UNIVERSITÀ DEGLI STUDI DI UDINE

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Tesi di Laurea
in Scienze dell'Informazione

FUNZIONI GENERICHE E MULTIMETODI NELL'AMBITO DEI LINGUAGGI TIPIZZATI

Laureando:
ANTONIO CUNEI

Relatore:
Prof.ssa MARIA STANISZKIS

ANNO ACCADEMICO 1998-99

A Mik, che non ha probabilmente idea
di quanto mi abbia aiutato negli anni,

a Luciano, che sa di avermi aiutato
anche se non ha ben chiaro come,

ed a Gabriele, che sa di avermi aiutato
e come, ma non me l’ha mai fatto pesare.

E, soprattutto, alla mia mamma.

Indice

Elenco delle figure

1. Introduzione

Per sviluppare un progetto software, come per qualunque altra attività, sono necessari strumenti idonei, come pure l'applicazione di tecniche appropriate. Non sorprende, quindi, che innumerevoli, nel corso degli anni, siano stati gli studi finalizzati a creare nuovi strumenti di sviluppo software, ed a introdurre nuove tecniche, che consentissero di semplificare e rendere più razionale l'attività di programmazione.

In particolare, come trattato diffusamente nella disciplina di Ingegneria del Software (vedasi ad esempio [BMP92]), per poter gestire più agevolmente un progetto di grandi dimensioni, è indispensabile poter operare una qualche forma di suddivisione del problema in sottoproblemi più piccoli affrontabili singolarmente; inoltre, tanto minore è l'interdipendenza dei sottoproblemi fra loro, tanto maggiore è la possibilità di affrontare la soluzione delle varie parti separatamente, e quindi di migliorare i tempi di preparazione del progetto (potendo effettuare in parallelo più fasi di sviluppo) e semplificare la gestione complessiva dell'attività di sviluppo software. Di conseguenza, si rivela fondamentale l'individuazione di unità separate, ben distinte fra loro, di cui è definita in modo chiaro e rigoroso l'interfaccia verso l'esterno, ossia la modalità di utilizzo delle diverse parti.

Nell'ambito dei linguaggi di programmazione, l'esigenza di modularizzare i progetti è stata sempre molto sentita e numerosi sono stati gli sforzi compiuti in tale senso durante l'evoluzione che i linguaggi stessi hanno avuto nel corso degli anni. Storicamente, possiamo individuare i primi tentativi in tal senso nelle suddivisioni del codice sorgente in file separati, adottate fin dai primi assemblatori, e nell'adozione di procedure e funzioni chiamabili da più punti, nel quadro delle tecniche di programmazione strutturata. Cruciale è poi stata l'introduzione della compilazione separata, che, riducendo i tempi necessari per la compilazione ed incoraggiando la suddivisione dei programmi in unità separate, è stato un altro passo nella direzione indicata.

In particolare, alcuni linguaggi di programmazione, più di altri, hanno posto l'accento sulla scomposizione del codice sorgente in unità (chiamate, di volta in volta, anche moduli, package o sim.), di cui definire in modo preciso l'interfaccia di utilizzo. Fra questi, è

impossibile non menzionare Ada (vedasi, ad es., [Pyl81], [Geh84]), la versione Borland del Pascal (TurboPascal, [Edi86]), come pure Modula-2 ([Wir85], [Wir]).

Tuttavia, certamente una delle più grosse rivoluzioni nelle tecniche di programmazione è stata l'introduzione della metodologia di programmazione orientata agli oggetti ([Boo91], [PC93], [Bea94]), che ha permesso la creazione di ampie librerie, non solo utilizzabili come in precedenza, ma adattabili e modificabili dall'utente utilizzando le tecniche di eredità tipiche degli oggetti, aggiungendo quindi alla modularità le caratteristiche di versatilità e di riusabilità del codice che hanno subito decretato il successo di tale tecnologia.

Come riassunto in [Cha], i linguaggi object oriented, escludendo quelli basati su agenti, possono essere divisi in due famiglie principali: tipizzati e non tipizzati. Nei linguaggi tipizzati si assume che le variabili appartengano ad un tipo ben preciso, e possano contenere valori solo appartenenti a tale tipo,[1] mentre nei non tipizzati le variabili possono contenere valori arbitrari, ed il tipo effettivo può variare a runtime.

Fra i linguaggi non tipizzati, è d'obbligo citare lo Smalltalk ([Gol84]) ed i derivati dal Lisp, come per esempio CLOS ([Dal97], [Clo]). Fra i tipizzati, che hanno come capostipite Simula, figurano tra gli altri C++ ([Str94]), Java ([Fla97]), Modula-3 ([Wya94], [Mod]) ed Oberon ([Ins], [Obea], [Obeb], [Obec]).

In entrambe le categorie, comunque, figurano linguaggi che fanno degli oggetti il loro unico fondamento (linguaggi orientati agli oggetti "puri", come Smalltalk) e linguaggi che adottano la filosofia ad oggetti "in aggiunta" ad un linguaggio tradizionale, conservando quindi la possibilità di avere variabili che riferiscono dati manipolati dal linguaggio non in veste di oggetto, ma solo come dato passivo di tipo tradizionale. Quest'ultima scelta ha diverse motivazioni: prima di tutto l'estendere un linguaggio già diffuso, in modo da aggiungerne funzionalità mantenendo però le modalità di programmazione cui lo sviluppatore è abituato (C++ e CLOS, per esempio, ma esistono estensioni object oriented per la maggior parte dei linguaggi tradizionali). Altro fattore critico, in certi ambiti addirittura determinante, è l'efficienza in fase di esecuzione. Le implementazioni disponibili di linguaggi che utilizzano esclusivamente oggetti offrono spesso, difatti, performance piuttosto povere, a causa dell'overhead necessario per racchiudere all'interno di oggetti anche i dati appartenenti ai tipi più elementari. Per questo motivo, molti progettisti di linguaggi hanno quindi ritenuto necessario, per raggiungere un'alta efficienza, affiancare agli oggetti perlomeno dei tipi primitivi quali interi, caratteri, numeri floating point, manipolabili al di

[1] Per la precisione, in conformità alla filosofia object oriented, se una variabile viene dichiarata in modo tale che i suoi valori possano essere istanze di una classe, valori legali per la variabile saranno anche istanze di tutte le sottoclassi della classe dichiarata, in quanto una istanza di una sottoclasse "è" anche un dato della classe di partenza.

fuori del paradigma object oriented. Dunque un primo problema si può porre nei seguenti termini: è possibile presentare all'utente del linguaggio un ambiente composto da soli oggetti, coerente e semplice da usare, mantenendo al tempo stesso una elevata velocità di esecuzione del codice finale? Nell'ambito di questo lavoro proporremo una possibile soluzione descrivendo un linguaggio che presenta all'utilizzatore solo oggetti, pur ammettendo implementazioni che consentano di gestire i tipi primitivi con elevata efficienza.

Entrando un po' più nel merito delle scelte che i diversi linguaggi orientati agli oggetti compiono, anche altri problemi meno evidenti tuttavia emergono. Per esempio, l'eredita multipla, strumento tanto potente quanto difficile da gestire, è spesso fonte di numerosi grattacapi: molti linguaggi la bandiscono, ritenendola pericolosa; altri la utilizzano affiancandola a regole di utilizzo complicate e difficili da ricordare. Dunque, un secondo problema al quale sarebbe interessante proporre soluzioni è: come introdurre l'eredità multipla in modo da evitare i problemi che tradizionalmente ad essa si accompagnano (l'ambiguità nella scelta del metodo invocato a run-time, innanzitutto)? Anche questo problema è stato discusso nell'ambito di questa tesi.

Un altro, ben più sottile problema, può derivare, nell'ambito dei linguaggi orientati agli oggetti dotati di tipizzazione statica, dall'utilizzo simultaneo che normalmente viene fatto dell'overloading, risolto staticamente, e della disambiguazione utilizzata dinamicamente per l'individuazione del metodo corretto da invocare a runtime in risposta ad un determinato messaggio; mostreremo difatti con un esempio, scritto in Java, che l'utilizzo di entrambi può portare ad incertezze inaspettate nell'individuazione dei metodi da chiamare durante l'utilizzo degli oggetti.

A tutti i problemi citati si è cercato di trovare risposta tramite soluzioni originali, frutto di adattamenti opportuni di tecniche usate in altri contesti; tali soluzioni hanno poi portato in modo naturale alla progettazione di un linguaggio di programmazione sperimentale, cui è stato dato il nome di BOH (Basic Object Handler).

L'elemento portante di tali soluzioni, e del linguaggio che ne deriva, è un meccanismo di disambiguazione piuttosto atipico per i linguaggi tipizzati, ispirato alle generic functions ed ai multimetodi utilizzati in CLOS.

L'uso di tale tecnica nell'ambito dei linguaggi orientati agli oggetti tipizzati e l'introduzione di alcune semplici regole di utilizzo dei metodi ci consentiranno, come vedremo, non solo di evitare i problemi derivanti dall'overloading, ma di avere uno strumento valido per effettuare, anche con i multimetodi, un type checking interamente statico, evitando ogni possibile violazione di tipo in runtime. A tale meccanismo daremo anche un opportuno supporto formale, con uno studio delle relazioni fra classi e metodi in questo particolare

quadro.

Anche per quanto riguarda l'ereditarietà multipla, verrà introdotta una possibile soluzione, che vedrà applicazione sempre nell'ambito dell'utilizzo dei multimetodi; lo scopo è in questo caso non di introdurre regole di risoluzione per individuare il metodo da invocare, come di solito accade, ma di individuare le circostanze in cui si verificano inconsistenze nell'utilizzo dell'eredità multipla, e di segnalarle al programmatore.

Come vedremo, alcuni semplici accorgimenti permetteranno di presentare all'utilizzatore del linguaggio un ambiente omogeneo ed apparentemente formato esclusivamente da oggetti, evitando quindi di dover gestire differentemente tipi primitivi ed oggetti. Verrà inoltre svincolata l'implementazione dell'ambiente sottostante da quanto viene presentato al programmatore, e verranno proposte alcune implementazioni alternative degli oggetti nel sistema, una delle quali particolarmente efficiente.

Ulteriori considerazioni verranno quindi compiute su alcuni utili strumenti sintattici che verranno introdotti nel linguaggio, al fine di rendere più comoda la programmazione.

Durante tutta la trattazione, lo scopo centrale sarà comunque quello di coniugare quanto più possibile l'eleganza e la semplicità del linguaggio, e l'assistenza all'utente nelle operazioni (omogeneità nella gestione dei dati, verifica e segnalazione delle condizioni di ambiguità, gestione automatica della memoria etc.) con una efficienza di esecuzione paragonabile a quella offerta da linguaggi orientati interamente all'ottenimento di codice efficiente a runtime, quale ad esempio il C++.

Entrando nella struttura della tesi, nel capitolo 2 riprenderemo le definizioni di overloading e richiameremo le definizioni di base legate alla tecnologia object oriented, mostrando le difficoltà nello stabilire con certezza il metodo invocato nei linguaggi a oggetti tipizzati tradizionali, in cui l'overloading viene risolto staticamente. Introdurremo quindi le generic functions ed i multimetodi e mostreremo come questi possano essere adattati al caso dei linguaggi tipizzati, costituendo una soluzione possibile a diversi problemi. Affronteremo quindi i problemi legati all'eredità multipla e concluderemo con delle considerazioni sull'efficienza delle soluzioni implementative adatte ai linguaggi orientati agli oggetti, sia nel caso dei messaggi convenzionali che in quello delle generic functions.

Nel capitolo 3 introdurremo le specifiche di un linguaggio sperimentale, teso a verificare la praticabilità delle tecniche descritte nel capitolo precedente, pur non trascurando le citate considerazioni sull'efficienza.

Nel capitolo 4, infine, illustreremo una implementazione reale di un sottoinsieme piuttosto esteso del linguaggio sperimentale proposto, che ci consentirà di mostrare concreta-

mente come le generic functions ed i multimetodi possano essere utilizzati con successo nell'ambito dei linguaggi tipizzati orientati agli oggetti.

2. Dalle Idee al Design del Linguaggio

2.1. Overloading e messaggi

In questo capitolo verrà presentata una panoramica di alcune delle tecniche utilizzate nei linguaggi object oriented tipizzati di più largo utilizzo, ed in particolare verrà mostrato come la presenza contemporanea di overloading e dispatching dinamico dei messaggi possa portare, in tali linguaggi, a situazioni difficili da prevedere.

Precisiamo comunque che, nel seguito, identificheremo fra loro i concetti di tipo di una istanza e di classe di appartenenza della medesima, ossia considereremo equivalenti i termini "tipo" e "classe", in conformità all'utilizzo effettuato in [Boo91, pag. 59], cui si può fare riferimento per ulteriori chiarimenti in proposito. Inoltre utilizzeremo per brevità la notazione "$fun(C_1, C_2, \ldots, C_n) : C$", e parleremo di funzioni di parametri $C_1, \ldots, C_n$, riferendoci in effetti a funzioni con parametri di tipo $C_1, \ldots, C_n$ che restituiscono valori di tipo C. Tale notazione riprende quella usata per i prototipi di funzione in ANSI C ([KR89]).

2.1.1. Overloading

Una tecnica comunemente adoperata nei linguaggi di programmazione di tipo tradizionale, al fine di semplificare l'utilizzo di operazioni concettualmente simili fra loro, è quella dell'"overloading". Con tale termine si intende la possibilità di utilizzare il medesimo identificatore per riferire più funzioni o procedure di tipo tradizionale, che differiscano fra loro solo la successione dei tipi dei parametri, ossia per il dominio.

Per fare un esempio immediato, in molti linguaggi il simbolo "+" è utilizzato per denotare sia la somma fra due interi che quella fra due numeri in virgola mobile, ovverosia si utilizza sempre il medesimo simbolo per indicare operazioni implementate in realtà in modo molto diverso. In questo senso, il simbolo "+" è "sovraccaricato" (overloaded) con più significati effettivi, rappresentando operazioni diverse qualora utilizzato con argomenti di tipo diverso.

L'individuazione di quale, fra le funzioni disponibili, vada utilizzata in un determinato

caso, viene effettuata in questo ambito in modo interamente statico, durante la compilazione, cercando fra le funzioni definite con quell'identificatore una il cui dominio coincida esattamente con quello della chiamata in esame.

Questo modo di operare, utilizzato nei linguaggi di tipo tradizionale, è stato mantenuto quasi inalterato anche con l'avvento della programmazione object-oriented, e viene ritrovato infatti nella gran parte dei linguaggi tipizzati orientati agli oggetti, quali C++, Java ed altri.

Vedremo tuttavia nei prossimi capitoli come gli effetti dell'overloading, in tale contesto, possano rivelarsi molto meno facili da gestire di quanto potrebbe apparire ad un primo esame.

2.1.2. Messaggi ed oggetti

Non effettueremo qui una trattazione completa della programmazione orientata agli oggetti, per la quale rimandiamo, per esempio, a [SB86] e [Boo91]; ne ripercorreremo tuttavia in sintesi le idee principali, allo scopo di rendere più chiare le discussioni successive.

L'idea portante della metodologia è quella di raggruppare in un'unica struttura dati ed operazioni, modellando un oggetto dotato di comportamento oltre che di stato interno. Lo scopo è quello di ottenere una entità in grado di rispondere autonomamente a messaggi inviati dall'esterno, reagendo a proprio modo sulla base delle operazioni in esso definite.

Formulando un esempio, se modelliamo come oggetto una stampante, potremo inviare a tale oggetto un testo da stampare, ed ottenere lo scopo voluto senza alcuna necessità di conoscere l'implementazione interna della procedura di stampa.

Ancora meglio, se modelliamo allo stesso modo tre stampanti di tipo molto diverso, potremo ottenere da ciascuna l'esecuzione della relativa procedura di stampa utilizzando *lo stesso* messaggio, anche se le implementazioni interne sono molto diverse.

I vantaggi sono evidenti: si incapsulano dati ed operazioni su tali dati in una unica entità, disaccoppiando l'interfaccia di utilizzo dell'oggetto (costituita dai messaggi accettati) dall'implementazione interna, aumentando quindi la modularità del progetto software risultante.

Le implementazioni interne, invocate all'arrivo di un messaggio, sono chiamate "metodi", in quanto rappresentano appunto il metodo secondo cui l'oggetto reagisce ad un determinato stimolo.

In molti linguaggi viene poi utilizzato il concetto di "classe" per indicare la definizione di un gruppo di oggetti di struttura e comportamento identici. Tali oggetti verranno definiti "istanze" della classe.

Per esempio, della classe delle stampanti modello “InkJet100” potranno far parte la stampante “Uno” (contenente 10 fogli, in stampa), la stampante “Due” (senza carta, non disponibile) e così via. Ogni istanza di stampante avrà il suo stato interno distinto da quello delle altre, ma tutte avranno la stessa sequenza di variabili interne (dette “variabili di istanza”), e si comporteranno nel medesimo modo (e avranno quindi in comune lo stesso insieme di metodi).

Inoltre le classi vengono di norma inquadrate in una gerarchia, facendo discendere classi più specifiche (es: “stampanti_InkJet100”), da altre più generiche (es: “stampanti”). L’idea è quella di estendere le definizioni più generiche arricchendone la struttura con eventuali nuove variabili di istanza e metodi. Se una istanza di una classe non è in grado di rispondere ad un messaggio con un metodo proprio, verrà fatto ricorso al primo metodo applicabile, risalendo la gerarchia, trovato in una sopraclasse della classe di partenza. Se una stampante InkJet100 non ha un suo metodo specifico per cambiare il colore di stampa, si può fare ricorso ad un metodo che vada bene per tutte le stampanti (es: ignorare il cambiamento di colore).

La relazione considerata fra le classi è quella “is-a”, ossia una istanza di una sottoclasse “è” anche una istanza di una sopraclasse, rappresentandone una estensione o modifica: una stampante “InkJet100” è tutto sommato una stampante, e quindi deve accettare anche gli stessi messaggi inviabili ad una stampante generica.

2.1.3. Linguaggi tipizzati

Nei linguaggi tipizzati ad oggetti, le idee generali rimangono le medesime, pur con le dovute cautele, ed anche l’overloading trova, come menzionato, il suo adattamento.

Se il linguaggio ad oggetti che consideriamo è tipizzato, ad ogni variabile dovremo assegnare un tipo, in modo da poter effettuare dei controlli di legalità durante la compilazione. In particolare, se per esempio indichiamo che una variabile è di tipo “stampanti”, intenderemo che la variabile potrà riferire solamente oggetti che siano stampanti.

Nel caso degli oggetti, tuttavia, come abbiamo visto, anche le istanze di sottoclassi delle stampanti “sono” stampanti, ed è quindi norma, in tale categoria di linguaggi, ammettere che una variabile che riferisce oggetti di una determinata classe possa riferire anche istanze di una qualunque sottoclasse della classe indicata. Dal momento poi che queste sono da considerarsi estensioni derivanti dalla classe di partenza, anche le istanze delle sottoclassi accetteranno sicuramente i messaggi accettati dalle istanze della classe originale.

