione delle prove e completato il popolamento del contesto con tutti gli elementi necessari (infrastruttura lessicale e frammenti importati), il traduttore procede con la conversione effettiva delle singole dimostrazioni.

Il primo passo per tradurre un'asserzione dimostrabile (il costrutto `$p` di Metamath) nel suo equivalente Hypatia consiste nell'analizzare la sua firma per incanalarla in una delle due categorie sintattiche previste dal dialetto Sunya:

- **Productum:** Rappresenta un teorema la cui tesi è formulata come una *Phrase*, ovvero una stringa testuale composta esclusivamente da costanti immutabili.
- **Derivatio:** Rappresenta un teorema la cui tesi è strutturata come uno *Schema* logico, composto da un insieme di premesse e una conclusione.

Il criterio discriminante per questa classificazione risiede nell'analisi delle variabili e delle relative ipotesi. Se il teorema Metamath non presenta variabili (e quindi ipotesi *floatings*) o non ha ipotesi *essentials* che sono *mandatory*, la sua tesi sarà composta da sole costanti e verrà tradotta come un `$Productum`. Al contrario, se sono presenti variabili (che in Metamath richiedono obbligatoriamente un'ipotesi *floating* associata) o premesse

logiche (*essential*), l'asserzione deve obbligatoriamente essere tradotta come una **\$Derivatio**. Questa scelta è vincolata da due fattori architetturali di Hypatia:

1. Il passo logico **\$Praemitto** è l'unico strumento valido per introdurre un'ipotesi all'interno di una dimostrazione, e il suo utilizzo genera intrinsecamente uno *Schema* nello *stack* del verificatore.
2. Anche le eventuali ipotesi legate alle variabili d'appoggio (*dummy variables*) devono essere incluse formalmente tra le premesse, per rispettare il vincolo di *safety* degli schemi (come illustrato nella sezione successiva).

Chiarita la natura dell'asserzione, il processo si biforca nella costruzione della tesi:

Costruzione della tesi per Derivatio e Productum

Nel caso di una **Derivatio**, la tesi originale non può essere trasposta così com'è. Il traduttore deve costruire uno *Schema* completo che includa come premesse tutte le ipotesi del teorema (sia quelle obbligatorie che quelle legate alle *dummy variables*), ponendo come conclusione la tesi originale di Metamath. All'interno dello script di dimostrazione, queste ipotesi verranno introdotte sequenzialmente tramite il comando **\$Praemitto**.

Nel caso di un **Productum**, invece, l'assenza di premesse rende la tesi statica: la formula Metamath viene tradotta direttamente in una *Phrase* di Hypatia senza subire ulteriori elaborazioni strutturali.

L'algoritmo di traduzione dei passi logici

Indipendentemente dalla categoria del teorema, la traduzione del corpo della dimostrazione segue un algoritmo unificato che mappa lo *stack* RPN di Metamath nella logica esplicita di Hypatia:

1. **Gestione delle ipotesi:** Durante lo scorrimento delle *label* della prova, tutte le etichette riferite a ipotesi obbligatorie (sia *floating* che *essential*) vengono ignorate nella generazione dei comandi Hypatia, in quanto il loro valore viene gestito implicitamente dal risolutore dello *stack* interno del traduttore. Le ipotesi vengono infatti aggiunte a prescindere come costruito *Praemitto* prima di tutte le dimostrazioni delle *Derivatio*.
2. **Importazione degli assiomi/teoremi:** Quando l'algoritmo incontra una *label* relativa a un'asserzione (assioma o teorema), genera il comando **\$Assumo** per introdurla nel contesto con la sua **Reference** (se è la prima occorrenza), oppure **\$Reitero** se l'asserzione è già presente tra i *proof items* correnti.
3. **Istanziamento:** Analizzando lo stato dello *stack* interno, il traduttore calcola le corrette sostituzioni delle variabili e genera il comando **\$Obtineo**, accompagnato dalla clausola **\$cum** per esplicitare l'assegnazione dei valori (*Assignments*).
4. **Deduzione (Modus Ponens):** Viene generata una catena mirata di comandi **\$Reitero** e **\$Consequor** per combinare le premesse e l'asserzione istanziata, completando l'applicazione strutturale del Modus Ponens.
5. **Iterazione:** I passaggi precedenti vengono ripetuti ciclicamente per ogni *label* rimanente, fino al raggiungimento della tesi finale o all'esaurimento della prova.

5.12 Gestione delle variabili d'appoggio (Dummy Variables)

Un caso di traduzione particolarmente complesso riguarda la gestione delle *dummy variables*. In Metamath, queste variabili vengono utilizzate all'interno dello script di dimostrazione, richiedono di essere dichiarate in apposite ipotesi (*floating* o *essential*), ma non compaiono nell'enunciato finale del teorema.

Nel dialetto Sunya di Hypatia, omettere queste variabili dalla dichiarazione dello *Schema* finale comporterebbe una grave violazione della regola di *safety* (la quale impone che la conclusione non possa contenere variabili libere non coperte dalle premesse). Al contempo, il tentativo di eseguire un `$Obtineo` su un teorema che nasconde delle variabili d'appoggio fallirebbe, poiché non vi sarebbero valori nello *stack* da sostituire alla variabile "fantasma".

Per risolvere questo disallineamento, il modulo Metis adotta un approccio basato sulla generazione di astrazioni sintetiche. Le ipotesi relative alle *dummy variables* vengono forzatamente incluse nella firma della `$Derivatio` principale, che viene dimostrata regolarmente. Immediatamente dopo, il traduttore inietta automaticamente nel documento una `$Regula` fittizia strutturata nel seguente modo:

```
Teorema_con_ipotesi_dummy $- Teorema_senza_dummy
```

Questa strategia architetturale consente al traduttore di invocare la *Derivatio* completa per poi applicare la *Regula* sintetica tramite un `$Consequor`, isolando e rimuovendo le variabili d'appoggio. In questo modo, l'interfaccia pubblica del teorema risulta pulita e i successivi sviluppi logici possono fare riferimento alla tesi pura, mantenendo intatta la verificabilità strutturale in quanto le *dummy variables*, da definizione, non sono necessarie al fine del teorema in cui sono usate e quindi la tesi è dimostrata anche senza la loro presenza.