Per chiarire, di ogni classe noi conosciamo l’interfaccia, e quindi l’elenco dei messaggi accettabili; possiamo quindi verificare staticamente se una istanza di quella classe (o di una

discendente) si trova nella posizione di poter ricevere un messaggio illegale. Di converso, se il messaggio appartiene all'interfaccia della classe, sappiamo che esisterà almeno un metodo idoneo definito.

Per via della possibilità che abbiamo di definire metodi più specifici nelle sottoclassi, coprendo i metodi più generici delle sopraclassi, non avremo però in generale modo di stabilire staticamente anche quale, fra i metodi disponibili, sarà quello da invocarsi a runtime. La scelta, difatti, potrà essere operata soltanto in fase di esecuzione, ossia quando sapremo di preciso a quale classe effettiva (sottoclasse o uguale alla classe considerata staticamente) apparterrà l'istanza a cui stiamo inviando il messaggio.[1]

Per questo motivo, nei linguaggi a oggetti tipizzati è solitamente presente una qualche forma di dispatching dei messaggi a runtime, con il quale viene ricercato dinamicamente, durante l'esecuzione, il metodo più specifico utilizzabile in risposta ad un messaggio inviato ad un oggetto.

Per quanto riguarda l'overloading, la situazione è simile a quella dei linguaggi tradizionali, ma tenendo stavolta in considerazione anche l'esistenza della gerarchia delle classi. Laddove, infatti, in un linguaggio sprovvisto di oggetti, l'unica funzione che può essere scelta deve avere l'elenco dei tipi dei parametri perfettamente coincidente con quello degli argomenti della chiamata, in questo nuovo caso le classi dei parametri della chiamata potranno anche essere sottoclassi di quelli di una funzione accettabile. Come risultato, può benissimo verificarsi il caso in cui due o più funzioni corrispondenti allo stesso identificatore sovraccaricato siano simultaneamente utilizzabili per una specifica chiamata.

Per esempio, se nella gerarchia compaiono una classe A ed una classe B figlia di A, ed abbiamo le definizioni delle due funzioni fun(A) e fun(B), con overloading, una chiamata fun(B) nel sorgente sarebbe adatta ad entrambe. La scelta che in questa categoria di linguaggi viene compiuta, come vedremo fra poco nel caso di Java, è quella di preferire la funzione più "specifica", seguendo la gerarchia; perciò, nel caso mostrato, verrebbe scelto fun(B). Si tratta di una estensione abbastanza diretta del meccanismo di overloading tradizionale, ma nasconde, come subito vedremo, delle pericolose insidie nel meccanismo utilizzato per la disambiguazione, che è e rimane anche in questo caso interamente statico.

2.1.4. Overloading vs. dispatching

Vediamo dunque un esempio che mostra come la presenza contemporanea di dispatching dinamico ed overloading risolto staticamente possa portare a comportamenti inaspettati nel

[1] Il C++, a questo riguardo, opera scelte un po' particolari, dettate da motivi di ricerca della massima velocità a runtime. Utilizzando la keyword "virtual", comunque, il comportamento è quello qui descritto ([Eck]).

funzionamento del sistema.

Supponiamo di voler implementare due classi, una classe padre ed una classe figlia, utilizzando Java: (per dettagli sulla sintassi e l'uso di Java si può fare riferimento a [Fla97])

```
class figlia extends padre {}

class padre
{
 void direct(padre b) {
  System.out.println(" Invocato padre - padre");
 }

 void direct(figlia b) {
  System.out.println(" Invocato padre - figlia");
 }

 void indirect(padre b) {
  direct(b);
 }

 public static void main(String av[])
 {
  padre p=new padre();
  figlia f=new figlia();

  System.out.print(p); System.out.print(p); p.direct(p);
  System.out.print(p); System.out.print(f); p.direct(f);
  System.out.print(p); System.out.print(f); p.indirect(f);
 }
}
```

Come si vede, vengono definiti due messaggi, `direct` ed `indirect`, che possono essere inviati ad istanze della classe padre. Questo è l'output del programma:

```
padre@80caf4a padre@80caf4a Invocato padre - padre
padre@80caf4a figlia@80caf4c Invocato padre - figlia
padre@80caf4a figlia@80caf4c Invocato padre - padre
```

Come si può vedere, la prima e la seconda invocazione si comportano come ci si potrebbe aspettare, chiamando correttamente il metodo più adatto alla particolare invocazione; grazie all'overloading, dato che figlia è più specifico di padre, viene scelto il metodo più specifico applicabile al caso in questione.

Nel terzo caso, però, l'overloading non entra in gioco e viene invocato in ogni caso `indirect(padre)`. All'interno di `indirect` il parametro figura come padre, e l'overloading, staticamente, risolve sempre il primo metodo anzichè il secondo.

In pratica, durante il passaggio attraverso `indirect` l'istanza f perde in qualche modo parte della propria identità: nella risoluzione tramite overloading l'oggetto viene "declassato" ad una classe più generica. Inaspettatamente, quindi, viene invocato un metodo più generico di quello previsto, che può avere una implementazione anche molto diversa. È evidente che tenere sotto controllo questo genere di comportamenti può risultare difficoltoso per l'utente, e portare a comportamenti difficili da prevedere.

2.2. Introduzione alle generic functions

Al fine di ovviare ai problemi derivanti dall'overloading, utilizzeremo nei capitoli successivi uno strumento alternativo, usato a volte in linguaggi non tipizzati. Tramite una analisi formale della situazione, riusciremo ad adattarlo anche al caso dei linguaggi tipizzati, ottenendone allo stesso tempo sia una valida alternativa alla tradizionale risoluzione statica dell'overloading che una risorsa utile per affrontare i problemi derivanti dall'eredità multipla.

2.2.1. Un approccio diverso: generic functions e multimetodi

Per affrontare il problema, proporremo, nel corso di questa discussione, una soluzione alternativa, adattando al caso dei linguaggi tipizzati una tecnica utilizzata nell'ambito del Common Lisp: le generic functions.

Come abbiamo illustrato precedentemente, nella programmazione object oriented il modello utilizzato è quello di oggetti in grado di rispondere autonomamente a messaggi provenienti dall'esterno. Ora, sebbene questa sia una astrazione di indubbia eleganza e praticità, non sempre, se usata strettamente in questi termini, permette di modellare la realtà che si intende descrivere con accuratezza.

Il forzare i metodi a descrivere il comportamento di oggetti singoli, infatti, può portare, in alcuni casi, ad estremizzazioni un po' discutibili. Pensiamo, per esempio, al caso dello Smalltalk, in cui tutti i dati vengono rappresentati come oggetti: per effettuare una somma fra interi, l'unico modo è quello di inviare al primo addendo un messaggio che abbia come parametro il secondo addendo. Dal punto di vista funzionale non vi è nulla da obiettare; può tuttavia risultare un po' forzata l'asimmetria nell'utilizzo dei due interi, uno usato

come destinatario e l'altro come semplice parametro, laddove la somma risulta in effetti da una interazione paritetica fra i due.

Allo scopo di meglio rappresentare tali situazioni, il CLOS (Common Lisp Object System, [Dal97], [Clo], [Gra96]) fornisce degli strumenti alternativi per la spedizione di messaggi: le "generic functions" ed i "multimetodi".

L'idea è quella di raggruppare funzioni che descrivono il comportamento di n-ple di oggetti di classi diverse in una unica funzione generica, invocabile con parametri di più tipi. Diversamente da quanto avviene con l'overloading, però, in questo caso la disambiguazione, e quindi l'individuazione della funzione da chiamare, viene effettuata in modo interamente dinamico, basandosi sulla classe di appartenenza a runtime di tutti i parametri anzichè su quella di un singolo destinatario .

Le funzioni utilizzate a questo scopo possono perciò essere considerate dei "multimetodi", in quanto definiscono il comportamento di una n-pla di oggetti in risposta ad un determinato messaggio, in modo affine a quanto avviene per un metodo ordinario nel caso unario; le generic functions che ne si ottengono sono delle sorte di "multimessaggi", rappresentando messaggi inviabili ad n-ple di oggetti nel loro complesso.[2]

Analogamente a quanto avviene nel caso unario, anche con le generic functions la disambiguazione del messaggio sarà finalizzata all'individuazione del multimetodo "più specifico" applicabile a quella combinazione di parametri. È abbastanza evidente, però, che il fatto di dover tenere in considerazione simultaneamente più parametri complica abbastanza la ricerca, ed in particolare si rende necessario prima di tutto puntualizzare cosa si intenda con "più specifico" nel caso di n-ple di classi. È inoltre da chiarire come i multimetodi possano essere impiegati nell'ambito di un linguaggio tipizzato, utilizzando tecniche simili a quelle illustrate precedentemente per i metodi semplici al fine di effettuare un type checking statico.

Nei capitoli che seguiranno tratteremo tutti questi argomenti, mostrando non solo come i multimetodi possano trovare una loro naturale applicazione anche nei linguaggi tipizzati, ma anche come il loro utilizzo consenta di ovviare ai problemi che abbiamo visto essere legati al comportamento statico dell'overloading. Troveremo poi che il medesimo strumento consente inoltre di affrontare i problemi derivanti dall'eredità multipla.

[2] In realtà le generic functions di CLOS sono un po' più complesse, prevedendo metodi "before" ed "after"; noi le utilizzeremo solamente nell'accezione indicata di messaggi applicabili a n-ple di oggetti.

2.2.2. Un primo esempio di utilizzo

Per osservare subito quale sia il nuovo comportamento del linguaggio nel caso delle generic functions, riprendiamo l'esempio precedente e proviamone una versione che utilizza i multimetodi.

L'esempio che segue è stato scritto nel linguaggio che verrà definito più avanti in dettaglio; per ora, anche se non ne abbiamo introdotto ancora la sintassi, la similitudine con l'esempio precedente e la descrizione delle generic functions dovrebbero rendere il codice di facile comprensione.

```
GenericVsOverload:  uses system
{
!padre:  super object {!padre():  super object() {}}
!figlia:  super padre {!figlia():  super padre() {}}

direct(a:padre,b:padre)
{ println(" Invocato padre - padre"); }

direct(a:padre,b:figlia)
{ println(" Invocato padre - figlia"); }

indirect(a:padre,b:padre)
{ direct(a,b); }

main()
{
 p:=padre();
 f:=figlia();

 p.print; p.print; p.direct(p);
 p.print; f.print; p.direct(f);
 p.print; f.print; p.indirect(f);
}
}
```

L'output del programma, compilato con il compilatore sperimentale descritto più avanti, risulta essere il seguente:

```
<padre><padre> Invocato padre - padre
<padre><figlia> Invocato padre - figlia
<padre><figlia> Invocato padre - figlia
```

Come si può notare, il comportamento è alquanto diverso: l'invocazione finale di `direct` è la medesima nel secondo e nel terzo caso: al contrario del caso dell'overloading, la risoluzione dinamica su tutti i parametri consente di mantenere fino alla chiamata finale di `direct` l'informazione secondo la quale f è istanza della classe figlia. Il comportamento è più omogeneo, e non si riscontrano i problemi rilevati nel caso dell'overloading risolto staticamente.

In base a questo esempio, non sembra azzardato affermare che, in un linguaggio object-oriented tipizzato, le generic functions sono preferibili rispetto all'utilizzo classico dell'overloading, in quanto permettono di mantenere più informazione riguardo alla natura degli oggetti trattati, e sembrerebbero quindi, curiosamente, più rispondenti alla metodologia object oriented della restrizione dei messaggi ad un unico destinatario.

Il problema, ovviamente, non si pone in un linguaggio non tipizzato come Smalltalk, in cui il meccanismo di overloading non è comunque presente, ed è possibile definire un solo metodo per classe corrispondente ad un dato messaggio.

2.3. Generic functions e multimetodi nei linguaggi tipizzati

Come abbiamo visto nel capitolo 2.1.3, utilizzando i messaggi ed i metodi nell'ambito di un linguaggio tipizzato, è possibile effettuare staticamente un controllo statico di legalità sull'applicabilità di messaggi ad oggetti di una certa classe (o sottoclassi), anche se la determinazione effettiva del metodo da chiamare potrà essere effettuata, in generale, solo a runtime. In pratica, siamo in grado di garantirci staticamente contro la presenza di errori di tipo in fase di esecuzione.

Avendo ora introdotto una estensione ad n-ple di oggetti per messaggi e metodi, il nostro scopo è quello di introdurre una metodica che ci garantisca, anche in questo caso, di poter raggiungere il medesimo risultato, ossia di poter effettuare un type checking interamente statico.

2.3.1. Il grafo dei multimetodi

Come strumento iniziale, avremo bisogno di un modo per stabilire quando un metodo è "più specifico" di un'altro, e dunque preferibile durante la disambiguazione; dovremo inoltre precisare con chiarezza qual'è l'ambiente entro il quale operiamo per effettuare le ricerche.

Per cominciare, notiamo che, nel caso dei messaggi unari, l'esplorazione avviene nel grafo delle classi considerando i metodi che potrebbero essere definiti in ciascuna di queste. Dato un messaggio potremo avere di fatto una o nessuna definizione di metodo in ogni classe; se una classe B è sottoclasse di A, il metodo definito per B sarà "più specifico" del metodo definito su A.

In effetti, potremmo pensare di avere utilizzato due grafi, di cui uno costituito da classi e dalla relazione "è una sottoclasse immediata di", e l'altro costituito da definizioni di metodi e dalla relazione "più specifico di". Se consideriamo il caso in cui, dato un messaggio, in ogni classe viene definito un metodo, i due grafi risultano isomorfi.

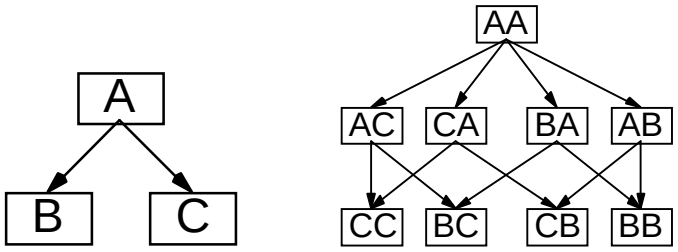
Passando ad analizzare i messaggi di arità maggiore di uno, avremo che per ogni messaggio, i metodi diversi costruibili saranno uno per ogni n-pla di classi; data una generic function, dotata di n parametri, l'insieme di supporto della relazione di maggiore specificità fra i metodi sarà dunque costituito da tutte le possibili n-ple di classi.

Per quanto riguarda la relazione, possiamo formalizzare il concetto intuitivo di metodo più specifico nel seguente modo:

- Date due n-ple di classi $\widetilde{A} = (A_1, A_2, \ldots, A_n)$ e $\widetilde{B} = (B_1, B_2, \ldots, B_n)$, allora la coppia $(\widetilde{A}, \widetilde{B})$ appartiene alla relazione SE esiste un j fra 1 ed n tale che per ogni $i = 1, \ldots, n$,

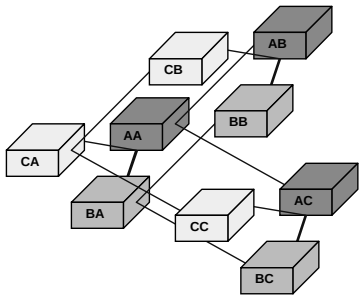
con $i \neq j$, A_i è uguale ad B_i, e si ha che A_j è sottoclasse diretta di B_j, ossia (A_j, B_j) appartiene alla relazione gerarchica fra le classi "è sottoclasse immediata di".

Questa relazione di “maggiore specificità diretta” induce una relazione d’ordine parziale sulle n-ple di classi, in base alla quale è possibile indicare se una n-pla è più specifica di un’altra, direttamente o indirettamente (passando attraverso n-ple intermedie), esattamente allo stesso modo in cui è possibile indicare una classe come sottoclasse diretta o indiretta di un’altra, utilizzando l’ordine parziale dato dalla relazione gerarchica diretta. La coppia



(a) Classi singole

(b) Coppie di classi



(c) Rappresentazione tridimensionale

Figura 2.1.: Estensione del grafo delle classi: eredità semplice

(n-ple di classi, relazione di specificità diretta) sarà assai importante nel seguito della nostra trattazione, e tornerà nuovamente in gioco quando tratteremo l’implementazione del dispatcher. In un certo senso, si tratta di una estensione polidimensionale del grafo originale, come se pensassimo di disporre un grafo delle classi lungo ognuna delle n dimensioni, ricavassimo tutti i punti delle n-ple nello spazio che ne si ottiene, e poi collegassimo tali punti con la nostra nuova relazione.

In Figura 2.1 è mostrato il risultato del procedimento di costruzione; è anche presente una raffigurazione tridimensionale che può forse consentire di visualizzare meglio la situazione.

Notiamo subito che, se il grafo originale è connesso, anche il grafo derivato lo sarà, e che quest'ultimo non avrà cicli se quello di partenza non ne aveva (tali proprietà si dimostrano facilmente, la prima considerando che ogni n-pla è collegata ad almeno un'altra, la seconda per assurdo).

Un'altra considerazione che possiamo effettuare è che, anche per i grafi delle classi più semplici, nel grafo derivato esistono cammini multipli che collegano due n-ple di classi. Come vedremo, questo può portare ad ambiguità affini a quelle che possono verificarsi nei linguaggi tradizionali con l'utilizzo dell'eredità multipla; di converso, vedremo come una analisi accorta della situazione ci consentirà di riconoscere e risolvere tali situazioni in modo assai semplice, sia nel caso dell'eredità singola che di quella multipla.

2.3.2. Tipi statici e dinamici

Come abbiamo detto, noi desideriamo avere la possibilità di effettuare un type checking statico. Nei linguaggi object oriented tradizionali, questo si riduce a verificare quale è il tipo dichiarato staticamente per una variabile, e a controllare che le istanze della classe dichiarata (e di conseguenza anche delle sottoclassi) possano accettare i messaggi inviati. Questo rispecchia in modo fedele il principio, già menzionato, secondo cui una istanza di una sottoclasse "è" anche un oggetto della sopraclasse, e dunque ne accetta anche gli stessi messaggi, rappresentandone una specializzazione o variazione.

Questo utilizzo ci porta in modo molto naturale all'attribuzione ad ogni variabile di due tipi distinti: un tipo statico, che corrisponderà alla classe specificata nella dichiarazione, ed un tipo dinamico, che corrisponderà alla classe effettiva di appartenenza dell'oggetto che la variabile riferirà a runtime.

Per formulare un esempio concreto, se una variabile è di tipo "tree", la variabile potrà riferire una qualsiasi istanza di "tree" o di una sua sottoclasse; il tipo statico sarà "tree", mentre il tipo dinamico potrà essere qualunque, e variare a runtime, a condizione che si tratti di una sottoclasse di "tree". In questo senso, il tipo statico identificherà il tipo "più generico" che tale variabile potrà avere in fase di esecuzione.

Come illustrato nel capitolo 2.1.3, per ogni utilizzo di un messaggio, potremo utilizzare il tipo statico del destinatario per verificare staticamente la validità dell'invocazione a runtime: se esiste un metodo in grado di gestire il messaggio per le istanze di classe corrispondente al tipo statico, saremo sicuri che, in esecuzione, esisterà almeno un metodo

valido per ogni destinatario, visto che il tipo dinamico dovrà essere sottoclasse o uguale di quella del tipo statico.

Estendendo opportunamente la tecnica, utilizzando la relazione di maggiore specificità sulle n-ple che abbiamo introdotto precedentemente, potremo effettuare analogamente un type checking statico anche con i multimetodi, verificando che esista, in corrispondenza di una invocazione, almeno un multimetodo adatto per la n-pla dei tipi statici dei parametri attuali (ossia definito per una n-pla che sia più generica di quella dei tipi statici). Dato che, in runtime, i tipi dinamici dei parametri attuali saranno sottoclassi o uguali ai relativi tipi statici, sapremo sicuramente che esisterà sempre almeno un multimetodo idoneo alle circostanze (anche se, in generale, ne potrà esistere più di uno).

Per fare un esempio concreto, supponiamo di avere una classe `A` ed una sua sottoclasse `B`, e di avere i seguenti multimetodi:

```
fun(B,A)
fun(B,B)
```

In tale caso, se abbiamo una invocazione che appare staticamente come `fun(B,A)`, allora sapremo certamente che, in runtime, al messaggio corrisponderà almeno un metodo per qualunque combinazione di parametri, i cui tipi dinamici dovranno essere sottoclassi di B ed A rispettivamente. Viceversa, se abbiamo una invocazione che vediamo staticamente come `fun(A,B)`, non potremo avere la garanzia che un metodo adatto esisterà a runtime, e dunque informeremo staticamente l'utente della possibilità di errore. Si tratta chiaramente di una semplice estensione di quanto viene normalmente compiuto nei linguaggi object oriented tipizzati per verificare la legalità dell'invio di messaggi nel caso di destinatari singoli.

I vantaggi portati dalla tecnica descritta sono essenzialmente la possibiltà di utilizzare i multimetodi pur preservando i vantaggi tipici della tipizzazione (verifica della coerenza degli utilizzi, assenza di errori di tipo in esecuzione).

Il dispatching effettivo verrà comunque compiuto a runtime, ma avremo sempre e comunque la garanzia dell'esistenza di almeno un metodo invocabile. È da notare che il fatto di effettuare il dispatching dei multimetodi a runtime non implica necessariamente che l'implementazione debba essere inefficiente; delle tecniche per realizzare efficientemente un dispatcher adatto verranno difatti presentate più avanti.

2.3.3. Risolvere le condizioni di ambiguità

Con la tecnica che abbiamo discusso, abbiamo ottenuto la capacità di verificare l'esistenza di almeno un metodo valido a runtime in corrispondenza di una determinata invocazione. Tuttavia, non possiamo ancora avere la garanzia che il metodo più specifico applicabile sia unico.

Per esempio, se abbiamo ancora la classe A e la sua sottoclasse B, ed abbiamo i due metodi:

```
fun(B,A)
fun(A,B)
```

nel caso avessimo una chiamata individuabile staticamente come `fun(B,B)`, entrambi i metodi sarebbero applicabili. Si tratta chiaramente di una condizione di ambiguità nella definizione; per fare un esempio, sarebbe come se avessimo due ricette per preparare torte, di cui una da usare per preparare torte con frutta generica e crema pasticcera, ed un'altra per preparare torte con fragole ed una crema generica. Se volessimo preparare una torta con sia con fragole che con crema pasticcera, non sapremmo quale delle due ricette scegliere; si renderebbe necessario quindi descrivere esplicitamente quale sia il comportamento da tenersi in questo particolare caso.

Più formalmente, abbiamo una condizione di ambiguità ogni volta che, dato un messaggio, non siamo in grado di individuare in ogni situazione un unico metodo da invocare, ossia che esiste una n-pla, per cui non è definito un multimetodo, che è simultaneamente più specifica di due o più n-ple scorrelate, ciascuna con un multimetodo corrispondente. La condizione di ambiguità si estende, naturalmente, anche a tutte le n-ple discendenti da quella dove si è verificato il conflitto.

Per risolvere l'ambiguità, perciò, è sufficiente individuare la n-pla "più generale" dove si verifica ed imporre, con una opportuna regola, che esista anche una definizione di metodo corrispondente a tale n-pla, in modo da eliminare il conflitto.

In assenza di eredità multipla, possiamo infatti verificare facilmente che tale n-pla, per ogni insieme di metodi in conflitto, esiste ed è unica, ed è costituita, per ogni indice i, dall'ultima sottoclasse, rispetto alla relazione di gerarchia fra le classi, dell'insieme $A_i = \{C_{j,i}\}$, con $C_{j,i}$ classe del parametro formale di posizione i del j-esimo metodo. Tale insieme deve essere infatti, e non è difficile convincersene, totalmente ordinato se i metodi in esame sono in conflitto.

Ne deriva perciò la seguente regola che, se rispettata, rende impossibile il verificarsi di condizioni di ambiguità in caso di eredità semplice:

- Per ogni sottoinsieme dell'insieme di metodi corrispondenti ad un messaggio, quindi con uguale identificatore ed arità, i cui parametri siano rispettivamente $(C_{j,1}, C_{j,2}, \ldots, C_{j,n})$, SE per ogni $i = 1, \ldots, n$, l'insieme $\{C_{j,i}\}$ è totalmente ordinato rispetto alla relazione gerarchica fra le classi, ALLORA deve essere definito anche un metodo per lo stesso messaggio sulla n-pla $(\widetilde{C_1}, \widetilde{C_2}, \ldots, \widetilde{C_n})$, dove ogni $\widetilde{C_j}$ è l'ultima sottoclasse di $\{C_{j,i}\}$.

Se tale condizione è rispettata, saremo quindi sicuri di poter individuare, in corrispondenza di ogni messaggio (funzione generica), uno ed un solo metodo da invocare in runtime, e di avere assenza di condizioni ambigue.[3]

Le stesse considerazioni, inoltre, possono essere applicate anche al caso statico. Se infatti è verificata la condizione descritta, anche durante la ricerca effettuata tramite i tipi statici sarà possibile individuare uno ed un solo metodo più specifico corrispondente, da utilizzare per il type checking statico. Questo si rivelerà particolarmente utile nel prossimo capitolo, in cui estenderemo le considerazioni su tipi statici e dinamici ad espressioni generiche.

2.3.4. Valori di ritorno ed espressioni

Come abbiamo visto, tramite la regola del capitolo 2.3.3 siamo in grado di individuare anche staticamente un unico metodo sulla base del quale effettuare un type checking statico. A questo punto, però, sorge naturale chiedersi cosa possiamo dire del valore di ritorno, ovvero se possiamo anche in questo caso ottenere delle informazioni sul suo tipo statico. A seconda del metodo individuato, infatti, potremmo avere in generale indicazioni diverse e non necessariamente coerenti sul tipo restituito. Questo ci impedirebbe di individuare un unico tipo statico, ossia una unica limitazione del tipo del valore di ritorno, e quindi procedere induttivamente nel type checking di espressioni complesse, contenenti funzioni annidate.

Il problema può essere tuttavia risolto facilmente tramite l'imposizione della seguente regola:

- Se due metodi hanno uguale identificatore ed arità, e quindi corrispondono al medesimo messaggio, allora le definizioni devono essere coerenti, ovverosia: SE per

[3] Un controllo simile viene effettuato, seppure nel caso dell'overloading, anche da Java e dal C++: le implementazioni da noi provate riconoscono infatti correttamente l'ambiguità anche se non sono in grado di proporre soluzioni, cosa invece possibile utilizzando la regola qui esposta. Si vedano i sorgenti dei programmi di test in appendice.

> ogni coppia di metodi $fun(A_1, A_2, \ldots, A_n) : A$ e $fun(B_1, B_2, \ldots, B_n) : B$, e per ogni $i = 1, \ldots, n$ si ha che A_i è sottoclasse o uguale a B_i, ALLORA deve essere A sottoclasse o uguale a B.

Tale regola è, in fin dei conti, rispondente all'intuizione: se un messaggio che lavora su A_1 e A_2 restituisce A, allora tutti i metodi corrispondenti devono funzionare nello stesso modo. Dal momento che, seguendo la filosofia object oriented, ogni istanza di B_1 "è" anche una istanza di A_1, e similmente per gli altri parametri, tutti i metodi devono restituire una istanza di tipo A (od una sottoclasse) per tutti i parametri di tipo $A_1, \ldots, A_n$ (o rispettive sottoclassi). Ad esempio, se abbiamo un messaggio prepara_torta(frutta):torta, è ragionevole aspettarsi che tutti i metodi relativi, applicati a frutta, restituiscano torte, e saranno dunque coerenti i metodi prepara_torta(mele):torta_di_mele, prepara_torta(pere):torta_di_pere e così via, dove mele e pere sono sottoclassi di frutta, e torta_di_mele e torta_di_pere sono sottoclassi di torta.

Una conseguenza immediata è che se due metodi corrispondenti ad uno stesso messaggio avessero uguale elenco dei parametri, l'eventuale tipo di ritorno dovrebbe essere coincidente. Per poter definire il comportamento di una funzione generica in modo univoco, e poter quindi scegliere in tale circostanza un unico metodo, imponiamo quindi, abbastanza ragionevolmente, che possa esistere solo un metodo per messaggio che accetti una fissata sequenza di tipi statici dei parametri.

In virtù della regola enunciata, possiamo utilizzare sicuramente il tipo del valore di ritorno del metodo individuato staticamente come una limitazione di tutti i valori di ritorno che possono essere restituiti da metodi più specifici a runtime; potremo quindi utilizzarlo per attribuire un tipo statico, oltre che alle variabili, anche alle invocazioni di generic functions nel loro complesso e, induttivamente, ad espressioni comunque complesse.

Inoltre, avremo la garanzia che qualsiasi estensione successiva della gerarchia delle classi dovrà mantenere il tipo più generico di ritorno così individuato inalterato, preservando la validità del codice scritto sino a quel momento.

Per effettuare un confronto con altri linguaggi, nel caso di Java un messaggio può avere soltanto uno ed un tipo di ritorno per tutti i metodi definiti, qualunque sia la classe relativa. Questa, tuttavia, è una restrizione piuttosto vincolante. Per esempio, supponiamo di voler modellare come oggetto una lista generica tramite la classe "list", e di voler definire il messaggio "flip" che restituisca la lista fornita come parametro ma con gli elementi in ordine inverso (lasciando la lista di partenza inalterata).

Se ora vogliamo definire una lista di interi, "list_int", sottoclasse di "list" ed estendere il messaggio con un nuovo metodo, con Java il tipo restituito dovrà essere sempre generi-

camente “list”, anche se noi restituiremo in effetti liste di interi. Procedendo in tale modo, perdiamo parte della comodità di effettuare un type checking statico, perchè non avremo modo di verificare completamente il corretto utilizzo del valore restituito, e non potremo utilizzare su di esso i messaggi specifici di “list_int”.

Viceversa, utilizzando la tecnica che abbiamo descritto,è possibile estendere il messaggio “flip” tramite un metodo che restituisca, da “list_int”, un valore di ritorno di tipo “list_int”, e avremo staticamente la possibilità di effettuare controlli completi sul tipo restituito.

Un esempio concreto di questo utilizzo flessibile, e tuttavia rigoroso, del valore di ritorno, è illustrato in appendice, contrapposto a Java che, nelle stesse circostanze, si arresta con un errore di compilazione. Il compilatore sperimentale che introdurremo più avanti si basa infatti sulle modalità di gestione dei tipi che abbiamo qui proposto.

2.4. Eredità multipla

L'ereditarietà multipla è, senza dubbio, la bestia nera di tutti i progettisti di linguaggi. Molti ritengono che adottarla significhi inevitabilmente andare incontro ad una serie inenarrabile di guai e problemi; altri dicono che introdurre l'eredità multipla sia come "aprire un barattolo pieno di vermi" [Mar93, pag. 77], per via delle complicazioni che introduce. In diversi linguaggi (Smalltalk, Java) è semplicemente bandita; in altri (C++, CLOS) si accompagna a regole di utilizzo esoteriche e decisamente poco chiare.

Noi qui accogliamo la sfida dell'eredità multipla, cercando di definirne il significato in modo chiaro e semplice, ed eliminando le possibili condizioni di ambiguità. Nostro scopo è quello di renderla, si spera, facile da usare ed anzi assai utile.

2.4.1. I problemi

I problemi legati all'eredità multipla sono essenzialmente di due tipi: decidere da quale ramo della genealogia delle classi debbano essere ereditati i metodi ed in che modo vengano ereditate le variabili di istanza.

2.4.1.1. Variabili di istanza

Per quanto riguarda queste ultime, i problemi sorgono principalmente per i linguaggi che permettono l'esportazione delle variabili di istanza. Supponiamo infatti di avere di avere una classe padre, due classi figlie, ed una classe che sia simultaneamente figlia di queste ultime due. Se tutte e due le classi figlie definiscono una nuova variabile di istanza di nome uguale, non è affatto chiaro cosa sarà visibile ed utilizzabile dalla discendente di entrambe. Potrebbero essere infatti visibili due variabili distinte, oppure una sola che unifichi le due, od ancora una sola che mascheri l'altra.

Un'altro problema, simile al precedente, riguarda le variabili di istanza delle sopraclassi raggiungibili tramite due percorsi distinti sull'albero della gerarchia; tali variabili potranno essere ereditate in una copia singola oppure in copie multiple, una per ogni percorso possibile. Per esempio, in C++, se non viene espressamente utilizzata la keyword "virtual", i record corrispondenti a tali sopraclassi vengono inclusi più volte, una per ogni cammino sul grafo orientato rappresentante le relazioni gerarchiche. Questo significa che, agli effetti pratici, la struttura interna di una sopraclasse verrà ereditata più volte, conservando record distinti; il C++ richiede infatti, nel caso vengano usate delle variabili ereditate in tale modo, di specificare quale di tali copie si intenda riferire.

Questo modo di operare, tuttavia, rappresenta in un certo qual modo una violazione del principio secondo il quale un oggetto di una sottoclasse "è" anche un oggetto della sopraclasse.

Facciamo un esempio: se abbiamo la classe degli animali, si può definire in questa una variabile che contiene, per esempio, il numero di zampe. Da questa classe facciamo discendere due sottoclassi distinte, una che descrive i felini, ed una che descrive gli animali domestici; se ora definiamo la classe dei gatti, ereditando sia da animali domestici che da felini, considerando un istanza di gatto, si avrà che il numero di zampe sarà lo stesso, per quel determinato gatto, sia che lo si consideri un animale domestico che un felino, in quanto un gatto è un animale (istanza di una sottoclasse), ed appunto in quanto tale godrà della caratteristica di avere un certo numero di zampe. Pensare di ereditare tale proprietà in duplice copia, ossia pensare che un gatto possa avere due diversi numeri di zampe, a seconda di come lo si considera, è sotto questa luce quantomeno bizzarro.

2.4.1.2. Eredità dei metodi

Un'altra categoria di problemi riguarda l'utilizzo dei metodi, ed il modo in cui questi vengono ereditati dalle sopraclassi. Possono verificarsi infatti facilmente situazioni di ambiguità, nel caso non sia chiaro da dove debbano venire ereditati i metodi quando definiamo una nuova sottoclasse.

Consideramo sempre il caso in cui una classe abbia due figlie con una sottoclasse in comune. Se in entrambe le due classi figlie è definito un metodo di nome uguale, la classe sottoclasse di entrambe potrebbe invocare uno dei due metodi oppure dichiarare errore in runtime, od ancora rilevare l'ambiguità staticamente.

Sia C++ che CLOS, in situazioni di questo tipo, danno la precedenza alle classi che, nella lista delle sopraclassi, compaiono per prime; in caso di ambiguità verrà quindi data a queste la preferenza per quanto riguarda l'eredità dei metodi. Tale metodo presenta alcuni svantaggi: prima di tutto, viene richiesto al programmatore di tenere a mente quali classi hanno la precedenza su quali altre, il che, specie in caso di dipendenze complesse, può essere abbastanza difficoltoso, ed in secondo luogo, il preferire un metodo ad un altro, per lo stesso messaggio, sulla base di regole più o meno aritrarie, può nascondere lo stesso tipo di incongruenze logiche che abbiamo visto poc'anzi nel caso delle variabli di istanza. Difatti, sotto una situazione ambigua di questo tipo, il fatto che due metodi distinti vengano ereditati simultaneamente per lo stesso messaggio è un po' come asserire che l'oggetto, ricevendo il messaggio corrispondente, dovrebbe comportarsi allo stesso tempo in due modi diversi.

2.4.2. Risolvere le ambiguità

Risolvere i problemi legati all'eredità delle variabili di istanza non è difficile, e le scelte possibili sono diverse.

2.4.2.1. Variabili di istanza

Per quanto riguarda variabili di istanza omonime, un possibile modo per risolvere il problema alla radice consiste nel non consentire l'esportazione delle variabili di istanza dichiarate nell'ambito di una classe, cosa che tra l'altro consente di migliorare l'incapsulamento e di rendere completamente opaca l'implementazione delle classi. Dato che le variabili di istanza, in questo caso, non sono visibili al di fuori della definizione della classe, ciascuno stadio della gerarchia avrà conoscenza solo delle "proprie" componenti, quelle definite in quel livello, e la classe discendente da più genitori potrà accedere alle componenti definite nei livelli superiori solo tramite messaggi. Dal punto di vista implementativo, nella istanza risultante verranno quindi conservate copie separate e distinte delle componenti definite lungo i due o più rami, che non avranno comunque modo di interagire fra loro.

Per quanto riguarda invece le variabili di istanza ereditate tramite più cammini sul grafo delle classi, la nostra scelta è quella di rimanere aderenti all'interpretazione più intuitiva; pertanto le istanze dei nostri oggetti conserveranno internamente in ogni caso una unica copia delle componenti dalle sopraclassi, qualunque sia il modo in cui queste ultime vengono ereditate.

2.4.2.2. Eredità dei metodi: il caso unario.

Risolvere il problema dell'eredità dei metodi è invece questione in generale assai meno banale, e vedremo ora come affrontare la cosa in modo sistematico.

Abbiamo visto come, in alcuni linguaggi, venga lasciata libertà di ereditare metodi che riferiscono lo stesso messaggio secondo più percorsi possibili sul grafo delle classi e come, in caso di ambiguità, alcune regole implicite consentano di individuare quale metodo fra quelli possibili venga effettivamente ereditato.

Un approccio piuttosto diverso è stato adottato in OOPLog ([CM91]), una estensione object oriented del Prolog sviluppata come progetto interno all'Università di Udine. In tale ambiente, infatti, viene considerato illegale ereditare due metodi distinti per la stessa classe da due o più sopraclassi distinte, ed è anzi richiesto che, in tale circostanza, venga definito un metodo locale alla nuova classe in corso di definizione, o comunque che venga risolto l'ambiguità. Ma vediamo un esempio concreto.