5.13 Inclusioni esterne e teoremi non risolti

Un'ultima casistica riguarda la robustezza del traduttore di fronte a dipendenze mancanti. Qualora una *label* invocata nella dimostrazione risulti sconosciuta, scenario tipico quando l'utente omette il flag `-i` per l'autorizzazione degli *include*, oppure quando la profondità massima di esplorazione (`-d`) viene superata, il programma non interrompe l'esecuzione.

In questi frangenti, il traduttore si avvale del comando `$Divino`. Il teorema mancante o non espanso viene assunto come congettura, producendo una pseudo-prova. Questa gestione tollerante degli errori garantisce che la porzione di albero sintattico correttamente risolta venga tradotta e conservata, consentendo il riutilizzo e l'ispezione parziale del documento Hypatia generato.

Se invece non sono presenti variabili o costanti chiamate nel testo inclusore, non sapendo come interpretare il testo, la traduzione viene immediatamente bloccata in quanto sarebbe impossibile ricondurre i simboli ad una determinata categoria, accertarsi della loro dichiarazione e dichiarare l'alfabeto.

5.14 Flusso di Esecuzione (Sequence Diagram)

Per comprendere a fondo l'interazione tra i layer architetturali e i singoli sottomoduli durante l'elaborazione di un documento, il *Sequence Diagram* del ciclo di vita dell'applicazione è stato diviso in tre fasi logiche, rispecchiando l'alternanza tra operazioni IO impure e operazioni pure dall'elaborazione del documento fino alla restituzione in output all'utente. Come illustrato nella Figura 5.2, la prima fase è dominata dall'*Interoperability Layer*. La CLI e l'*Include Manager* interagiscono con il file system, gestendo l'I/O impuro per risolvere ricorsivamente le dipendenze del documento Metamath e fornire al dominio una stringa pronta per il parsing.

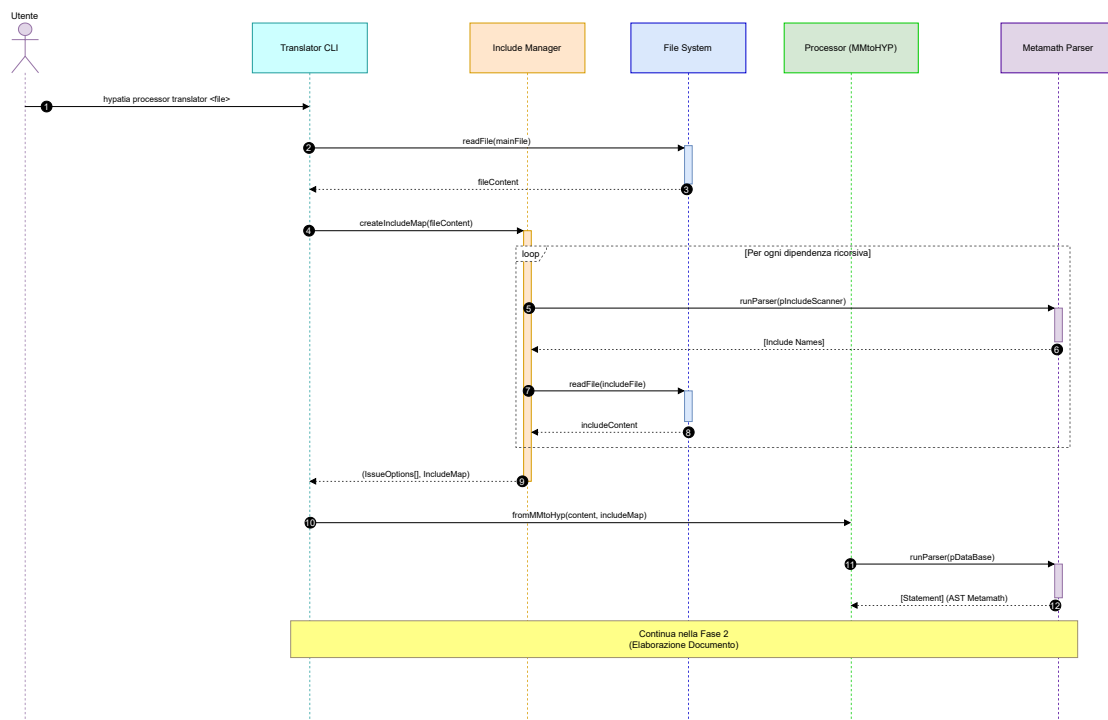


Figura 5.2: Sequence Diagram (Parte 1): Setup iniziale, lettura dei file e generazione del primo AST di base.

Una volta inizializzato il **Processor**, il sistema entra nel dominio puramente funzionale (Figura 5.3). Il *loop* centrale evidenzia la sequenza di invocazione dei moduli di calcolo (Decompressor, Translator, Verificator).