Nell'esempio illustrato in Figura 2.2, se abbiamo le definizioni fun(B) e fun(C), e non abbiamo alcuna definizione di fun in D, si verificherà un conflitto. Questo è conforme all'idea secondo cui una istanza di D non può comportarsi simultaneamente in due modi distinti, come l'eredità simultanea di fun da B e C lascerebbe supporre. Riconoscere queste condizioni di ambiguità è piuttosto semplice, e consente una prima semplificazione del problema.

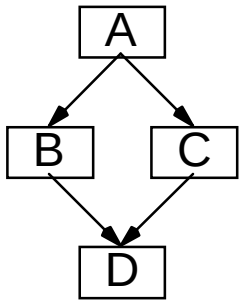


Figura 2.2.: Eredità multipla

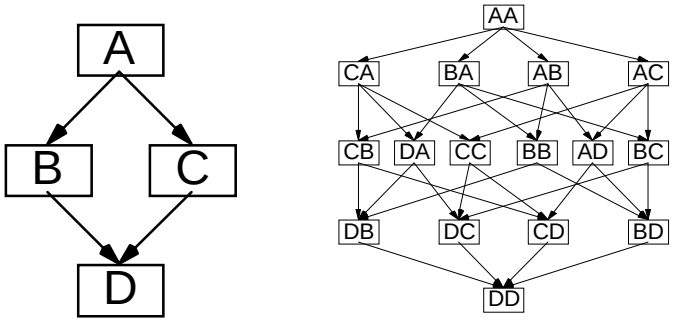
Ora, potremmo chiederci cosa accade se abbiamo invece fun(A) e fun(B), ma non fun(C) e fun(D). Le scelte sono due: nella prima, attenendosi ad un rigido criterio logico, C eredita il comportamento da A, e, all'atto della definizione di D, si ha un conflitto fra fun(A) (in C) e fun(B). Nella seconda, il sistema sceglie il metodo più specifico fra quelli ereditati, limitando i casi di errore a quelli in cui una scelta non è possibile. Secondo questo secondo meccanismo, quando D viene definito, viene scelto il metodo più specifico fra fun(A) e fun(B); in buona sostanza, D eredita dunque fun(B). Non ci sono particolari pro e contro fra le due possibilità; si tratta solo di effettuare una scelta; nel nostro caso, optiamo per questa seconda, più tollerante interpretazione: una classe eredita il metodo più specifico (se è possibile inviduarne uno univocamente) fra quelli delle sue sopraclassi.

Per effettuare operativamente il controllo della legalità, possiamo procedere per induzione: sapendo che non possono esserci cicli nel grafo delle classi (e che il grafo è connesso, anche se questa non è una condizione necessaria) possiamo effettuare un topological sort, e considerare le classi una alla volta. Come condizione iniziale, controlleremo la legalità della definizione della sola classe di partenza (object); nelle iterazioni successive, analizzeremo tutte le sopraclassi della classe esaminata, e controlleremo i metodi da queste ereditate corrispondenti al messaggio che stiamo controllando; se non è possibile individuare un unico metodo più specifico, e non vi è una ridefinizione locale, l'eredità sarà illegale.

È da notare che le condizioni di errore riconosciute con questa analisi sono quelle in cui si manifesta una palese incongruenza nella definizione dei messaggi, cosa che porta direttamente ad una situazione di ambiguità. In generale, utilizzando una strutturazione coerente di classi e metodi, non si avrà alcuna condizione di errore del tipo descritto.

2.4.2.3. L'eredità multipla nel caso dei multimetodi

Quanto detto finora vale nel caso di metodi unari; per analizzare il caso di messaggi di arità maggiore (necessario per le generic functions), faremo nuovamente ricorso al grafo descritto precedentemente.



(a) Classi singole (b) Coppie di classi

Figura 2.3.: Estensione del grafo delle classi: eredità multipla.

Il risultato del procedimento di costruzione nel caso di una semplice gerarchia di classi contentente eredità multipla è rappresentato in Figura 2.3; il grafo risultante è un po' più complesso, ma al momento stiamo effettuando solo una trattazione teorica; un tale grafo non sarà effettivamente costruito, durante la compilazione.

Cerchiamo adesso di scoprire cosa accade alle nostre generic functions in questo caso, cercando un riscontro sul grafo costruito. Notiamo immediatamente che una n-pla che sia la destinazione di altre n-ple tramite la nostra relazione dovrà essere una specializzazione di tutte quante. Per esempio, nel caso illustrato in figura, la coppia DB sarà una specializzazione di CB, DA e BB. È chiaro che, se i metodi ereditati da queste altre n-ple sono in discordanza, si verificherà una ambiguità, ossia si avrà un conflitto nella definizione del messaggio per la coppia DB se almeno due dei metodi ereditati dalle n-ple progenitrici saranno incompatibili, e se non ne esiste una ridefinizione locale. Per esempio, se abbiamo due definizioni in AB e in CA, si avrà che BB eredita da AB, CB eredita da CA, e, all'atto di stabilire il metodo da applicare per DB, non sarò in grado di scegliere un metodo più specifico fra quello di BB (cioè AB) e quello di CB (ossia CA), in quanto fra AB e CA nessuno è "più specifico" dell'altro (cfr. grafo), ma le n-ple sono scorrelate.

Anche in questo caso, osservando il grafo derivato, abbiamo una relazione d'ordine parziale ed anche in questo caso possiamo effettuare un topological sort delle n-ple, e

cercare, per induzione, eventuali conflitti analizzando le n-ple una alla volta.[4]

Riassumendo, con l'analisi del grafo possiamo individuare con certezza staticamente tutte le condizioni di ambiguità derivanti dall'eredità multipla, sia nel caso unario che nel caso n-ario, ed inoltre, riconoscendo le condizioni di conflitto tramite l'esplorazione del grafo, possiamo anche suggerire all'utente il punto esatto in cui si è verificato il conflitto, e quindi la forma del multimetodo che è necessario definire per risolvere il problema.

Rendendo quindi impossibili le situazioni di ambiguità, e restando al contempo fedeli ai principi base della programmazione object-oriented, viene quindi anche a cadere interamente la necessità di introdurre meccanismi aggiuntivi, spesso difficili da gestire da parte dell'utilizzatore, quali ad esempio regole implicite, oppure la tecnica interface-subinterface utilizzata da Java[Fla97], od altro.

Se ne ottiene quindi una semplificazione notevole nell'utilizzo dell'eredità multipla, e la certezza dell'unicità del metodo da invocare.

Come importante effetto collaterale della ricerca descritta, vengono inoltre individuate, allo stesso tempo, anche le condizioni di ambiguità descritte descritte nel cap. 2.3.3, che si manifestano, allo stesso modo di quelle derivanti dall'eredità multipla, nei casi in cui non sia possibile scegliere per un messaggio un metodo unico in corrispondenza di ogni n-pla.

Di conseguenza, con un unico strumento, ossia la ricerca di conflitti tramite l'esplorazione del grafo, risolviamo simultaneamente tutte le condizioni di ambiguità, comunque originate, rendendo la regola del cap. 2.3.3 solo un caso particolare.

[4] In una tipica implementazione efficiente si partirà per l'analisi dai soli metodi definiti, che saranno normalmente molti meno di tutte le n-ple possibili.

2.5. Considerazioni sull'efficienza

Introdotto il quadro in cui intendiamo operare, faremo ora alcune considerazioni su alcune soluzioni implementative per i linguaggi object-oriented, ed introdurremo le tecniche che il nostro linguaggio sperimentale potrà utilizzare per conciliare al meglio l'eleganza e la semplicità di programmazione con una buona velocità in fase di esecuzione.

2.5.1. I problemi da risolvere

I problemi di efficienza sono principalmente di due tipi: come trattare i tipi di base del linguaggio, e come ridurre l'overhead durante la ricerca del metodo corrispondente ad un determinato messaggio.

2.5.1.1. I tipi semplici

Per quanto riguarda i tipi più semplici quali interi, numeri floating point, caratteri e simili, i progettisti di linguaggi si trovano di solito di fronte ad una alternativa: o realizzare un linguaggio object-oriented "puro", nel quale tutti i dati disponibili siano manipolabili sotto forma di oggetto, oppure creare dei tipi a parte, al di fuori dalla gerarchia delle classi, per riferire i tipi più semplici.

La prima scelta è certamente la più elegante, in quanto presenta allo sviluppatore un ambiente omogeneo, costituito esclusivamente da oggetti, ma solitamente è molto più onerosa in termini operativi, in quanto anche i dati più semplici vanno trattati come oggetti, il che comporta un certo overhead: se bisogna creare un record in memoria per ogni dato, anche il più semplice, ed invocare il dispatcher anche per le operazioni più semplici, la quantità di operazioni di contorno per effettuare le manipolazioni dei dati diventa assai elevata.

La seconda scelta esce invece dal paradigma della programmazione ad oggetti per poter utilizzare direttamente i registri del microprocessore, o singole locazioni di memoria, per la manipolazione dei dati primitivi, e quindi guadagnare in velocità di esecuzione.

Per la realizzazione del nostro linguaggio, cercheremo di conciliare l'aspetto dell'eleganza e della omogeneità dell'ambiente con una elevata efficienza. Per fare questo, presenteremo al programmatore un ambiente composto da soli oggetti, ma cercheremo un'approccio implementativo che ci consenta di eliminare quasi totalmente l'overhead che solitamente si ritiene necessario. Alcune delle possibilità implementative, ed il nostro suggerimento per una soluzione efficiente, verranno presentate nel capitolo 2.5.2.

2.5.1.2. Il dispatching dei messaggi

Per quanto riguarda il dispatching, le soluzioni sono da un lato il non effettuare alcuna operazione in run-time per la scelta del metodo, ed effettuare la disambiguazione staticamente, e dall'altra avere un dispatcher che risale dinamicamente gli elenchi dei metodi esplorando la gerarchia delle classi, cercando un metodo adatto. La prima soluzione rappresenta, in qualche modo, una forzatura all'idea stessa di programmazione object-oriented, eppure è usata in C++: se non viene utilizzata esplicitamente la keyword "virtual", ossia se non viene richiesta appositamente la disambiguazione dinamica, la scelta del metodo avviene staticamente in fase di compilazione. Dal lato opposto si possono certamente annoverare le varie estensioni object oriented del LISP, linguaggio dinamico per natura: ad ogni invocazione viene cercato il metodo adatto tramite una esplorazione dell'albero delle classi (che peraltro può modificarsi in runtime, quindi non ci possono essere molte alternative a questo modo di procedere). Naturalmente la prima soluzione è la migliore in termini di velocità, mentre la seconda, al prezzo di considerevoli rallentamenti durante l'esecuzione, è la più elegante dal punto di vista concettuale. Un'approccio intermedio è scelto dalla maggior parte dei linguaggi compilati orientati agli oggetti: l'utilizzo in runtime di tabelle di metodi costruite staticamente. In caso di ereditarietà singola, non è difficile ottenere una buona efficienza utilizzando delle tabelle di dispatching associate alle classi, mentre nel caso di ereditarietà multipla le cose si complicano non poco.

È quest'ultima la soluzione che adotteremo anche noi, effettuando durante l'esecuzione una ricerca del metodo su tabelle costruite durante la compilazione; dato che intendiamo utilizzare i multimetodi, dovremo naturalmente adattare di conseguenza il meccanismo di dispatching. Come vedremo, la costruzione delle tabelle sarà leggermente più laboriosa di quanto non accada per i linguaggi convenzionali, ma, tramite meccanismi opportuni, saremo in grado di realizzare una implementazione che riduca veramente all'osso l'overhead necessario, come illustreremo nel capitolo 2.5.3. Nella maggior parte dei casi, il dispatching si potrà ridurre infatti all'esecuzione di pochissime istruzioni macchina, incluse direttamente nel codice nel punto in cui si deve effettuare la chiamata.

2.5.2. Implementare efficientemente gli oggetti

L'implementazione degli oggetti nel sistema naturalmente non può prescindere da considerazioni di efficienza, ed abbiamo visto come il presentare allo sviluppatore un ambiente composto esclusivamente da oggetti sia solitamente associato ad un grosso overhead, tanto che molti linguaggi preferiscono uscire dalla metodologia object-oriented per guadagnare in velocità di esecuzione.

Fra i linguaggi “ibridi”, che non hanno come base esclusiva la tecnologia object oriented, possiamo certamente citare il C++, dove i tipi di base sono decisamente trattati come locazioni di memoria, sia per via della similitudine del linguaggio con il suo progenitore, il C, e sia perchè uno degli scopi del C++ è quello di generare codice il più efficiente possibile. Java è un po’ meno drastico nella scelta, e comprende infatti diverse caratteristiche orientate più all’eleganza del codice che alla velocità ad ogni costo (es.: un meccanismo di garbage collection [Fla97], l’assenza di puntatori utilizzabili dall’utente etc.). Malgrado questo, anche Java rinuncia ad avere oggetti ovunque, ed offre sia tipi nativi (int, float, long...) che oggetti veri e propri che racchiudono gli stessi dati (Integer, Float, Long) e che consentono l’applicazione di messaggi.

L’approccio scelto da Smalltalk è viceversa estremo nell’altro senso: ogni dato è un oggetto, ma di converso purtroppo la velocità non è una delle caratteristiche che di solito si annoverano fra le qualità di tale linguaggio (che, comunque, è di norma solo semicompilato). L’approccio scelto da Dylan[App95] utilizza un’interessante escamotage: gli oggetti sono presenti ovunque, ma è possibile definire delle classi come "sealed" (sigillate);[5] di tali classi non possono essere ricavate ulteriori sottoclassi. Risulta chiaro che, se la classe degli interi è sealed, dove compare un intero nel codice non potrà esserci altro che un intero e non una istanza di una sottoclasse. È quindi possibile determinare staticamente il metodo (la funzione) da invocare, ed il dato può essere tenuto senza complicazioni in un registro singolo, od in una sola locazione di memoria.

Per focalizzare con precisione il problema, pensiamo all’implementazione più immediata di un oggetto in memoria: un record (un blocchetto di dati), riferito tramite un puntatore, che contiene i valori delle variabili di istanza più alcune informazioni aggiuntive, quali dati per il garbage collector e soprattutto, se è necessario effettuare delle disambiguazioni a run time, un qualche riferimento (puntatore, identificatore od altro) alla classe di appartenenza dell’oggetto.

Consideriamo ora una semplice spedizione di messaggio:

```
test(a:frutta);
```

Il parametro del messaggio verrà passato al dispatcher tramite un puntatore all’oggetto in questione, conservato in un registro macchina del processore. Il dispatcher si occuperà poi di individuare la classe di appartenenza del parametro e di scegliere il metodo adatto.

Fino a qui nulla di strano; ma cosa succede nel caso di un tipo primitivo?

[5] Una opzione simile è offerta anche da Java con la keyword “final”.

```
test(a:long);
```

Considerazioni di efficienza suggerirebbero di passare il parametro in un registro, che tutto sommato è fatto apposta per conservare tipi nativi (interi, byte, caratteri singoli e affini). Ma passando il long nel registro, il dispatcher non ha più modo di determinare dinamicamente il tipo del parametro passato. Ancora peggio, poi, se costruiamo una sotto-classe di long, per esempio: al dispatcher potrebbe arrivare un numero oppure un puntatore, ma a livello macchina non c'è alcun modo di distinguere fra i due. E d'altronde, costruire un oggetto effettivo in memoria per ogni dato comporta un overhead improponibile.

Mettiamoci dunque nell'ottica di avere un dispatcher, di volere presentare tutti i dati come oggetti, e quindi poter passare ai metodi sia oggetti veri che tipi primitivi, e proviamo a cercare delle soluzioni efficienti al problema.

2.5.2.1. Identificare con un tag un puntatore ad oggetto

Una delle possibilità è quella di rinunciare a parte della capacità del registro per conservarvi un tag che identifichi il parametro passato. Per esempio, se il registro è di 32 bit, si può pensare di utilizzare 4 o 8 bit per registrare un tag identificativo.

La soluzione risolve un problema ma purtroppo ne crea altri. Innanzitutto viene persa parte della capacità del registro, limitando sia lo spazio indirizzabile quando il registro contiene un puntatore, che il range di valori memorizzabili quando si tratta di un intero. In secondo luogo, per distinguere il tipo ed effettuare le scelte del caso, il dispatcher dovrebbe effettuare un pesante lavoro sul registro per estrarre le due componenti, perdendo così in tempo quanto si guadagna (poco) in numero di registri usati. Tutto sommato questa soluzione è quindi sconsigliabile.

2.5.2.2. Bitmask, oppure tag separati

Simile alla precedente, ma i tag vengono conservati in una zona separata dell'istanza, in modo da non perdere capacità di indirizzamento. Si tratta di una soluzione migliore, per quanto concerne la memorizzazione degli oggetti dentro alle istanze, ma non viene risolto il problema del passaggio dei parametri al dispatcher ed al metodo chiamato. Può essere usato se il dispatcher viene espanso inline nel codice nel punto corrispondente alla chiamata, ma non è una soluzione ottimale.

2.5.2.3. Simulare un oggetto

Si può pensare di riservare in memoria (sullo stack, per esempio), spazio per due parole macchina, in cui nella prima sia registrato un identificativo per l'oggetto e nella seconda il dato di tipo semplice che ci interessa, giocando sul fatto che i tipi base non hanno bisogno del trattamento completo riservato agli oggetti "veri". Al dispatcher viene ugualmente passato un puntatore.

In pratica viene realizzata la parte minima dell'oggetto sufficiente per far funzionare il dispatcher. Si risparmia parte dell'overhead necessario per l'inizializzazione di un oggetto completo, ma ugualmente si perde tempo per gli accessi indiretti tramite puntatore al tipo ed al valore del tipo base. Inoltre ogni volta che si vuole effettuare una operazione su un intero, per esempio, è necessario registrarlo in memoria, vanificando così la comodità di poter avere il dato nel registro pronto per l'esecuzione di una istruzione macchina.

La soluzione è migliore delle precedenti, ma è possibile escogitarne di più efficienti.

2.5.2.4. Due registri per parametro

Visto che ci troviamo sempre a dover manipolare un dato ed il suo tipo, possiamo pensare di utilizzare, per ogni parametro, due registri distinti: uno per l'identificatore del tipo ed uno per il dato (intero o puntatore all'istanza che sia). Il primo registro viene passato al dispatcher per la determinazione del tipo, e la coppia viene passata infine al metodo trovato per l'esecuzione.

L'efficienza di questo metodo è decisamente superiore alle tecniche precedenti: il dato di tipo base viene conservato in un registro, e si può impiegare direttamente, senza richiedere accessi in memoria aggiuntivi. Gli unici svantaggi consistono nel fatto che viene impiegato il doppio di registri per i parametri rispetto alla norma, e che quindi deve essere salvato il doppio di dati nello stack ad ogni chiamata, il che può portare ad un overhead sensibile. Su architetture che contemplano pochi registri, gli accessi alla memoria per il parcheggio dei parametri possono essere parecchi.

2.5.2.5. Due registri, ma non sempre: una implementazione efficiente

Per migliorare ulteriormente, proponiamo qui una implementazione alternativa, altamente efficiente, che risolve in modo effettivo il problema.

Consideriamo con più attenzione le condizioni nelle quali può verificarsi che il dispatcher debba scegliere fra un oggetto ordinario ed un tipo base. Ad esempio, osserviamo questa invocazione:

```
test(a); // tipo statico di a:  frutta
```

Noi sappiamo che, dinamicamente, il messaggio potrà essere chiamato solo con istanze della classe frutta, o di sue sottoclassi. In ogni caso, si tratterà sempre di un oggetto ordinario, quindi sarà sufficiente in ogni caso passare al dispatcher solo il puntatore alla istanza. Viceversa, consideriamo questa chiamata:

```
test(a); // tipo statico di a:  long
```

Il tipo dinamico di a non potrà che essere long o una sua sottoclasse. Se assumiamo che i tipi base non possano generare sottoclassi, il tipo dinamico sarà sempre long, e dunque il metodo da chiamarsi potrà essere determinato staticamente, evitando interamente l'uso del dispatcher. Se i parametri sono più di uno, non sarà comunque necessario effettuare la disambiguazione sul parametro dichiarato in questo modo. Il passaggio del parametro avverrà utilizzando un solo registro.

Vediamo adesso il caso critico:

```
test(a); // tipo statico di a:  object
```

Al posto di a potrà essere passato un oggetto normale oppure un tipo base; allora, solo in questo caso, utilizzeremo due registri per passare il parametro.

In pratica, se il parametro formale ha come tipo statico una classe che ha un tipo base come sottoclasse (object, e pochi altri), utilizzeremo due registri, altrimenti sarà sufficiente utilizzarne uno solo. Dal momento che la stragrande maggioranza delle classi non ha tipi base come sottoclassi, verrà utilizzato nella maggior parte dei casi un unico registro contenente il puntatore all'istanza oppure il dato semplice, di cui è noto staticamente il tipo dinamico.

Laddove necessario, sarà facile, all'atto dell'invocazione del dispatcher, convertire il singolo puntatore all'istanza in una coppia di registri, caricando il registro che identifica il tipo con il valore del tipo memorizzato nell'istanza (una sola istruzione macchina, un accesso indiretto con offset), oppure caricare il secondo registro con l'identificatore del tipo base, che è noto a priori (un caricamento immediato di registro, dunque una sola istruzione macchina anche in questo caso).

Agli effetti pratici, sia le istanze che i tipi primitivi verranno conservati nelle istanze

in una sola parola macchina, eccettuato object e pochi altri, che saranno trattati a tutti gli effetti come tipi primitivi composti da due parole macchina.

Naturalmente, visto che il linguaggio presenta allo sviluppatore soltanto oggetti, questi dettagli implementativi saranno del tutto trasparenti; tuttavia, utilizzando la soluzione proposta, si può rendere il linguaggio uno strumento non solo coerente e semplice da utilizzare, ma anche agile in termini di efficienza e velocità di esecuzione.

2.5.3. Implementazione efficente del dispatching

Come accennato precedentemente, il dispatcher può essere implementato utilizzando un approccio parzialmente statico oppure dinamico: statico costruendo delle tabelle contenenti puntatori ai metodi oppure dinamico effettuando una ricerca in runtime sulle strutture dati che rappresentano la gerarchia delle classi. Uno degli scopi del nostro linguaggio è quello di raggiungere, ove possibile, una elevata efficienza di esecuzione a runtime; d'altronde, tutto il controllo su tipi statici e tipi dinamici è finalizzato proprio ad una implementazione efficiente del dispatcher, ed all'eliminazione dei possibili errori di tipo (e quindi di dispatching) a runtime. L'approccio scelto, dunque, è quello delle tabelle costruite staticamente, in modo non dissimile da molti altri linguaggi. Quello che caratterizza il nostro caso è il particolare approccio derivante dall'uso delle generic functions; la disambiguazione, infatti, non viene effettuata solo considerando un singolo oggetto, ma una n-pla. Ciò porta ad una implementazione leggermente più complessa delle tabelle di dispatching, come ora vedremo.

2.5.3.1. Messaggi unari

Cominciamo a vedere un esempio concreto, limitandoci a considerare, per il momento, la spedizione di messaggi a destinatari singoli, secondo le modalità classiche.

Con riferimento alla semplice gerarchia di classi quale quella rappresentata in Figura 2.4, un messaggio fun() potrà avere una definizione di metodo in corrispondenza di ciascuna delle classi.

Se disponiamo dei metodi fun(Object), fun(A) e fun(C), avremo bisogno, per una qualsiasi istanza di una classe fra quelle rappresentate, di un modo rapido per scoprire quale, fra i tre metodi, sia quello corretto da invocare. La soluzione più intuitiva è quella di includere nelle istanze, insieme con le variabili, anche un puntatore al metodo da invocarsi per il messaggio fun(). In tale modo il dispatcher dovrà solamente effettuare un salto alla routine puntata dal valore situato ad un offset noto nell'istanza per chiamare il metodo

corretto. L'unico inconveniente di tale soluzione è che il puntatore deve essere replicato in ogni istanza della classe, il che comporta, se i metodi sono molti, uno spreco di spazio in memoria, ed una perdita di tempo per la copiatura della tabella ad ogni nuova creazione di oggetto.

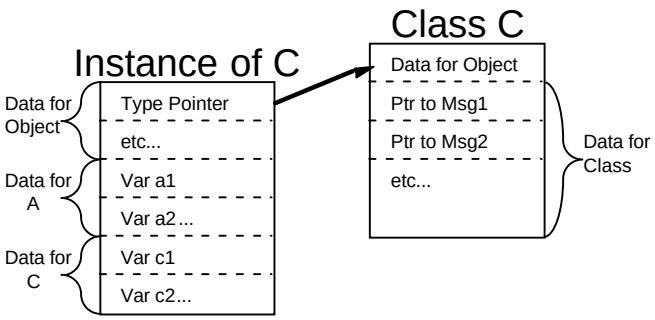


Figura 2.5.: Puntatori ai messaggi contenuti nel record delle classi

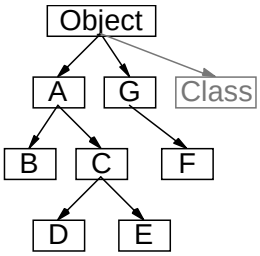


Figura 2.4.: Esempio di gerarchia

Una soluzione molto simile ma assai più efficiente è quella di conservare i puntatori ai metodi nel record della classe corrispondente alle istanze; questo è anche ragionevole, dal momento che si suppone che tutte le istanze di una determinata classe si comportino nello stesso modo. Se utilizziamo come identificatore di tipo per gli oggetti un puntatore alla classe di appartenenza, il recupero del puntatore al metodo si può ottenere facilmente tramite un singolo accesso indiretto, come evidenziato nella Figura 2.5.

In buona sostanza, ogni istanza conterrà il puntatore alla propria classe, nella quale, ad un offset dato, sarà contenuto il puntatore al metodo cercato. Per esempio, nelle istanze di C sarà contenuto, come si vede nello schema, un puntatore alla classe C, nella quale, ad un offset prefissato per quel messaggio, sarà contenuto il puntatore al metodo applicabile per C, ed analogamente per le altre classi. Il dispatcher non dovrà fare altro che estrarre il puntatore dal record della classe, e saltare all'indirizzo del metodo così ottenuto.

Scendiamo ora un po' più in dettaglio nella costruzione delle tabelle di dispatching, e nella assegnazione degli offset.

Supponiamo di avere un metodo definito su Object. Tutte le istanze di qualunque classe, essendo istanze di sottoclassi di Object, dovranno essere in grado di rispondere al mes-

saggio corrispondente a tale metodo; di conseguenza, l'offset relativo sarà assegnato per tutte le istanze. Se ora definiamo un ulteriore messaggio, i cui metodi sono definiti soltanto per la classe A e sue sottoclassi, non avrà senso allocare l'offset anche per Class, G, F e le altre classi; dunque, l'offset per questo nuovo messaggio andrà riservato soltanto per le sottoclassi di A. Chiaramente, se abbiamo un metodo definito solamente su G, potremo usare nuovamente lo stesso offset per il nuovo messaggio, in quanto non si potrà verificare alcun conflitto, in modo del tutto analogo a quanto accade con l'assegnazione degli offset per le variabili di istanza.

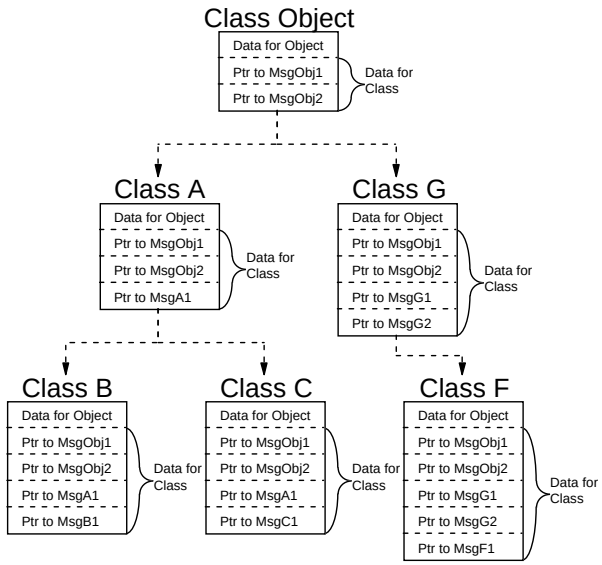


Figura 2.6.: Assegnazione degli offset per i messaggi, caso unario

Lo schema raffigurato in Figura 2.6 può dare una idea piuttosto chiara, quindi, di come possano venire assegnati gli offset per i messaggi nel caso unario di Figura 2.4.

Un meccanismo di questo genere viene normalmente utilizzato in C++ e linguaggi similari. In pratica viene effettuata una visita in profondità sull'albero delle classi e, per ogni nuovo messaggio incontrato, verrà aggiunto l'offset relativo. Nel nostro caso, possiamo inoltre fare una interessante osservazione.

Supponiamo di avere due definizioni di metodi, con lo stesso nome ed uguale arità, in C ed in G, ma non in Object ed in A. Grazie al type checker che costruiremo sulla base di quanto detto nei capitoli precedenti, sarà possibile disambiguare già staticamente se l'oggetto sarà di C od una sottoclasse, o piuttosto di G od una sottoclasse, in quanto l'utilizzo del metodo su Object o su A sarà illegale. Di conseguenza, agli effetti della compilazione, i messaggi definiti nei due sottoalberi di radice C e G potranno essere considerati come

completamente distinti, e potranno addirittura avere assegnati due offset diversi. Se poi tali due metodi sono gli unici corrispondenti al messaggio in esame, potremo risolvere la chiamata in modo interamente statico, eliminando quindi interamente la necessità di invocare il dispatcher.

Compiuta questa analisi, vediamo di andare un po' oltre, modificando opportunamente la costruzione per contemplare anche le n-ple di classi, necessarie per l'implementazione delle generic functions.

2.5.3.2. Messaggi di arità arbitraria

Dato che i messaggi sono ora definiti su n-ple, in teoria anche le tabelle di dispatching dovrebbero essere n-dimensionali, il che però non è rappresentabile in modo diretto in memoria; possiamo comunque aggirare il problema affrontandolo per induzione, ossia scomponendo la disambiguazione di un messaggio msg(a,b,c,d,...) in una prima fase in cui divideremo l'insieme di metodi eleggibili come candidati in gruppi risolvendo il primo parametro, e riducendo quindi il problema di una dimensione. Successivamente suddivideremo ancora l'insieme dei candidati rimanenti sulla base del secondo parametro e così via, finchè non ne resterà uno solo, che sarà quello scelto. Per esempio, se abbiamo le definizioni:

```
msg(A,A)                    msg(B,A)
msg(A,B)                    msg(B,B)
```

con B sottoclasse di A, possiamo risolvere dapprima il primo parametro (se sottoclasse di A, resteranno msg(A,B) e msg(A,A), se sottoclasse di B gli altri due) ed al secondo passo distinguere sul secondo, ottenendone quindi uno solo. Se ad uno dei passi non troviamo nessun candidato adatto, l'invocazione del messaggio è, ovviamente, illegale.

Vediamo come la cosa può essere tradotta in termini di offset.

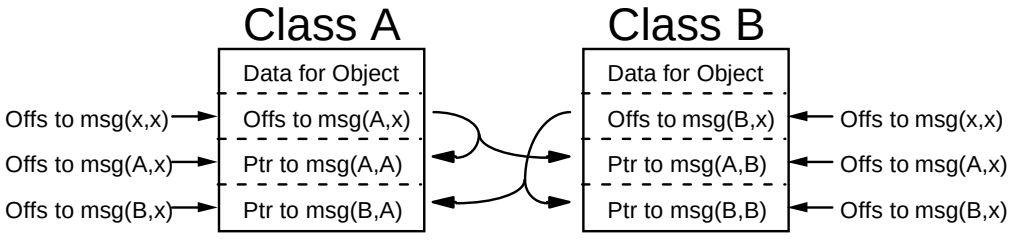


Figura 2.7.: Assegnazione degli offset per messaggi binari.

Come si può vedere in Figura 2.7, nella posizione corrispondente a msg(x,x), verrà trovato un ulteriore offset che indicherà dove trovare il puntatore al metodo durante la fase successiva. In questo modo, per la risoluzione di un messaggio definito su n-ple comprendente metodi con parametri distinti su tutte le n posizioni possibili, saranno necessari al più O(n) istruzioni macchina (di solito proprio n, se è disponibile una move indiretta indicizzata da memoria a registro) per ottenere il puntatore desiderato. Viceversa, il numero di posizioni occupate nella tabella di dispatching può essere abbastanza elevato, essendo in linea di principio proporzionato al numero di classi fra cui fare la disambiguazione elevato al numero di parametri. È comunque da tener presente che i metodi definiti non saranno normalmente corrispondenti a tutte le possibili variazioni su tutte le posizioni (anche perchè il numero di metodi sarebbe esponenziale anch'esso), e dunque non sarà necessario effettuare la disambiguazione su tutti i parametri e che il numero totale di parametri di un messaggio è, nell'uso normale, assai modesto, superando raramente i 5 o 6. Possiamo pensare che il numero di parametri su cui verrà effettuata la disambiguazione sia di norma al massimo tre o quattro, con variazioni comunque su tre o quattro classi distinte, il che porterebbe, nei casi più complessi ipotizzabili, a dimensioni della tabella di circa 300 posizioni, numero decisamente più che accettabile, considerate anche le elevate disponibilità di memoria di oggigiorno.

L'implementazione descritta, qualora la disambiguazione venga fatta solamente su uno dei parametri, permette di ottenere la medesima efficienza degli analoghi meccanismi disponibili in C++ od in altri linguaggi compilati che utilizzino un approccio simile.

Un ulteriore risparmio può essere ottenuto con una soluzione alternativa, vantaggiosa nel caso di metodi definiti in modo molto asimmetrico (ossia con una preponderanza di definizioni lungo alcuni rami rispetto ad altri): si può infatti registrare, all'offset corrispondente al messaggio multidimensionale, un puntatore ad un subdispatcher specializzato, in grado di gestire il caso n-1 in modo ottimizzato a seconda delle circostanze.

In tale modo non sarà necessario registrare nella tabella tutte le possibilità per il dispatching, ma solo quelle effettivamente necessarie. Ad esempio, supponiamo di avere la classi A,B e C, con A al vertice della gerarchia, B sottoclasse di A, e C sottoclasse di B, e di avere definiti i seguenti metodi:

```
msg(A,A)
msg(B,C)
msg(C,B)
msg(C,C)
```

Il metodo generico di calcolo delle tabelle porterebbe alla situazione rappresentata in Figura 2.8.

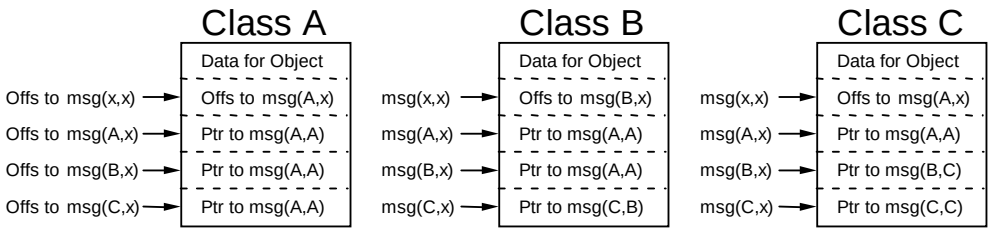


Figura 2.8.: Offset calcolati con il metodo generale

Di converso, utilizzando degli handler appositi per le varie situazioni, si potrebbe ottenere la situazione di Figura 2.9.

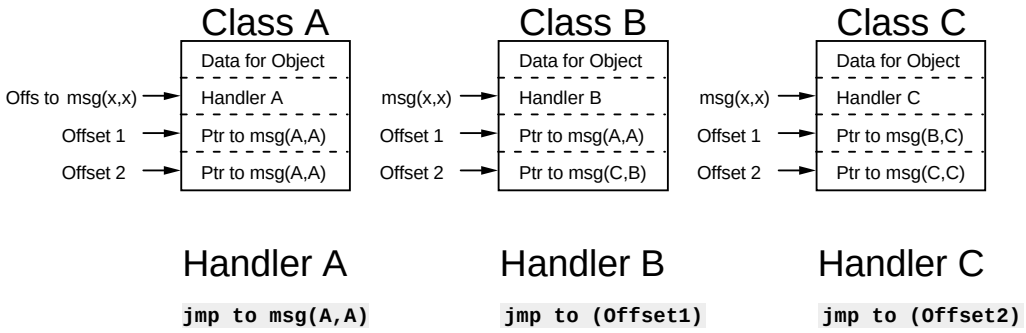


Figura 2.9.: Offset calcolati utilizzando handler ad hoc

Con questa seconda soluzione si risparmia spazio nelle tabelle ed anche tempo di esecuzione: se il primo parametro è A, non è necessario discriminare ulteriormente, ma si può saltare subito al metodo corretto. Non vi è nessun compromesso particolare in questo modo di procedere; l'unica difficoltà è per l'implementatore, dato che individuare le condizioni in cui si può ottimizzare non è proprio banalissimo e richiederebbe un ulteriore studio teorico (se ne dovrebbe venire a capo partizionando il grafo derivato e verificando in quali condizioni un metodo definito su una n-pla non ha ulteriori figli, e quindi ridefinizioni).

In determinati casi è anche possibile ridurre il tempo di dispatching usando gli indici in modo non sequenziale, ossia individuando esplicitamente la sequenza di utilizzo degli indici che permette mediamente di giungere alla risoluzione nel minor numero di passi.

Visto questo, ritorniamo al caso delle tabelle, e verifichiamo in quali casi è possibile

effettuare una disambiguazione statica equivalente a quella che avevamo già visto nel caso unidimensionale.