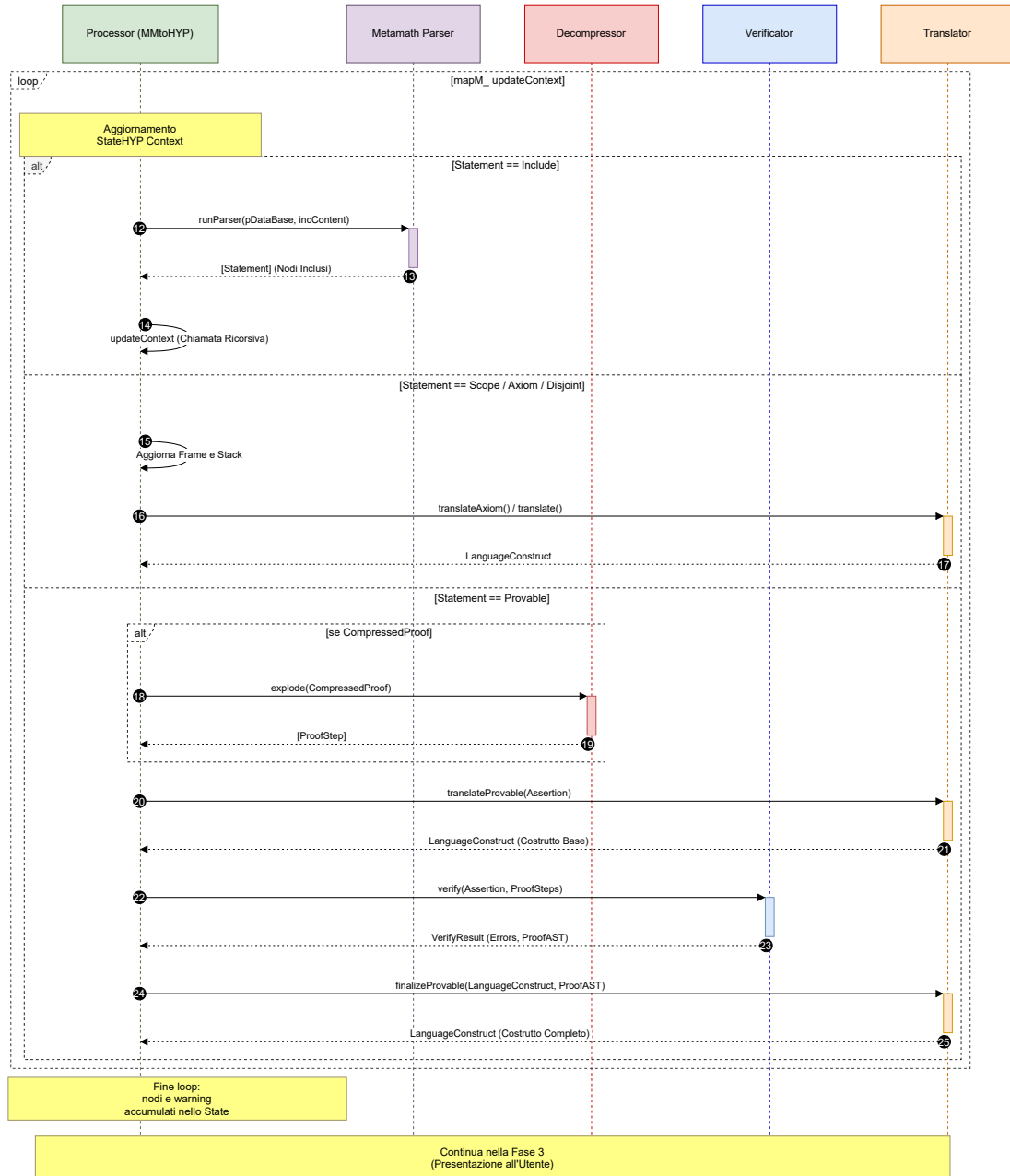


Figura 5.3: Sequence Diagram (Parte 2): Esecuzione del loop monadico di valutazione.

Infine, la struttura dati elaborata fino a quel momento e gli errori raccolti vengono mandati al **PrettyPrinter**, che li formatta e rimanda il formato leggibile alla **CLI** (Figura 5.4).

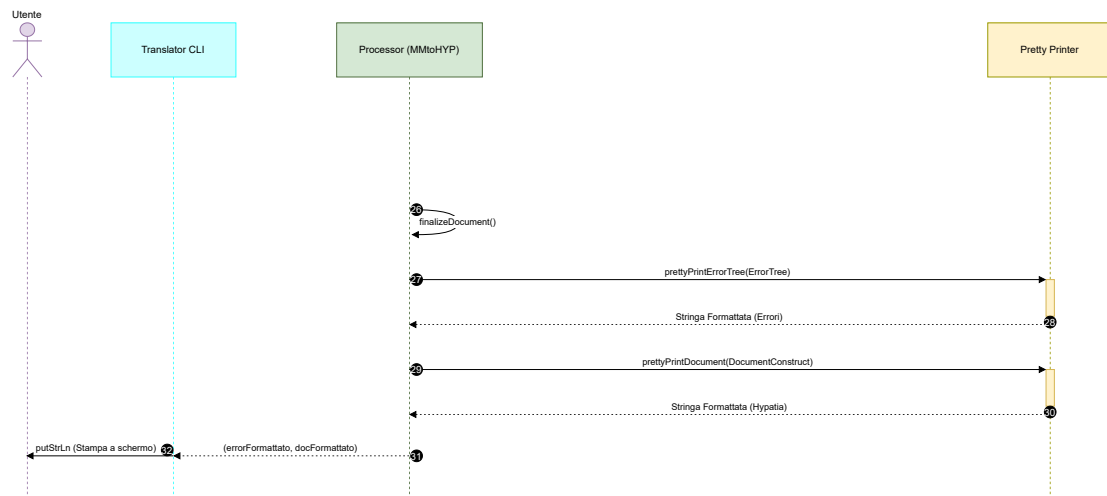


Figura 5.4: Sequence Diagram (Parte 3): Rendering dell'output.

Validazione e test

6.1 Test effettuati

Al fine di garantire la correttezza e la robustezza del software, è stata definita una suite di test sui moduli analizzati nei capitoli precedenti. Nello specifico, per la validazione del flusso di esecuzione relativo ai database Metamath, è stato impiegato il framework di testing *Hspec* per Haskell.