Per fare questo, consideriamo nuovamente il nostro grafo derivato, e notiamo che, su tutti i nodi (n-ple) di tale grafo, solo alcuni avranno associato effettivamente un metodo. In particolare, sarà possibile certamente individuare, nell'interno di tale grafo, un sottografo costituito dai nodi che hanno una definizione di metodo associata, più tutte le n-ple che da queste discendono. Ora, in questo sottografo sarà possibile identificare delle componenti connesse, indipendenti fra loro. Ebbene, sarà proprio nell'ambito di ciascuna componente connessa che andrà effettuata la disambiguazione. Se consideriamo infatti due componenti separate fra loro, per il modo in cui abbiamo definito il grafo, nessuno dei nodi in una delle due componenti sarà "più specifico" di un altro nodo nell'altra componente, ma saranno, dal punto di vista gerarchico, scorrelate. Di conseguenza, possiamo individuare staticamente a quale delle componenti una determinata invocazione di messaggio farà capo. Analogamente a quanto visto nel caso unario, quindi, anche in questo caso potremo considerare le componenti scorrelate fra loro come messaggi distinti, con una conseguente semplificazione delle tabelle di dispatching.

Facendo riferimento all'esempio di Figura 2.10, se definiamo

```
msg(B,B)
msg(B,C)
msg(C,B)
```

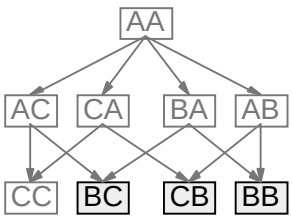


Figura 2.10.: Multimetodi disambiguabili staticamente

scopriremo che nessuno dei metodi è collegato, sul grafo derivato, con gli altri; potremo quindi effettuare la disambiguazione in modo interamente statico, ed il dispatcher non sarà affatto necessario.

Considerare il sottografo del grafo derivato costituito dai nodi con metodi più tutti i figli di questi, ci permette di individuare con grande facilità i casi illegali: se una invocazione non rientra nel sottografo, sarà sicuramente non risolvibile a runtime.

2.5.3.3. Eredità multipla

Tutto quanto detto per il dispatcher vale inalterato anche nel caso dell'eredità multipla, ma occorre porre attenzione all'assegnazione degli offset, e questo vale anche per quanto

riguarda le variabili di istanza.

Come di consueto, un piccolo esempio permetterà di porre subito in luce il problema.

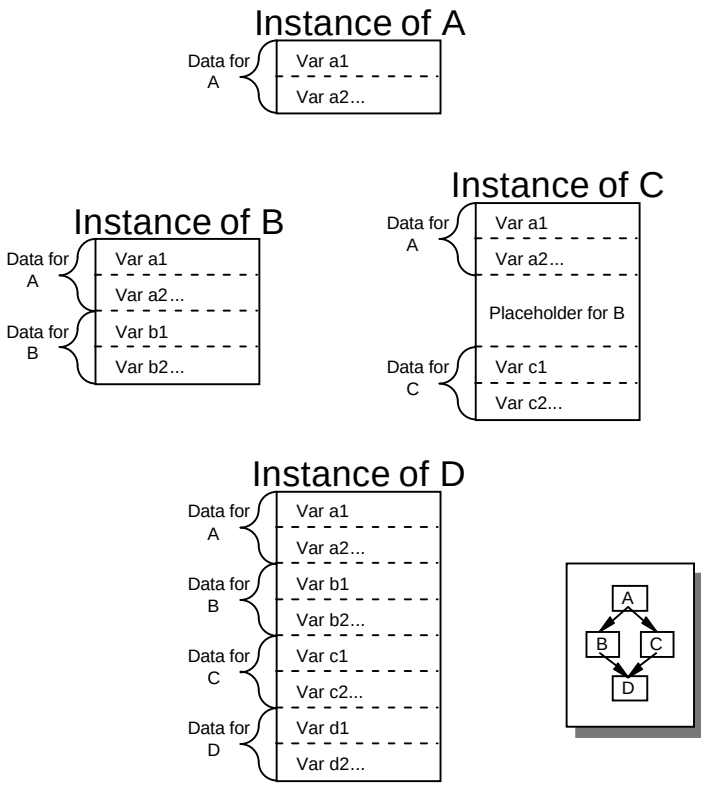


Figura 2.11.: Eredità multipla: allocazione delle variabili di istanza

Come si può vedere in Figura 2.11, nel caso dell'eredità multipla non è possibile assegnare gli offset sui vari rami in modo indipendente. Se così si facesse, le variabili di istanza per B e per C avrebbero offset uguali, e si avrebbe un conflitto. È dunque necessario sacrificare dello spazio in uno dei due rami, al fine di non avere una collisione quando poi le due istanze verranno raggruppate in una sola nella figlia comune. Una risoluzione del tutto simile si può applicare anche per l'analogo conflitto che si genera con i puntatori ai metodi; anche in questo caso, infatti, sarà necessario estendere un po' il record descrittivo per fare spazio agli offset dei messaggi definiti in altri rami, ma che poi saranno utilizzati da sottoclassi della classe in esame.

La soluzione, in sè, non ha particolari complicazioni a parte il consumo di spazio. Ora, mentre questo consumo è abbastanza irrilevante nel caso dei metodi, vuoi perchè i puntatori vengono registrati una volta per tutte nel solo record della classe, vuoi perchè in ogni caso un puntatore occupa comunque una sola parola macchina, nel caso delle variabili di istanza

lo spreco di memoria può risultare significativo. Supponiamo, per esempio, che in B ed in C vengano definiti due distinti vettori di 20 KBytes ciascuno: in tutte le istanze della classe C, seguendo lo schema raffigurato, si avrebbe uno spreco sistematico di 20 KBytes, il chè è poco efficiente.[6]

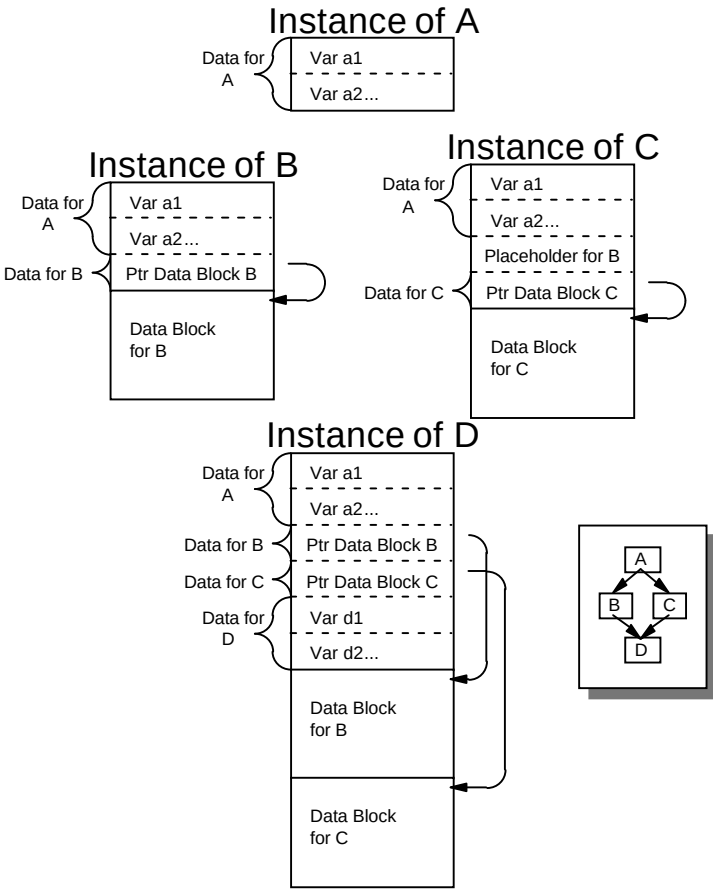


Figura 2.12.: Eredità multipla: variabili di istanza tramite indirezione

Se si desidera limitare lo spreco di spazio, è possibile aggirare il problema, al prezzo di un piccolo rallentamento nell'accesso ai dati del record.

Utilizzando dei puntatori indiretti ai dati che avrebbero offset in conflitto, si può ridurre il consumo di memoria extra al minimo, come mostrato in Figura 2.12. L'unico overhead richiesto è il caricamento del puntatore al blocco dati di B oppure di C tramite una lettura

[6] In C++ il comportamento di default (senza l'utilizzo della keyword "virtual") può portare, se usato per errore, a consumi anche maggiori: B e C avranno allocata solo la memoria necessaria, ma nelle istanze di D ci saranno *due* copie dello spazio per A, copie che verranno inoltre ereditate successivamente anche da tutte le sottoclassi di D.

dalla memoria (una istruzione macchina). Lo spazio sprecato per ogni conflitto, in compenso, si riduce ad un semplice puntatore. È da notare che, per l'ultimo blocco in conflitto, per esempio C, in questo caso, il puntatore non sarà necessario, ma si potrà usare direttamente un normale blocco con offset.

Si potrà scegliere una soluzione oppure l'altra (offset diretti, oppure puntatori) a seconda che si desideri ottenere una maggiore velocità in fase di esecuzione oppure un maggiore risparmio di memoria.

3. Il Linguaggio BOH (Basic Object Handler)

Le considerazioni esposte nei capitoli precedenti hanno portato, in modo naturale, alla definizione di un linguaggio sperimentale, finalizzato alla verifica della praticabilità delle soluzioni proposte. Nella creazione di tale linguaggio, cui è stato dato il nome BOH (Basic Object Handler) si è cercato di utilizzare caratteristiche proposte in diversi linguaggi quali Prolog, Dylan e altri, con la finalità di sintetizzare uno strumento dotato di caratteristiche di flessibilità e versatilità nell'uso, pur mantenendo come obiettivi principali la semplicità complessiva d'utilizzo e l'implementabilità con tecniche che possano portare ad elevate velocità di esecuzioni in runtime. Si è cercato inoltre di mantenere una sintassi che possa risultare familiare a coloro che utilizzano i linguaggi attualmente più diffusi quali C++ e Java, in modo da ridurre i tempi di apprendimento ed offrire un ambiente di programmazione familiare. Naturalmente sono state effettuate anche scelte, per quanto riguarda taluni aspetti sintattici, che rivestono un certo carattere di arbitrarietà, essendo diverse le soluzioni possibili; tutte le scelte compiute, tuttavia, hanno avuto origine in un modo o nell'altro dai criteri di base di semplicità, omogeneità nelle definizioni, efficienza e similitudine con altri linguaggi, quali abbiamo precedentemente esposto.

3.1. Introduzione al linguaggio

Dal punto di vista della classificazione del linguaggio, BOH si qualifica come un linguaggio object-oriented "puro" (gli oggetti sono alla base del linguaggio, ed ogni dato manipolabile è rappresentato sotto forma di oggetto), compilato, fortemente tipizzato con type checking statico, strutturato, orientato all'incapsulamento forte, pensato per supportare una compilazione modulare.

Un tipico programma BOH consiste in uno o più "package", ciascuno dei quali definisce una risorsa o gruppo di risorse in termini di classi e messaggi applicabili a tali classi. Per esempio, nella implementazione di un sistema operativo, un package potrebbe rac-

chiudere le classi e le relative operazioni relative alla grafica, un altro package potrebbe implementare le strutture necessarie per l'accesso alla rete e così via.

Ogni package comprende zero o più definizioni di classi, metodi globali e variabili globali, di cui daremo una descrizione particolareggiata entro breve. Ogni definizione di classe comprende un certo numero di descrizioni di variabili di istanza e di metodi locali.

Le variabili globali saranno utilizzate per tenere traccia di oggetti che possono essere utilizzati in tutto il corpo del package, e che devono godere di caratteristiche di persistenza fra una invocazione di un metodo del package e la successiva.Tali variabili verranno normalmente create all'atto del caricamento del package nel sistema, e distrutte quando il package viene rimosso. L'utilizzo delle variabili globali è limitato al corpo del package, e non può esserne esteso all'esterno.

Come descritto, in luogo dei messaggi convenzionali, useremo in BOH le generic functions, similmente al CLOS, rimanendo però in un'ottica di type checking statico e di tipizzazione forte. In conseguenza di ciò, i messaggi potranno avere un numero arbitrario (finito) di destinatari; la disambiguazione, e l'individuazione quindi del metodo da invocare, avverrà scegliendo dinamicamente, fra tutti i metodi disponibili, quello più specifico applicabile alla combinazione dei parametri del messaggio.

Dal momento che non siamo vincolati ad associare un destinatario esplicito ai messaggi, utilizzeremo perciò i metodi definiti all'interno di una classe per rappresentare le operazioni che è possibile eseguire sulle istanze di tali classi, mentre i metodi globali, descritti al di fuori delle classi, verranno utilizzati per realizzare operazioni non specifiche di determinate classi.

Per migliorare l'incapsulamento, consentiremo ai soli metodi locali di una classe di accedere alle variabili interne delle istanze di tale classe, mentre tutti gli altri non avranno visibilità della struttura interna di tali oggetti, nemmeno quelli descritti in sottoclassi della classe data. Questa scelta ha diversi effetti positivi collaterali: in primo luogo viene risolto il problema sul conflitto fra variabili di istanza omonime nel caso dell'eredità multipla, come descritto nel paragrafo "Variabili di Istanza" (pag. 33). In secondo luogo, ci consentirà di risolvere in modo elegante il problema di presentare una semantica omogenea per l'assegnazione, come vedremo più avanti nel paragrafo "Condivisione di istanze ed effetti collaterali" (pag. 81).

Ricordiamo ancora che, dal momento che ogni dato trattato in BOH è un oggetto appartenente ad una qualche classe, i termini "tipo" e "classe" sono per noi equivalenti. Utilizzeremo inoltre a volte il termine "componenti" per riferirci alle variabili di istanza.

Nel corso di questa presentazione, illustreremo dapprima alcuni esempi di definizio-

ne di classi, che ci consentiranno di introdurre la struttura generale del linguaggio; quindi illustreremo gli aspetti legati alla dichiarazione ed alla inizializzazione delle variabili, successivamente parleremo dei tipi primitivi, delle dichiarazioni delle variabili di istanza, delle strutture di controllo e quindi dei rimanenti aspetti sintattici, mostrando anche le caratteristiche di estendibilità della sintassi stessa. Per concludere, discuteremo dei dettagli legati ai costruttori di nuovi oggetti e di come preservare l'illusione di lavorare con oggetti anche nel caso dei tipi primitivi.

3.2. Un primo esempio

Per dare un'idea dell'aspetto generale di un programma scritto in BOH, vediamo un primo esempio, che, come prassi, stampa sul video il messaggio "Hello, world!":

```
first_example :  uses system
{
 !test()
 {
  println("Hello, world!");
 }
}
```

Come si può vedere, il programma definisce un package "`first_example`", contenente un unico metodo globale "`test`", che stampa la stringa richiesta. Il punto esclamativo indica che il metodo è pubblico, ossia che ne è possibile l'invocazione anche al di fuori del package stesso.

La clausola "`uses`" indica quali sono gli altri package dai quali sono importate le definizioni (messaggi e classi) che verranno utilizzate internamente.

Supponendo di disporre di un ambiente interattivo per effettuare le chiamate, proviamo dunque il funzionamento del nostro package:

```
# test();
Hello, world!
# _
```

3.3. Definiamo le classi

Avendo visto un primo esempio, definiamo ora qualche classe, introducendo così alcuni degli altri elementi chiave del linguaggio BOH.

```
second_package:  uses system
{
//
// first class definition:  glass
//

 !glass :  super object
 {
  !glass():  super object()
  {
  }

  !break(w:glass)
  {
   println("Crash!");
  }
 }
//----------
//
// second class definition:  mattress
//

 !mattress :  super object
 {
  springs:long;

  !mattress(n:long):  super object()
  {
   mattress.springs:=n;
  }

  !break(m:mattress)
  {
   for a:=0; a<m.springs; a:=a+1;
   {
    println("Sproingg!!");
   }
  }
 }
}
```

Tabella 3.1.: Esempio di definizione di classi

L'esempio, illustrato nella Tabella 3.1 (a pag. 56), è un po' lungo, ma il significato è facilmente desumibile: vengono definite due classi, quella dei vetri e quella dei materassi, le cui istanze accettano entrambe il messaggio "break", con le quali si richiede all'oggetto di

rompersi; in conformità all'idea stessa di programmazione object oriented, ciascun oggetto risponde nel proprio modo al messaggio: il vetro stampa "Crash!", mentre il materasso stampa tanti "Sproingg!!" quante sono le molle che contiene.

Vediamo subito un esempio di utilizzo, dopodichè consderemo in dettaglio i nuovi elementi introdotti:

```
# a:=glass();
# b:=mattress(3);
# break(a);
Crash!
# break(b);
Sproingg!!
Sproingg!!
Sproingg!!
# _
```

Analizziamo dunque in dettaglio il programma. Le definizioni delle classi hanno la forma:

```
<classDef>::=<optionalBang> <id> ":" <superList> "{" <classBody> "}"
```

Per ogni classe è possibile specificare uno o più sopraclassi da cui ereditare le caratteristiche (messaggi accettati e componenti). Il punto esclamativo prima del nome della classe specifica che la classe è utilizzabile anche al di fuori del package.

Il corpo della classe è composto nel seguente modo:

```
<classBody>::=<fieldList> <methodList>
```

La dichiarazione[1] dei campi (delle variabili di istanza) di una classe precede tutti i metodi locali della classe; un'esempio è la dichiarazione del numero di molle trovata in "mattress":

```
springs:long;
```

In BOH, l'oggetto dichiarato è in prima posizione e vene seguito da due punti e dalla

[1] Manterremo distinti i termini "dichiarazione" e "definizione", seguendo [Mar93, pag. 51]: una dichiarazione dichiara il tipo di una variabile, mentre una definizione la istanzia, allocandone lo spazio (ed eventualmente inizializzandola).

specificazione del tipo, analogamente a quanto avviene in Pascal.[2] Una descrizione dei possibili tipi per le componenti, e dei tipi primitivi disponibili, verrà data entro breve.

Una delle caratteristiche di BOH è quella di obbligare il programmatore ad utilizzare tipi di dati opachi. Per fare questo, l'accesso alle componenti di una istanza viene concesso esclusivamente all'interno dei metodi definiti all'interno del corpo della definizione di classe. Questo vale in senso stretto: neppure le sottoclassi di una classe data hanno la capacità di utilizzare le componenti definite nelle sopraclassi. Tutti gli accessi a tali componenti devono obbligatoriamente avvenire tramite metodi accessori. Viene garantito in tale modo l'incapsulamento e l'opacità dei dati, svincolando così completamente l'implementazione di una classe, per esempio appartenente ad una libreria, dall'utilizzo che di tale classe verrà poi fatto, sia in termini di utilizzo delle istanze che per quanto riguarda la definizione di sue sottoclassi.

Nell'esempio in questione, per esempio, l'accesso al campo "springs" è possibile solo per i metodi definiti all'interno della definizione della classe "mattress".

L'elenco dei metodi è composto da zero o più definizioni di metodi, che hanno le seguenti forme:

```
<method>::= <methHead> <retVal> <methTail>
```

con

```
<methHead>::= <optionalBang> <id> "(" <paramList> ")"

<retVal>::=
<retVal>::= ":" <id>
<retVal>::= ":" "super" <superCallList>

<methTail>::= "{" <cmdList >"}"
```

La lista dei parametri è composta da zero o più dichiarazioni di parametri, e `cmdList` da zero o più comandi.