Le suite di test sono state progettate per coprire i principali flussi e casi limite del codice. Di seguito sono riportati i report di esecuzione dei test, tutti superati con successo:

6.1.1 Struttura e Operazioni Metamath

Questa suite di test si concentra sulla corretta gestione dello stato interno e delle regole di *scoping*. Le verifiche assicurano che le operazioni di manipolazione dello stack (*push*, *pop*) avvengano in modo sicuro e che il sistema applichi bene i vincoli formali sulle dichiarazioni. In particolare, i test verificano nei casi considerati la capacità del modulo di impedire la duplicazione di costanti, l'uso di variabili non dichiarate e l'applicazione di ipotesi a simboli inesistenti, garantendo la generazione di *Assertion* sintatticamente corrette.

```
Hypatia.Toolkits.Metamath
Metamath Stack & Frame Operations
  Base Stack operations
    EmptyStack has no frame [PASS]
    Push and Pop [PASS]
    Top return currentFrame [PASS]
    Add fails if there is no frame on the stack [PASS]
  Edge Cases e Statement non supportati
    Fails to add a constant in a nested scope [PASS]
    Fails on unsupported statements like Include [PASS]
    Extracts variables correctly from a formula [PASS]
  Assert su Provable ($p)
    Generates the correct Assertion object for a Provable
      statement [PASS]
  Constants ($c) and Variables ($v)
    Recognizes existing constants and variables [PASS]
    Prevents declaring a constant if it is already a variable [PASS]
    Prevents declaring a variable if it is already a constant [PASS]
    Prevents duplicate constants [PASS]
    Prevents duplicate variables [PASS]
  Floating ($f) ed Essential ($e)
    lookUpFloating finds label from the variable name [PASS]
    hasFloating verify the existence of a variable [PASS]
    lookUpEssential finds label from the formula [PASS]
    Prevents floating on non-existent variables [PASS]
    Prevents floating on non-existent constants [PASS]
```

```

Prevents duplicate of essential labels [PASS]
Disjoints ($d)
  Treats disjoints correctly [PASS]
  Disjoint with less then 2 variable, should fail [PASS]
  Disjoint with no-declared variable, should fail [PASS]
Scope e Multiple Frame
  Stack scoping [PASS]
Assert
  Reject all it's not an axiom [PASS]
  Generates the correct Assertion object for ax-mp [PASS]

```

6.1.2 Decompressore Metamath

I test di questa sezione validano la robustezza dell'algoritmo di decompressione delle dimostrazioni. Viene verificata la corretta decodifica della notazione, e la precisa mappatura dei passi logici. La suite collauda inoltre la gestione del tag di memorizzazione Z per il salvataggio e il recupero delle sotto-dimostrazioni, assicurandosi che il sistema intercetti casi anomali come il superamento dei limiti degli indici o la presenza di caratteri illegali nella stringa compressa.

```

Hypatia.Toolkits.DecompressorMM
  Metamath Decompressor
    decode
      A-T (1-20) [PASS]
      U-Y [PASS]
      Z [PASS]
      Ignore spaces and \n [PASS]
      Fails on illegal characters [PASS]
    Steps
      External labels (1 <= i <= m) [PASS]
      Local labels (m < i <= m+n) [PASS]
      SaveMarkers (-1) [PASS]
      LoadMarkers (i > m+n) [PASS]
    decompress (Pipeline base decode + steps)
      Decompress a string and map steps [PASS]
    Tree
      1 for leaf node [PASS]
      Recursive tree of assertion [PASS]
      Index out of bounds [PASS]
      Not declared label [PASS]
      Tree fails with ErrorAtIndexCompressed if it encounters
        a SaveMarker or LoadMarker [PASS]
    Explode
      successfully recovers multiple saves using a series of Zs [PASS]
      fails if the LoadMarker index exceeds the available saves [PASS]
      Expand correctly with Z [PASS]
      Fails with a LoadMarker not saved [PASS]
      Explode handles empty compressed string triggering the
        expand base case [PASS]

```

6.1.3 Parser Metamath

Questa *suite* analizza il comportamento del modulo di (*parsing*). Ogni costrutto (costanti, variabili, ipotesi, asserzioni, blocchi di *scope* e direttive di inclusione) viene sottoposto a test

per garantirne la corretta trasposizione nell'albero sintattico (AST). Particolare attenzione è dedicata alla gestione degli errori, come l'assenza di delimitatori di chiusura o la presenza di commenti tra i vari costrutti, al fine di prevenire interruzioni anomale durante l'analisi del documento.

Hypatia.Toolkits.ParserMM

Parser Metamath - Hypatia

pConstant (\$c)

Parses a single constant [PASS]

Parses multiple constants without extra spaces [PASS]

Parses no constants, should throw error [PASS]

pVariable (\$v)

Parses a single variable [PASS]

Parses a multiple variables without extra spaces [PASS]

Parses no variables, should throw error [PASS]

pDisjoint (\$d)

Parses a couple of disjoint [PASS]

Parses more than 2 disjoint [PASS]

pHypothesis (\$f)

Parses a Floating [PASS]

pHypothesis (\$e)

Parses an Essential [PASS]

Parses Labels with special symbols (.-_) [PASS]

pAssert (\$a)

Parses an Axiom [PASS]

pAssert (\$p) e Proof

Provable assertion with extended proof [PASS]

Provable assertion with compressed proof [PASS]

pBlock (\${ \$})

parses a nested scope [PASS]

parses a scope containing multiple statements [PASS]

pComment (\$(\$))

parses an isolated comment [PASS]

pDataBase

parses an entire database [PASS]

correctly handles EOF and fails if there is invalid

residual text [PASS]

Edge Cases and Error Tolerance

correctly ignores multiple spaces and newlines at the

beginning and around tokens [PASS]

fails explicitly if the closing delimiter \$. is missing [PASS]

fails explicitly if the scope block closure \$} is missing [PASS]

Parses a file inclusion directive (\$[\$]) via pDataBase [PASS]

Parses statements with interlaced comments [PASS]

Handles missing tokens in \$f, \$e, \$a gracefully

(fallback to empty strings) [PASS]

6.1.4 Verificatore Metamath

L'ultima *suite* analizza il motore di verifica logica. I test confermano che le sostituzioni testuali per l'applicazione del *Modus Ponens* avvengano preservando la coerenza semantica dello *stack* e che le restrizioni sulle variabili disgiunte (\$d) siano applicate per prevenire collisioni. Infine, si verifica la resistenza del modulo rispetto alle anomalie logiche del

file in input: in presenza di etichette mancanti, l'esecuzione non viene interrotta, ma la dimostrazione viene declassata a pseudo-prova (tramite *WarnDivino*), segnalando l'anomalia e permettendo la prosecuzione della traduzione.

Hypatia.Toolkits.VerificatorMM

Verificator.applyStep

- correctly preserves the stack and does not fail on identity substitutions (e.g., *ph* replaced with *ph*) [PASS]
- successfully applies *LabelFloating* by pushing it onto the stack (e.g., step '*vx*') [PASS]
- returns an error for a non-existent label in the context [PASS]
- returns an error for stack underflow (missing arguments for the assertion) [PASS]
- returns an error for typecode mismatch (e.g., *wff* instead of *setvar*) [PASS]
- successfully applies an essential hypothesis (*\$e*) if the stack matches the construct (Modus Ponens) [PASS]
- returns an error if an essential hypothesis substitution fails [PASS]

Rules on Disjoint Variables (*\$d*)

- returns an error for a crossed *\$d* violation on 3 or more variables [PASS]
- returns an error (*\$d* Violation) if a target variable is replaced with itself in disjoint branches [PASS]
- returns an error if the substituted variables are not explicitly declared disjoint in the target theorem [PASS]
- successfully applies the *ax-ext* axiom if the target variables are declared disjoint [PASS]

Verificator.verify

- successfully verifies the entire chain of steps for the syntactic construction of *ax-ext* [PASS]
- returns a parsing error if the proof ends with a stack size > 1 [PASS]
- returns False if the proof concludes without execution errors but produces an incorrect assertion [PASS]

Branch Coverage & Edge Cases

- buildSubst* returns *ErrorOfProgram* on invalid formula (missing typecode) [PASS]
- verify* uses *ProductProofList* when target has no floatings/essentials and *dummyMap* is empty [PASS]
- applyStep* pushes *?UNKNOWN?lbl* when a label is completely missing from global context [PASS]
- resolveSchema* retrieves *Regula* from *NotSubset_Dummy* suffix when *isDummy* is True [PASS]
- alreadyAssumed* prevents duplicate *Assumo* generating correct structural branches [PASS]
- resolveSchema* handles unmatched constructs safely via *Nothing* (e.g. *Principium* fallback) [PASS]
- isDivino* generates *WarnLabelNotFound* when label is legitimately missing from *hypMap* [PASS]
- Throws a runtime error when *unsafeBuildSchema* is called with no premises [PASS]
- Injects *WarnDivino* and closes proof when a

label is missing and stack is invalid [PASS]

6.2 File di test Metamath

La fase conclusiva della validazione ha previsto il collaudo del processo di traduzione sull'intero ciclo di esecuzione. A tale scopo, sono stati tradotti diversi file sorgente Metamath, di dimensioni e complessità crescenti, estratti dalla *repository* ufficiale del progetto. L'analisi dei risultati ottenuti da questi file è risultata funzionale per l'individuazione e la risoluzione delle criticità logiche durante lo sviluppo.

Di seguito si riportano i risultati dei collaudi più significativi.

6.2.1 Gestione degli errori strutturali: anatomy-bad1.mm

Il primo test è stato condotto sul file `anatomy-bad1.mm`, il quale contiene una dimostrazione intenzionalmente errata a livello strutturale.

L'output atteso dal compilatore originale Metamath segnala che lo stack RPN contiene più di un elemento al termine della dimostrazione:

```
?Error on line 27 of file "anatomy-bad1.mm" at statement 8, label "wnew", type "$p":
wnew $p wff ( s -> ( r -> p ) ) $= ws wr wp w2 ws $.
      ~
After proof step 5 (the last step), the RPN stack contains 3 entries: "ws" (step 1), "w2" (step 4),
and "ws" (step 5). It should contain exactly one entry.
```

Il traduttore Metis rileva correttamente l'anomalia logica, specificando quale teorema risulta errato, ma riesce a tradurre e preservare la parte di codice (e di commenti) strutturalmente valida:

```
[Error] In statement: wnew
- The proof for wnew is logically or structurally INCORRECT.

$Hypatia Sunya $:
$Lingua /mnt/c/Users/espov/Downloads/anatomy-bad1.mm $:
$Alphabetum ( ) - > f p q r s w $.
$Intervalla \nh \sp \th \tv \cn \ln \pn $.
$Lexicon ( ) -> wff $.
$Mutabilia p q r s $.
anatomy.mm
PUBLIC DOMAIN DEDICATION

This file is placed in the public domain per the Creative Commons Public
Domain Dedication. http://creativecommons.org/licenses/publicdomain/

Norman Megill
This file is the introductory formal system example described
in Chapter 4 of the Metamath book, "Anatomy of a proof".
Declare the constant symbols we will use
Declare the metavariables we will use
Declare wffs
$Regula w2 $: wff p $- ${ wff q $- wff ( p -> q ) $} $.
Prove a simple theorem.
$.
$.

End of files.
```