Mentre le prime due forme di `retVal` corrispondono a metodi che compiono elaborazioni su dati esistenti, la terza forma (comprendente il token "super") identifica un costruttore.

[2] Cfr. BNF Pascal, spesso elencato alla fine dei manuali del linguaggio, oppure i relativi diagrammi sintattici, come ad esempio in [Met], oppure in [Gro92, pag. 527].

I costruttori si comportano in modo del tutto simile a quanto accade in C++: il metodo invoca i costruttori delle sottoistanze dell'oggetto corrispondenti alle sopraclassi; tali costruttori sono le funzioni elencate in `superCallList`. Ovviamente, il loro numero deve essere uguale ed il tipo corrispondente alla lista `superList`, già vista nell'intestazione della classe.

Nel nostro esempio, mattress è sottoclasse di object; dunque, quando creiamo un nuovo oggetto con il metodo mattress(), la chiamata object() si occuperà di inizializzare propriamente la parte object di un mattress. Il resto del corpo del metodo si occuperà di inizializzare propriamente le componenti definite in questo livello della gerarchia delle classi (in questo caso il campo springs).

I costruttori possono essere più di uno per classe ed avere nome e numero di parametri qualsiasi.[3]

È da notare che l'eventuale valore di ritorno del metodo viene riferito con lo stesso identificatore usato per il metodo, come avviene in Pascal. A differenza di quest'ultimo, però, il valore è utilizzabile anche sul lato destro dell'assegnamento; i messaggi vengono infatti sempre seguiti dall'elenco dei parametri (eventualmente costituito dalla sola coppia di parentesi vuote), mentre il valore di ritorno ne è privo.

3.4. Dichiarazioni e inizializzazioni

Nella definizione del metodo break() di mattress è possibile notare un'altra peculiarità di BOH: la variabile "a" sembra non avere alcuna dichiarazione. In realtà la dichiarazione è costituita dalla assegnazione "a:=0". Durante il parsing del sorgente, infatti, il compilatore tratta la prima assegnazione ad una variabile sconosciuta come una dichiarazione più la relativa inizializzazione.

Naturalmente, questo è possibile grazie al fatto che, utilizzando i meccanismi esposti nei capitoli precedenti, in qualsiasi punto del sorgente siamo in grado di determinare il tipo statico (il tipo più generico) di una espressione, e quindi di ottenere il tipo (più generico) della variabile in corso di dichiarazione.

Il fatto di avere le dichiarazioni legate indissolubilmente alle inizializzazione porta diversi vantaggi: in primo luogo non è possibile avere variabili dichiarate ma non inizializzate[4] (il che costituisce un punto debole di Pascal e C, per esempio), ed in secondo luogo

[3] Per ulteriori informazioni sui costruttori e sulle restrizioni al loro utilizzo, fare riferimento al capitolo 3.14.1.

[4] Non viene offerta analoga garanzia per l'inizializzazione dei campi all'interno di un metodo costruttore; viene lasciata infatti interamente al programmatore la responsabilità di definire la semantica necessaria per l'implementazione e l'inizializzazione interna dell'oggetto e delle sue componenti.

non è necessario dichiarare anticipatamente, come avviene in Pascal, tutte le variabili che verranno usate nel corso della procedura, anche quelle di importanza minore quali indici di ciclo e variabili temporanee. Naturalmente, nulla proibisce di procedere alla dichiarazione anticipata, all'inizio del metodo, delle variabili più importanti, assegnando però al contempo ad esse un valore iniziale opportuno.

Per formulare un esempio concreto, supponiamo di avere nel corpo del sorgente il seguente frammento:

```
a:=object(); // il tipo statico di a è ora object
a:=5;        // legale, long è sottoclasse di object
a:="ciao";   // legale, text è sottoclasse di object
println(a);  // legale solo se println() è
             // definito per object
```

La legalità della chiamata println(a) viene controllata staticamente sulla base del tipo statico di a (quindi object). Dinamicamente verrà invocato il metodo più specifico per println; nell'esempio specifico verrà chiamato println(text).

Le variabili verranno trattate dichiaratamente come riferimenti ad istanze; questo significa che, se incontramo nel sorgente una assegnazione come "a:=b", il significato sarà: "b riferisce ora lo stesso oggetto riferito da a"; in pratica, quindi, l'oggetto materialmente riferito dalle due variabili sarà lo stesso. Questo comportamento può comportare problemi di opacità referenziale, ma l'unica alternativa plausibile sarebbe la duplicazione completa dell'oggetto riferito, eccessivamente penalizzante dal punto di vista delle performance. Analogamente alle variabili verranno trattate le variabili di istanza, quindi sempre come riferimenti agli oggetti relativi, come pure i parametri dei metodi.

Per una discussione sull'argomento, con un confronto con gli altri linguaggi object oriented ed una descrizione del trattamento dei tipi primitivi, si rimanda alla sezione 3.14.2, "Condivisione di istanze ed effetti collaterali".

3.5. Tipi primitivi e costanti

In BOH è disponibile, come in ogni linguaggio, un certo numero di tipi primitivi, che sono elencati nella Tabella 3.2, insieme alle relative definizioni.[5]

[5] La scelta di definire gli interi a 16 bit come "word" deriva da motivi storici: i termini byte, word, long e quad risulteranno certamente familiari a chiunque abbia lavorato con un processore della famiglia 68000, che prevedeva appunto questi come tipi base (vedasi p.es. [Mot92]). In realtà il termine word è normalmente riferito alla lunghezza naturale delle parole macchina di una determinata architettura, ed è quindi

byte	numero intero nel range -128...127 (-2^7...2^7-1)
ubyte	idem, 0...255 (0...2^8-1)
word	idem, -32.768...32.767 (-2^{15}...2^{15}-1)
uword	idem, 0...65.535 (0...2^{16}-1)
long	idem, -2.147.483.648..2.147.483.647 (-2^{31}...2^{31}-1)
ulong	idem, 0...4.294.967.296 (0...2^{32}-1)
quad	idem, -9.223.372.036.854.775.808...9.223.372.036.854.775.807 (-2^{63}...2^{63}-1)
uquad	idem, 0...18.446.744.073.709.551.615 (0...2^{64}-1)
bool	true oppure false
text	testo, oppure stringa di caratteri.
float	floating point in singola precisione (IEEE-754)
double	floating point in doppia precisione (IEEE-754)

Tabella 3.2.: Tipi primitivi disponibili in BOH.

Dal momento che BOH è un linguaggio fortemente tipizzato, ogni costante numerica, o di altro tipo, deve essere riconoscibile come appartenente ad un tipo ben preciso. La caratterizzazione delle costanti avviene tramite suffissi, analogamente a quanto avviene in C, solo in modo un po' più esteso,[6] nel modo che segue. Sia D una cifra qualsiasi compresa fra 0 e 9, e L un segno meno opzionale, allora:

```
<L><D>+         è di tipo long  es:  61342
<D>+"u"         è di tipo ulong es:  3183714311u
<L><D>+"b"      è di tipo byte  es:  34b
<D>+"ub"        è di tipo ubyte es:  34ub
<L><D>+"w"      è di tipo word  es:  -11621w
<D>+"uw"        è di tipo uword es:  34uw
<L><D>+"l"      è di tipo long  es:  -81l
<D>+"ul"        è di tipo ulong es:  63ul
<L><D>+"q"      è di tipo quad  es:  711q
<D>+"uq"        è di tipo uquad es:  3uq

"true"          è di tipo bool
"false"         è di tipo bool
```

dipendente dal processore; noi qui ne fissiamo convenzionalmente la dimensione a 16 bit.

Per lo standard IEEE-754 si può fare riferimento a [Mot93], oppure [Mot94] (disponibili sui siti Motorola). Si considerano i float di 32 bit ed i double di 64 bit. Le dimensioni utilizzate internamente dall'implementazione possono essere diverse, ma devono essere rispettate le caratteristiche minime dello standard IEEE.

[6] In C una costante come 4L è definita di tipo long, per esempio, una come 42112U è unsigned, la costante 4.2f5 è un float e la costante 4.2e5 è un double. ([KR89])

```
<L><D>+"."<D>+            è di tipo double  es:  3.25

<L><D>+"."<D>+"f"         è di tipo float   es:  -2.4f
<L><D>+"f"                è di tipo float   es:  61f
<L><D>+"f"<L><D>+         è di tipo float   es:  4f-11
<L><D>+"."<D>+"f"<L><D>+  è di tipo float   es:  3.6f9

<L><D>+"."<D>+"e"         è di tipo double  es:  -2.4e
<L><D>+"e"                è di tipo double  es:  61e
<L><D>+"e"<L><D>+         è di tipo double  es:  4e-11
<L><D>+"."<D>+"e"<L><D>+  è di tipo double  es:  3.6e9
```

Oltre a queste, sono disponibili alcune sintassi alternative per i numeri floating point, che verranno descritte più avanti, nel capitolo 3.11, "Facilitazioni Sintattiche".

Gli oggetti di tipo text possono memorizzare una sequenza di caratteri di lunghezza potenzialmente illimitata (i limiti sono dettati solo dall'implementazione). Nessuna assunzione viene fatta sul formato di memorizzazione interno degli oggetti text che potrebbero contenere anche caratteri Unicode, informazioni sulla direzione di scrittura, attributi del testo od altro. Le uniche caratteristiche note sono quelle definite dalle funzioni di libreria: essere stampabili, avere un modo per leggere da console una parte di testo, poter concatenare più oggetti di testo e poche altre operazioni.

Nell'ambito del codice sorgente, un testo viene identificato dalla porzione di sorgente comprese fra doppie virgolette ("). Due doppie virgolette consecutive all'interno di un testo ne indicano una sola, come in Pascal. Nulla impedisce, disponendo di un editor capace di gestire più sistemi di scrittura diversi, di inserire porzioni di testo non latino dentro una stringa direttamente all'interno del sorgente.

È opportuno notare che il testo incluso in un sorgente costituisce l'unica parte case sensitive di un programma BOH; tutto il resto del codice, infatti, viene trattato dal compilatore come case insensitive.

3.6. Gli identificatori

Per quanto riguarda gli identificatori, i caratteri validi rientrano in un range decisamente più esteso rispetto a quello dei normali linguaggi; un identificatore può essere infatti composto da una sequenza qualsiasi di:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789‘~|\/?><+=_-*&^%$#@!
```

a patto che il primo carattere non sia numerico. (NB: l’Implementazione preliminare ha alcune limitazioni, derivanti dall’algoritmo usato per il parsing lessicale. Consultare la sezione 4.2.3 per ulteriori informazioni)

È quindi perfettamente legale definire un metodo di questo genere:

```
!  ~B%#@@$?  ():long
{
 ~B%#@@$?  := 15;
}
```

Dal punto di vista del linguaggio, anche gli operatori infissi come "+", "-" e simili sono considerati come identificatori; un algoritmo apposito provvede al parsing, ed alla trasformazione interna degli operatori infissi in normali chiamate di funzione.

Proprio a questo riguardo, una delle feature più interessanti di BOH consiste nella possibilità data all’utente di estendere liberamente il linguaggio con i propri operatori infissi, utilizzando identificatori composti da tutti i caratteri sopra indicati; tale caratteristica di BOH ricorda molto quella equivalente, ed assai comoda, disponibile in Prolog ([Proa],[Prob]). Una descrizione esauriente di tutto il meccanismo verrà fatta fra poco, nella sezione 3.12, “Gli operatori”.

3.7. Componenti di una istanza

Per ogni livello della gerarchia delle classi, potranno essere definite una o più componenti (variabili di istanza), con la seguente sintassi:

```
<field>::=<idList> ":" <fieldType>
```

idList è una sequenza di uno o più identificatori separati da virgola; fieldType è definito in questo modo:

```
<fieldType>::=<id>
<fieldType>::=<id><indexList>

<indexList>::=<index>
<indexLst>::=<indexList><index>
```

```
<index>::="[" <numLong> "]"
<index>::="[" <numLong> ".." <numLong> "]"
```

Come si può vedere, una componente di una classe (di tutte le sue istanze) può essere un oggetto, oppure un vettore (anche multidimensionale) di oggetti. Per ogni indice è possibile indicare il numero di elementi complessivi (ed in tal caso gli elementi saranno indicizzati da 0 a n-1, come in C), oppure specificando sia lower che upper bound, alla maniera del Pascal.

È da notare che le componenti sono l'unico elemento di BOH dove sia possibile definire dei vettori; non è possibile definire come vettori variabili globali oppure locali. Questo ha una motivazione nel fatto che specificare un numero massimo di elementi per un qualche tipo di dato crea una dipendenza indesiderabile dall'implementazione della classe. Se l'indice massimo del vettore, ed ogni suo accesso, è incapsulato viceversa dentro una classe, risulta molto più agevole sostituire l'implementazione in qualsiasi momento con un vettore di dimensioni massime maggiori, o con un altro tipo di dato (alberi, liste etc.). Per esempio, Java, per ovviare allo stesso problema, tratta i vettori dinamicamente, permettendo di variarne a runtime il numero di elementi, e gestendoli come oggetti. Quest'approccio, anche se formalmente elegante, ha una penalità in termini di performance: i vettori dovranno essere trattati internamente come liste, oppure dovrà essere gestita la copia dell'intero vettore ogni volta che se ne varia la dimensione. In BOH, di converso, i vettori sono "veri" vettori di parole macchina: la dimensione occupata in memoria sarà pari tipicamente al numero di elementi per la dimensione di un puntatore. L'efficienza non viene però sacrificata all'eleganza di programmazione: incapsulando forzatamente i vettori dentro oggetti, l'indipendenza dalla realizzazione del tipo di dato viene preservata.

Laddove si definirebbe normalmente una variabile globale come vettore, in BOH si dovrà prima definire un tipo di dato adatto (una classe), e poi la variabile globale diverrà una istanza di tale classe.

Se poi si desiderasse avere un oggetto con lo stesso comportamento di vettori di dimensione variabile a runtime, sarà sempre possibile definire banalmente un tipo di dato astratto utilizzando liste o simili.

Per quanto riguarda l'accesso alle parti di una istanza appartenenti alle sopraclassi, questo viene effettuato utilizzando la pseudovariabile "super" che identifica, appunto, la sottoistanza corrispondente alla sopraclasse della classe in fase di definizione. L'invocazione dei metodi della sopraclasse avviene quindi automaticamente utilizzando "super" come parametro di una qualsiasi invocazione. Se la classe in corso di definizione possiede

più di una sopraclasse, le corrispondenti sottoistanze sono identificabili con super:id, dove id è la sopraclasse cui si fa riferimento.

In BOH non sono infine disponibili variabili di classe (cioè condivise fra tutte le istanze). Una funzionalità simile si può ottenere utilizzando le variabili globali del package.

3.8. I parametri

La sintassi dell'elenco dei parametri è la seguente:

```
<paramList>::=
<paramList>::=<idList> ":" <id>
```

dove `idList`, come già accennato, è un elenco di uno o più identificatori separati da virgola, e `id` è l'identificatore di un nome di classe.

Come già accennato, i parametri vengono sempre trattati come riferimenti agli oggetti passati; dal punto di vista implementativo, potranno essere passati puntatori alle istanze oppure direttamente i dati nel caso dei tipi semplici; la tecnica utilizzata per mantenere la simmetria nei due casi è illustrata più avanti, nella sezione 3.14.2, "Condivisione di istanze ed effetti collaterali".

3.9. Contesti annidati

Anche in BOH, come in C e altri linguaggi con sintassi simii, è possibile creare dei contesti, o blocchi, ossia delle zone di programma dotate di proprie variabili locali private, trattabili alla stessa stregua di una istruzione singola.

Questa è la definizione sintattica:

```
<context>::= "{" <cmdList> "}"
```

I contesti sono annidabili a piacere; ad ogni livello potrà essere definito un insieme di variabili locali che risulteranno non definite al di fuori del contesto in questione. Ecco un esempio:

```
a:=5;        // supponiamo sia l'unica var.  definita.
{
 b:=a;       // b è locale di questo contesto
 {
```

```
 c:="ciao"; // c viene definita e subito dimenticata
}
// x:=c;    // darebbe errore!  c non è definita, qui.
{
 c:=811.23; // una variabile distinta dalla precedente
}
}            // b e c non sono più utilizzabili.
```

Curiosamente, per via del modo in cui le variabili sono dichiarate e subito inizializzate in BOH, non è possibile definire variabili locali, in un contesto, che mascherino variabili locali definite precedentemente. Infatti, consideriamo:

```
{
 x:=espressione;
 ...
```

I casi sono due: o la variabile "x" è già stata definita, ed allora si tratta di una assegnazione ordinaria, oppure "x" non è ancora stata definita, e si tratta quindi di una dichiarazione con relativa inizializzazione.Questa non è in alcun modo una limitazione, ma anzi contribuisce ad evitare potenziali condizioni di errore.

Il meccanismo per cui una variabile più interna maschera una variabile di nome uguale più esterna, infatti, pur se elegante da un punto di vista concettuale, ha una utilità pratica alquanto dubbia, e viene utilizzato in realtà assai di rado, appunto per via dello sforzo mnemonico richiesto al programmatore e dei possibili errori che ne potrebbero derivare.[7]

I contesti sono utilizzabili liberamente ogni qual volta si desideri introdurre una zona isolata di programma dotata di proprie variabili locali; il loro utilizzo più naturale è comunque nell'ambito delle strutture di controllo, descritte di seguito.

3.10. Le strutture di controllo

Gli strumenti per controllare il flusso dell'esecuzione del programma sono quelle tradizionali della programmazione strutturata. Vediamo sintassi ed utilizzo di quelle disponibili in BOH:

[7] Il compilatore gcc, p.es., permette di attivare, fra i controlli in fase di compilazione, un warning se una variabile locale ne maschera una più esterna (flag -Wshadow), appunto perchè tale utilizzo potrebbe portare ad errori ([gcc]).

3.10.1. While

ll ciclo while ha la seguente struttura:

```
<cmd>::= "while" <expr> "{" <cmdList> "}"
```

La sequenza di comandi cmdList viene eseguita sintantochè l'espressione non diventa falsa. La valutazione dell'espressione viene eseguita prima di ogni iterazione; se la condizione è inizialmente falsa, i comandi elencati non vengono mai eseguiti. L'espressione expr deve essere di tipo statico bool. La parte compresa fra "{" e "}" è un contesto, ed è quindi possibile definire delle variabili temporanee al suo interno.

È possibile anche utilizzare la seguente sintassi alternativa:

```
<cmd>::= "do" "{" <cmdList> "}" "while" <expr> ";"
```

In questa seconda forma, la sequenza cmdList viene sempre eseguita almeno una volta.

3.10.2. If...elseif...else

Questa è la definizione dell'istruzione di scelta:

```
<cmd>::= <ifHead>
<cmd>::= <ifHead> else "{" <cmdList> "}"