6.2.2 Traduzione nominale: anatomy.mm

Il file `anatomy.mm` rappresenta la controparte corretta del test precedente. In questo caso, la dimostrazione rispetta i vincoli formali. Il traduttore elabora il file senza sollevare eccezioni, restituendo la *Derivatio* completa e la corretta espansione dei passi logici in Hypatia:

```

No errors detected.

$Hypatia Sunya $:
$Lingua /mnt/c/Users/espov/Downloads/anatomy.mm $:
$Alphabetum ( ) - > f p q r s w $.
$Intervalla \nh \sp \th \tv \cn \ln \pn $.
$Lexicon ( ) -> wff $.
$Mutabilia p q r s $.

-- [Commenti iniziali omessi per brevità] --

$Regula w2 $: wff p $- ${ wff q $- wff ( p -> q ) }$ $.
Prove a simple theorem.
$Derivatio wnew $:
  wff p $- ${ wff r $- ${ wff s $- wff ( s -> ( r -> p ) ) }$ }$
$ex
  $Probatione
    $Praemitto wff p $.
    $Praemitto wff r $.
    $Praemitto wff s $.
    $Assumo wff p $- ${ wff q $- wff ( p -> q ) }$ $per $Regulam w2 $.
    $Obtineo wff r $- ${ wff p $- wff ( r -> p ) }$ $cum p $= r $, q $= p $.
    $Reitero wff r $.
    $Reitero wff r $- ${ wff p $- wff ( r -> p ) }$ $.
    $Consequor wff p $- wff ( r -> p ) $.
    $Reitero wff p $.
    $Reitero wff p $- wff ( r -> p ) $.
    $Consequor wff ( r -> p ) $.
    $Reitero wff p $- ${ wff q $- wff ( p -> q ) }$ $.
    $Obtineo wff s $- ${ wff ( r -> p ) $- wff ( s -> ( r -> p ) ) }$ $cum p $= s $, q $= ( r ->
p ) $.
    $Reitero wff s $.
    $Reitero wff s $- ${ wff ( r -> p ) $- wff ( s -> ( r -> p ) ) }$ $.
    $Consequor wff ( r -> p ) $- wff ( s -> ( r -> p ) ) $.
    $Reitero wff ( r -> p ) $.
    $Reitero wff ( r -> p ) $- wff ( s -> ( r -> p ) ) $.
    $Consequor wff ( s -> ( r -> p ) ) $.
  $.
$.
$.
$.

```

6.2.3 Risoluzione delle inclusioni: demo0-includer.mm

Per verificare la corretta risoluzione della gestione delle inclusioni, è stata testata la coppia di file `demo0-includer.mm` e `demo0-includee.mm`.

Attivando la flag `-i` (autorizzazione delle inclusioni), il traduttore elabora il file dipendente, generando un'istanza `$Lingua` interna. Si noti come le costanti vengano sottoposte a *hoisting*, venendo dichiarate nel livello più esterno per garantire la visibilità:

```

No errors detected.

$Hypatia Sunya $:
$Lingua /mnt/c/Users/espov/Downloads/demo0-includer.mm $:
$Alphabetum ( ) + - 0 = > P Q e f m r s t w | $.
$Intervalla \nh \sp \th \tv \cn \ln \pn $.
$Lexicon ( ) + -> 0 = term wff |- $.
$Mutabilia P Q r s t $.

-- [Commenti e definizione Lingua interna omessi per brevità] --

$Derivatio th1 $:
  term t $- |- t = t

```

```
$ex  
$Probatione  
-- dimostrazione omessa per brevità'  
$.  
$.  
$.  
$.  
$.
```

Se lo stesso test viene eseguito omettendo la flag di inclusione, il traduttore solleva immediatamente un’eccezione. Non avendo accesso al file incluso (il quale contiene tutte le definizioni delle variabili e delle costanti), il sistema non possiede l’alfabeto necessario per analizzare le asserzioni, arrestando logicamente la verifica:

```
[Error] In statement: th1
- Alphabetum creation error:
  Simboli utilizzati ma non dichiarati (manca $c o $v): |- t = t
```

6.2.4 Stress test di unificazione: big-unifier.mm

Al fine di valutare il consumo di memoria e la robustezza dell’algoritmo di unificazione, il traduttore è stato testato su `big-unifier.mm` (21.5 KB). Questo file è noto per contenere espressioni formali di grandi dimensioni in input che si espandono fino a generare molti simboli durante la risoluzione. Il file è stato elaborato con successo, producendo un file di 5.2 MB:

```
No errors detected.

$Hypatia Sunya $:
  $Lingua /mnt/c/Users/espov/Downloads/big-unifier.mm $:
    $Alphabetum ( ) , - 0 1 2 3 4 5 6 7 8 9 e f u v w x y z | $.
    $Intervalla \nh \sp \th \tv \cn \ln \pn $.
    $Lexicon ( ) , e wff |- $.
    $Mutabilia u v v1 v10 v11 v2 v3 v4 v5 v6 v7 v8 v9 w x y z $.

-- [Commenti e dichiarazioni omissi per brevita'] --

The first axiom.
$Regula ax-maj $: wff x $- ${ wff y $- ${ wff z $- ${ wff w $- ${ wff v $- ${ wff u $- ${ wff v6
$- ${ wff v7 $- ${ wff v8 $- ${ wff v9 $- ${ wff v10 $- ${ wff v11 $- |- e ( e ( e ( e ( e ( x ,
e ( y , e ( e ( e ( e ( e ( z , e ( e ( e ( z , u ) , e ( v , u ) ) , v ) ) , e ( e ( w , e ( e (
e ( w , v6 ) , e ( v7 , v6 ) ) , v7 ) ) , y ) ) , v8 ) , e ( v9 , v8 ) ) , v9 ) ) ) , x ) , v10
) , e ( v11 , v10 ) ) , v11 ) $} $} $} $} $} $} $} $} $} $} $.

-- [Restante output omissi per brevita'] --
```

6.2.5 Sistemi logici estesi: hol.mm

L'ultimo test di rilievo è stato effettuato su `hol.mm`, un file di grandi dimensioni (94.7 KB) che formalizza la logica di ordine superiore (Higher-Order Logic). Il test conferma la capacità di Metis di generare l'alfabeto, il lessico e le variabili per sistemi formali dotati di una semantica densa e di un set di caratteri esteso, con un output di 16.32 MB.

6.3 Test con il Verificatore

A ulteriore conferma della correttezza del processo di traduzione, i documenti Hypatia generati in output sono stati sottoposti al *parser* e al verificatore ufficiale del linguaggio, sviluppati da Roberto Fiorino [19]. Questa fase di collaudo ha permesso di individuare e risolvere diverse problematiche architetturali descritte nei capitoli precedenti, tra cui la

corretta concatenazione dei passi logici ($\$Assumo$, $\$Reitero$, $\$Obtineo$ e $\$Consequor$) e la gestione della visibilità delle variabili tramite il meccanismo di *hoisting*.

L'esito dei test ha certificato la validità strutturale e logica di *big-unifier.mm*, *demo0-includee.mm* (e di conseguenza *demo0-includee.mm*) e *anatomy.mm*.

Tuttavia è stata invece necessaria una rielaborazione testuale per la verifica dell'output generato a partire da *hol.mm*. Sebbene il traduttore di Metis lo abbia tradotto correttamente, il documento finale conteneva costrutti che, pur essendo univoci per la logica di Metamath, introducono ambiguità grammaticali in Hypatia.

Poiché l'attuale implementazione del verificatore supporta esclusivamente la modalità di interpretazione **Unica**, la quale esige un singolo albero di derivazione deterministico e scarta a priori la presenza di ambiguità sintattiche, è stato necessario intervenire manualmente sul file tradotto per rinominare alcune costanti ed eliminare le collisioni lessicali. Un esempio di errore, generato dall'ambiguità tra il simbolo T. e la stringa T seguita dal terminatore ., è illustrato di seguito:

```

      |
82 |      $Principium kt $: term T. $.
      |
the phrase starting here can be read in more than one way, for example
  as term T .
  as term T.

```

Questa anomalia deriva dal diverso approccio di elaborazione adottato dai due sistemi. Nella traduzione originale, le intestazioni lessicali dichiaravano esplicitamente sia T. come singola costante, sia T e . (rispettivamente variabile e costante):

```

$Alphabetum ! ( ) , - . / 1 : = > ? @ A B C F R S T [ \ ] a b c ... | ~ $.
$Intervalla \nh \sp \th \tv \cn \ln \pn $.
$Lexicon ! ( ) , -> . /\ 1-1 : = ==> ? ?! @ F. T. [ \ \/ ] bool ... |= ~ $.
$Mutabilia A B C F R S T al be de f g ga p q x y z $.

```

Il *parser* di Hypatia, non assumendo a priori un'interpretazione di tipo **longest-match**, non dispone delle informazioni sufficienti per determinare autonomamente se la stringa T. debba essere interpretata come la costante T., oppure come la variabile T seguita dalla costante ..

A seguito delle correzioni applicate direttamente all'infrastruttura lessicale del documento generato, la validazione di *hol.mm* si è conclusa con esito positivo.

Conclusioni

L'implementazione del modulo Metis stabilisce un ponte semantico e sintattico tra il linguaggio Metamath e l'infrastruttura di Hypatia. L'architettura sviluppata copre attualmente la quasi totalità dei costrutti e dei meccanismi presenti nel linguaggio originale (tranne per quanto riguarda il costrutto di disgiunzione $\$d$), garantendo un *parsing* totale, la decompressione delle prove e una traduzione quasi completa alla logica del dialetto Sunya.

Tuttavia, il trasferimento di teoremi tra due sistemi formali eterogenei lascia aperti alcuni margini di miglioramento e di ricerca. I futuri sviluppi del traduttore si concentreranno principalmente sui seguenti aspetti:

- **Gestione nativa delle variabili disgiunte ($\$d$):** Poiché il dialetto Sunya di Hypatia non prevede attualmente un costrutto primitivo per forzare la disgiunzione tra due o più variabili, questo vincolo necessita di una codifica esplicita. La strategia più promettente prevede di sfruttare il linguaggio oggetto nativo di Hypatia per gestire le frasi e generare teoremi di supporto per garantire la separazione delle sostituzioni ad ogni istanziazione. Tale implementazione richiederà una revisione completa del meccanismo di traduzione, che verrà adottata prossimamente.
- **Integrazione con l'ecosistema Hypatia in evoluzione:** Essendo Hypatia un progetto di ricerca in fase di sviluppo attivo, alcune funzionalità del *framework* non sono ancora state integrate nel traduttore, come l'elaborazione di un meccanismo in grado di determinare a priori l'univocità del *parsing* della traduzione. Quest'ultima permetterà al sistema di selezionare dinamicamente la modalità di interpretazione più appropriata (Unica, Omnes, Valida, Aliqua).
- **Flag per disambiguare la traduzione con Unica:** Un altro cambiamento che potrebbe essere apportato è l'introduzione di una flag aggiuntiva che, come è stato fatto per la verifica di `hol.mm`, provi a disambiguare direttamente il testo per farlo gestire nella modalità Unica.

Questi aggiornamenti verranno implementati di pari passo con la maturazione dell'intero *framework*. In definitiva, il traduttore Metis pone le fondamenta per l'importazione e la certificazione di Metamath, ed è destinato a evolvere con i futuri cambiamenti del progetto Hypatia.

Bibliografia

- [1] A. Asperti et al. «The Matita Interactive Theorem Prover». In: *11*. 6803. 2011, pp. 64–69. DOI: 10.1007/978-3-642-22438-6_7.
- [2] Dario Amodei et al. «Concrete problems in AI safety». In: *arXiv preprint arXiv:1606.06565* (2016).
- [3] Abhishek Anand et al. «Towards certified meta-programming with typed Template-Coq». In: *International Conference on Interactive Theorem Proving*. Springer. 2018, pp. 20–39.
- [4] Ali Assaf et al. «Dedukti: a logical framework based on the $\lambda\Pi$ -calculus modulo theory». In: (2016).
- [5] Gilles Barthe et al. «EasyCrypt: A Tutorial». In: *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*. A cura di Alessandro Aldini, Javier Lopez e Fabio Martinelli. Cham: Springer International Publishing, 2014, pp. 146–166. ISBN: 978-3-319-10082-1. DOI: 10.1007/978-3-319-10082-1_6. URL: https://doi.org/10.1007/978-3-319-10082-1_6.
- [6] Mathieu Boespflug, Quentin Carbonneaux e Olivier Hermant. «The $\lambda\Pi$ -calculus Modulo as a Universal Proof Language.» In: *PxTP*. 2012, pp. 28–43.
- [7] N.G. de Bruijn. «The Mathematical Language Automath, its Usage, and Some of its Extensions». In: *Symposium on Automatic Demonstration*. 125. Springer, 1970, pp. 29–61.
- [8] N.G. de Bruijn. «A Survey of the Project Automath». In: *Selected Papers on Automath*. Vol. 133. North-Holland, 1994, pp. 73–100. DOI: 10.1016/S0049-237X(08)70203-9.
- [9] N.G. de Bruijn. «Reflections on Automath». In: *Selected Papers on Automath*. Vol. 133. North-Holland, 1994, pp. 201–228. DOI: 10.1016/S0049-237X(08)70205-2.
- [10] N.G. de Bruijn. «The Mathematical Language Automath, its Usage, and Some of its Extensions». In: *Selected Papers on Automath*. Vol. 133. North-Holland, 1994, pp. 73–100. DOI: 10.1016/S0049-237X(08)70200-3.
- [11] N.G. de Bruijn. «The Mathematical Vernacular, a Language for Mathematics with Typed Sets». In: *Selected Papers on Automath*. Vol. 133. North-Holland, 1994, pp. 865–935. DOI: 10.1016/S0049-237X(08)70231-3.
- [12] N.G. de Bruijn. «Verification of Mathematical Proofs by a Computer: A Preparatory Study for a Project Automath». In: *Selected Papers on Automath*. Vol. 133. North-Holland, 1994, pp. 57–72. DOI: 10.1016/S0049-237X(08)70199-X.
- [13] Mario Carneiro. «Conversion of HOL Light proofs into Metamath». In: *arXiv preprint arXiv:1412.8091* (2014).
- [14] Mario Carneiro. «Reimplementing Mizar in Rust». In: *arXiv preprint arXiv:2304.08391* (2023).

- [15] Mario Carneiro. «Lean4lean: Towards a verified typechecker for lean, in lean». In: *arXiv e-prints* (2024), arXiv-2403.
- [16] Véronique Cortier et al. «Machine-checked proofs of privacy for electronic voting protocols». In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 993–1008.
- [17] Judy Crow et al. «A tutorial introduction to PVS». In: Wift. 1995.
- [18] Leonardo De Moura et al. «The lean theorem prover (system description)». In: *International conference on automated deduction*. Springer. 2015, pp. 378–388.
- [19] Roberto Fiorino. «Aristarchus: Design and Implementation of a Verifier for the Sunya Dialect of Hypatia». Tesi di Laurea Triennale. Università degli Studi di Napoli Federico II, 2026.
- [20] Arve Gengelbach, Johannes Åman Pohjola e Tjark Weber. «Mechanisation of model-theoretic conservative extension for HOL with ad-hoc overloading». In: *arXiv preprint arXiv:2101.03807* (2021).
- [21] Georges Gonthier et al. «Formal proof—the four-color theorem». In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.
- [22] Michael J Gordon, Arthur J Milner e Christopher P Wadsworth. *Edinburgh LCF: a mechanised logic of computation*. Springer, 1979.
- [23] Mike Gordon. «From LCF to HOL: A short history.» In: *Proof, language, and interaction*. 2000, pp. 169–186.
- [24] Alexander Granin. *Functional Design and Architecture: Examples in Haskell*. Simon e Schuster, 2024.
- [25] Thomas Hales et al. «A formal proof of the Kepler conjecture». In: *Forum of mathematics, Pi*. Vol. 5. Cambridge University Press. 2017, e2.
- [26] John Harrison. «HOL Light: A tutorial introduction». In: *International Conference on Formal Methods in Computer-Aided Design*. Springer. 1996, pp. 265–269.
- [27] John Harrison. «Towards self-verification of HOL Light». In: *International Joint Conference on Automated Reasoning*. Springer. 2006, pp. 177–191.
- [28] Douglas R Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*. Basic books, 1999.
- [29] Joe Hurd. «The OpenTheory standard theory library». In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 177–191.
- [30] Gerwin Klein et al. «seL4: Formal verification of an OS kernel». In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.
- [31] Laura Kovács e Andrei Voronkov. «First-order theorem proving and Vampire». In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 1–35.
- [32] Amélie Ledein e Elliot Butte. «Traduire l’univers des mathématiques en Dedukti, sans univers». In: *JFLA 2023-34èmes Journées Francophones des Langages Applicatifs*. 2023, pp. 172–189.
- [33] Jan Leike et al. «AI safety gridworlds». In: *arXiv preprint arXiv:1711.09883* (2017).

- [34] Norman Megill e David A Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.
- [35] Leonardo de Moura e Sebastian Ullrich. «The lean 4 theorem prover and programming language». In: *International Conference on Automated Deduction*. Springer. 2021, pp. 625–635.
- [36] César A Muñoz e Ricky Butler. *Rapid prototyping in PVS*. Rapp. tecn. 2003.
- [37] Adam Naumowicz e Artur Korniłowicz. «A brief overview of Mizar». In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 67–72.
- [38] A. Newell e H. Simon. «The logic theory machine—A complex information processing system». In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 61–79. DOI: 10.1109/TIT.1956.1056797.
- [39] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.
- [40] Johannes Åman Pohjola e Arve Gengelbach. «A mechanised semantics for HOL with ad-hoc overloading». In: *arXiv preprint arXiv:2002.10212* (2020).
- [41] John Alan Robinson. «A machine-oriented logic based on the resolution principle». In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.
- [42] Konrad Slind e Michael Norrish. «A brief overview of HOL4». In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, pp. 28–32.
- [43] Morten Rotvold Solberg e Kristian Gjøsteen. «Automated theorem provers and their role in cryptography». In: ().
- [44] Matthieu Sozeau. «Touring the MetaCoq Project». In: *arXiv preprint arXiv:2107.07670* (2021).
- [45] François Thiré. «Sharing a library between proof assistants: Reaching out to the HOL family». In: *arXiv preprint arXiv:1807.01873* (2018).
- [46] Alfred North Whitehead e Bertrand Russell. *Principia mathematica*. Vol. 2. The University Press, 1927.
- [47] Y. Bertot e P. Castéran. *Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions*. 2004. DOI: 10.1007/978-3-662-07964-5.