<ifHead>::= "if" <expr> "{" <cmdList> "}"
<ifHead>::= <ifHead> "elsif" <expr> "{" <cmdList> "}"
```

Se la valutazione dell'espressione booleana che segue "if" dà risultato true, allora viene eseguito il contesto subito seguente, altrimenti vengono valutate, in seguenza, le diverse espressioni "elsif" (se presenti), e viene eseguito unicamente il contesto che segue l'espressione vera. In caso nessuna delle espressioni sia vera, viene eseguito il solo contesto "else", se presente.

3.10.3. For

La definizione del costrutto for è:

```
<cmd>::= "for" <cmd> <expr> ";" <cmd> "{" <cmdList> "}"
```

Analogamente al C, il primo comando viene eseguito una volta all'inizio del ciclo, e sarà utilizzato normalmente per l'inizializzazione. Viene valutata quindi l'espressione; se è falsa si esce direttamente, altrimenti viene eseguita cmdList, quindi il secondo comando, usato di norma per incrementi di indici od affini, e viene ripetuta la valutazione.

Da notarsi, nella definizione sintattica, l'uno del punto e virgola: tale segno viene utilizzato per concludere una istruzione semplice, e viene quindi considerato parte del comando. Per esempio:

```
<cmd>::= <id> ":=" <expr> ";"
```

Esempi di utilizzo del ciclo for:

```
for a:=0; a<10; a:=a+1;
{ println(a);}

for {a:=0;b:=2;} a<10; {a:=a+1;b:=b*2;}
{ println(b);}
```

3.10.4. Case

La sintassi di questo costrutto è la seguente:

```
<cmd>::=<caseStruct>

<caseStruct>::=<caseBody> "}"

<caseBody>::=<caseHead>
<caseBody>::=<caseBody> <exprList> ":" <cmd>
<caseBody>::=<caseBody> "default" ":" <cmd>
<caseHead>::= "case" <expr> "{"
```

Il comportamento è identico a quello della analoga struttura C: viene eseguito solamente il comando associato al valore assunto dall'espressione expr al momento del test.

L'espressione può essere di tipo arbitrario; il test di uguaglianza viene effettuato usando il metodo di libreria `=(object,object):bool` (descritto più avanti).

3.10.5. Altri costrutti

In BOH, perlomeno allo stato attuale di definizione del linguaggio, non è presente un costrutto tipo "break", per forzare uscite anticipate da cicli. Questo significa che è impossibile, a meno di non simularle con variabili logiche, scrivere strutture con più condizioni di uscita;[8] tali strutture sono comunque di uso assai infrequente, ed è possibile comunque simularle in altro modo. È possibile comunque che una simile opzione venga aggiunta durante la definizione di nuove specifiche per il linguaggio.

Non sono previsti salti non condizionati (goto), nè tantomeno etichette associate alle istruzioni.

[8] Ad es. i costrutti di tipo Omega ed RE; tali forme sono impossibili da realizzare utilizzando solo le strutture di base della programmazione strutturata (per esempio, non sono realizzabili in Pascal), a meno di non simularle utilizzando delle variabili logiche. Le strutture sono state citate, p.es. nell'Esame di Teoria e Applicazioni delle Macchine Calcolatrici, Claudio Mirolo e Cinzia Costantini, Univ. di Udine, anno 1987/88

3.11. Facilitazioni sintattiche

Anche nel linguaggio BOH, come in quasi tutti i linguaggi di programmazione, sono disponibili strumenti di tipo sintattico che hanno lo scopo di rendere più comprensibile il codice e di semplificare la notazione.

Questi strumenti sono talora indicati in letteratura come “zucchero sintattico”, in quanto non aggiungono effettive funzionalità al linguaggio, ma sono solo una comodità offerta al programmatore per rendere il codice più leggibile.

Vediamo dunque cosa il linguaggio mette a disposizione su questo fronte.

3.11.1. I commenti

È possibile rendere più comprensibile il codice aggiungendo al sorgente delle annotazioni, che verrano comunque ignorate dal compilatore.

I commenti posono essere di due tipi: su singola linea, oppure blocchi di commenti.

Su ogni linea, viene ignorato dal compilatore tutto quanto segue i caratteri "//" (a meno che questi non compaiano dentro una stringa). Esempio:

```
a:=5;   // il valore di a è ora 5.
```

I blocchi di commenti, invece, sono tutto quanto è racchiuso fra le sequenze "/*" e "*/". Diversamente da altri linguaggi, in BOH i blocchi di commenti possono essere annidati a piacere; questo semplifica il commentare parti di codice a fini di debug, quando altri commenti siano già presenti. Es:

```
a:=5;
/*
a:=4;   /* ora a vale 4 */
*/
// in realtà vale 5.
```

3.11.2. L’utilizzo del punto

Il particolare utilizzo del punto è una delle caratteristiche peculiari del linguaggio, ed è ispirato dall’uso che ne viene fatto in Dylan ([App95], [Sha97], [Uni]).

In BOH l’utilizzo del punto è perfettamente equivalente, a livello sintattico, all’applicazione di una coppia di parentesi, secondo il seguente criterio:

```
.x              è equivalente a x()
a.x             è equivalente a x(a)
a.x()           è equivalente a x(a)
a.x(b,c,...)    è equivalente a x(a,b,c,...)
```

Questo vale nel caso di chiamate di messaggi, di accesso a componenti di oggetti, e persino di numeri con punto decimale.

Per esempio, invece di

```
println("Ciao!");
```

come abbiamo scritto sinora, è perfettamente legale scrivere:

```
"Ciao!".println;
```

Similmente, riferendosi ad uno dei primi esempi, avremmo potuto scrivere anche:

```
springs(a)
```

per riferirci al campo a.springs.

Infine, anche se la cosa è di scarsa utilità, il numero 4.25, per esempio, può essere scritto come 25(4), ed il numero 6.3e-8 come 3e-8(6).

L'implementazione sperimentale, descritta più avanti, è perfettamente in grado di compilare codice che segua questa sintassi.

Al di là del fattore estetico, questo tipo di notazione riveste anche notevoli aspetti pratici.

Per esempio, un utilizzatore abituale di SmallTalk troverà qualcosa di familiare scoprendo che 4+5 può essere scritto anche come 4.+(5); secondo questa notazione, infatti, l'utilizzo delle generic functions ricorda molto quello dei linguaggi orientati agli oggetti dove i messaggi devono essere obbligatoriamente inviati ad un destinatario. La scrittura:

```
myWindow.close();
```

si può perfettamente interpretare come la spedizione del messaggio close() all'oggetto myWindow, ed in effetti è così, con una sintassi molto più familiare per coloro che hanno programmato in C++ oppure in Java.

Analogamente, il codice:

```
myWindow.move(30,80);
```

è di aspetto assolutamente identico a quello che avrebbe in un linguaggio tradizionale. In realtà la disambiguazione avviene su tutti e tre i parametri, anzichè solo sul primo, ma, se lo si desidera, un uso accorto delle definizioni di metodi[9] permette di ottenere risultati equivalenti.

Sotto questa luce, l'uso delle generic functions non rappresenta certo un limite, od un utilizzo "impuro" della metodologia object oriented, ma anzi ne rappresenta una realizzazione efficace e, come visto, anche funzionale.

Per ritornare all'aspetto sintattico, l'utilizzo del punto secondo queste modalità consente anche di rendere il codice più chiaro nel caso di molte operazioni annidate. Le due scritture che seguono sono equivalenti:

```
annidate:       quattro(tre(due(uno(a,b),c,d)),e);

uso del punto:  a.uno(b).due(c,d).tre.quattro(e);
```

Le operazioni eseguite sono le medesime, ma la seconda forma è più semplice, e rende meglio l'idea di operazioni successive applicate in sequenza.

Un vantaggio aggiuntivo è che, seguendo questa sintassi, BOH potrebbe essere utilizzato anche in modo interattivo come linguaggio di scripting. Per esempio, confrontiamo le seguenti:

```
Unix:   cat testo | cut -c1-5 | grep "pattern" | more

BOH:    testo.cutc(1,5).grep("pattern").more;
```

Come si può vedere, l'aspetto delle due righe è piuttosto simile; quello che nella shell Unix viene fatto con i file può essere fatto in BOH con gli oggetti di tipo testo (o di altro tipo). Non è quindi improbabile ipotizzare una shell basata su BOH che permetta la manipolazione di oggetti in modo interattivo.

[9] Se per un messaggio dato viene definito un unico metodo per ciascun tipo del primo parametro, ad esempio, il comportamento è indistinguibile da quello di un linguaggio che preveda destinatari espliciti.

3.12. Gli operatori

Una delle caratteristiche peculiari di BOH è la gestione degli operatori. Difatti, mentre nella maggior parte dei linguaggi gli operatori infissi sono definiti staticamente a livello di specifica sintattica, e non è possibile aggiungerne altri, nel linguaggio BOH la definizione degli operatori è una caratteristica modificabile dall'utente.[10] Agli operatori di sistema (importati insieme al package system), è possibile aggiungerne altri, con livelli di priorità ed associatività a piacere, in modo da rendere la notazione assai leggibile e comprensibile.

Facciamo un esempio: se desideriamo definire un operatore infisso chiamato %, che associa a sinistra, di priorità superiore a "+" e "-", ma inferiore a "*" e "/", potremo scrivere:

```
operator x % xx 450;
```

Fatto ciò, nel corpo del package potremo scrivere:

```
a:=5+4%6*9%8;
```

ed il compilatore si preoccuperà di trasformare internamente il comando in:

```
a:=+(5,%(4,%(*(6,9),8));
```

Vediamo la cosa un po' più formalmente. La sintassi della definizione di nuovi operatori è:

```
<operDef>::="operator" "x" <id> "x" <numLong> ";"
<operDef>::="operator" "x" <id> "xx" <numLong> ";"
<operDef>::="operator" "xx" <id> "x" <numLong> ";"
<operDef>::="operator" <id> "x" <numLong> ";"
<operDef>::="operator" "x" <id> <numLong> ";"
```

Le dichiarazioni di operatore devono obbligatoriamente apparire subito all'inizio della definizione del package, prima di ogni altra classe. Il numero identifica la priorità: numeri più alti indicano operatori che legano più fortemente; Le priorità suggerite degli operatori più frequentemente, importati dal package system, sono indicate nella sezione 3.13, "Funzioni di libreria".

[10] Questa funzionalità discende direttamente da quella equivalente disponibile in Prolog, dove peraltro, data la particolare natura del linguaggio, è usata abbastanza saltuariamente. Cfr. [Proa] e [Prob].

La notazione `x id x` indica un operatore non associativo (es. ":="); `x id xx` un operatore che associa a sinistra, ossia `a id b id c` diventa `id(a,id(b,c))` ; `xx id x` un operatore che associa a destra, ossia `a id b id c` diventa `id(id(a,b),c)` ; `id x` un operatore monadico prefisso, ed infine `x id` un operatore monadico postfisso. Il livello di priorità può essere qualsiasi numero compreso fra 1 e 999.

È da sottolineare che gli operatori possono essere degli identificatori generici, i quali possono essere composti da caratteri non solo alfanumerici; si possono quindi definire operatori chiamati "adatto?", "+-quasi", "miXed", "!$*??o91@#" e così via.

Per dare un'idea delle capacità del parser di BOH, l'implementazione realizzata per collaudare il linguaggio riesce a convertire correttamente il codice:

```
operator x leone x 150;
operator a+ww x 150;
operator x *rrww 55;
operator x +j x 800;
operator x -j x 800;
operator xx + x 300;
operator xx - x 300;
operator xx * x 500;
operator xx / x 500;
operator - x 750;

 tok:=4+5*3--6/2.5e-6+-y*rrwwleonex*rrww;
```

nel seguente:

```
 tok:=*rrww(leone(*rrww(+(-(+(4,*(5,3)),/(-(6),5e-(6)(2))),-(y))),x));
```

Più concretamente, vediamo un esempio pratico, in cui degli operatori definiti dall'utente sono utilizzati per costruire i numeri complessi, conservando però la notazione utilizzata in matematica: a+jb.[11]

[11] A voler essere precisi, i matematici usano molto più frequentemente a+ib, mentre l'uso di "j" per indicare l'unità immaginaria è più frequente in ingegneria. Usare l'uno o l'altro è equivalente ai fini dell'esempio.

Consideriamo il seguente frammento di codice:

```
operator x +j x 650;

class complex:  super num
{
 re,im:double;

 +j(r,i:double):  super num()
 {
  +j.re:=r;
  +j.im:=i;
 }
```

In virtù di questa definizione, è ora possibile scrivere direttamente dentro al codice espressioni come `5.2+j6.4`, oppure `-3e-8+j2.25e6`.

Un esempio più completo di definizione dei complessi è disponibile in appendice. Sulla base di tale definizione è possibile scrivere, ad esempio:

```
println(3.0+j1.24 + 4.21-j7.11);
```

Questa notazione si fa chiaramente apprezzare per chiarezza e comodità.

3.13. Funzioni di libreria

Come in quasi tutti i linguaggi di recente concezione, anche nel nostro caso considereremo le chiamate utilizzate per l'I/O e per la maggior parte delle operazioni non come parte integrante del linguaggio ma come features esterne, eventualmente sostituibili e modificabili a piacere, in modo simile a quanto fanno, ad esempio, C e Java.

In particolare, laddove in C tutte le operazioni su stringhe, le funzioni di I/O, le operazioni sulla memoria, gli accessi al sistema operativo e così via sono disponibili in librerie separate, di cui viene inclusa una definizione (con la direttiva #include) nel codice sorgente, i linguaggi più recenti utilizzano un meccanismo di interfacciamento con le librerie più netto, come nel caso di Java, tramite la keyword "import", ed analogamente per Modula, Oberon, Turbo Pascal, Ada etc.

Avendo definito nel linguaggio la possibilità di avere packages separati, utilizzeremo un

package standard, denominato “system” per racchiudere le funzioni, e le classi, di utilizzo più comune.

Di seguito proporremo di conseguenza una gerarchia minima di classi da supportare, con i messaggi relativi, in modo da offrire un supporto di base all’utilizzatore del linguaggio. Naturalmente, quanto segue è passibile di modifiche e revisioni, essendo da considerarsi nulla più che una proposta di nucleo di funzionalità di libreria da mettere a disposizione.

3.13.1. Le classi standard

In Figura 3.1 compare uno schema delle classi standard che potrebbero essere supportate, al minimo, dal package system:

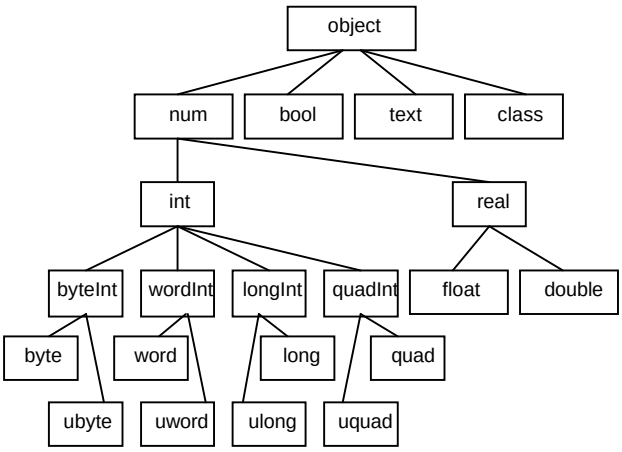


Figura 3.1.: Classi standard per il linguaggio

Le classi num, int, real, byteInt, wordInt, longInt e quadInt sono solo classi che definiscono proprietà comuni per le varie classi di numeri. Uno schema dei messaggi accettati da ciascuna classe compare poco più avanti nel testo.

3.13.2. Le operazioni matematiche

Il package system dovrà implementare almeno le quattro operazioni base ("+", "-", "*", "/") su tutti i tipi base numerici. Deve essere disponibile un modo per convertire i numeri di un tipo in un’altro, tipicamente tramite un messaggio con lo stesso nome del tipo di destinazione (similmente al cast C); per es.: long(5.2) restituirà 5L. Sui reali saranno definite trunc

e round, nonchè logaritmi e funzioni trigonometriche principali. Le operazioni aritmetiche dovranno accettare anche argomenti di tipo diverso, ed effettuare le necessarie conversioni ove necessario. Per il passaggio da un tipo intero con più bit ad un tipo con meno bit dovranno essere disponibili funzioni per estrarre la parte alta e bassa del valore binario (es: word alta e word bassa di un long, oppure byte alto e byte basso di una word e così via). Occorreranno gli shift sugli interi (sia logici che aritmetici[12]), incrementi e decrementi.

3.13.3. Relazioni logiche

Sui dati numerici sono applicabili le relazioni "<", "<=", ">", ">=", "=" e "<>", con l'usuale significato matematico. Tutti questi operatori restituiscono valori di tipo bool.

3.13.4. Operazioni booleane

Le operazioni sono le usuali and, or, not, applicabili sia a booleani che ad interi considerati come campi di bit. Non includiamo nella libreria standard, per il momento, ulteriori operazioni logiche (xor, eqv etc.).

3.13.5. Gli operatori standard

Questi sono gli operatori che dovranno essere supportati, con le relative priorità:

```
operator xx + x 300;
operator xx - x 300;
operator xx * x 500;
operator xx / x 500;
operator - x 750;
operator x < x 200;
operator x > x 200;
operator x = x 200;
operator x <= x 200;
operator x >= x 200;
operator x <> x 200;
operator x and x 150;
operator x or x 130;
operator not x 170;
```

[12] Nello shift logico, il dato viene considerato come un campo di bit; nelle posizioni libere entrano bit a zero; nello shift aritmetico, il dato viene considerato come un intero dotato di segno; se viene applicato uno shift a destra ad un numero negatvo, quindi, entreranno dei bit a uno nelle posizioni più a sinistra, in modo da preservare il segno.

Assegniamo al punto priorità 700 nel caso degli identificatori (ossia id.id(abc)), mentre lo consideriamo superiore a quella di ogni altro operatore nel caso di numeri floating point.[13]

3.13.6. Le operazioni di I/O

In via provvisoria, definiamo le operazioni print(object) (stampa di un oggetto, numero o stringa), println(object) (print line, viene fatto seguire alla stampa un new line), nl() (ritorno a capo). Sono inoltre definite readLong(), readText() e così via (lettura da console), con comportamento affine a quello della read Pascal.

Non definiamo, al momento, una modalità di stampa formattata.

3.13.7. Altre funzioni

È definito il test di uguaglianza fra due oggetti generici `=(object,object):bool`, col significato di uguaglianza fra puntatori (o di valore uguale nel caso dei tipi primitivi). Inoltre, la funzione class(object):class restituirà il puntatore alla classe di appartenenza di un determinato oggetto.

Mettiamo a disposizione, provvisoriamente, una funzione panic(text), che interrompe l'esecuzione del codice, visualizzando il testo fornito come parametro. Una realizzazione più completa dovrà contemplare invece, in caso di errore, la possibilità di inviare e gestire eccezioni.

3.13.8. Elenco delle chiamate

Queste sono le chiamate minime generali; tipicamente, le varie sottoclassi implementeranno chiamate apposite per i casi più specifici (es: ci sarà un +(word,word):word, per esempio, e così dovranno essere implementati metodi specifici per le operazioni matematiche e logiche su tipi diversi, purchè numerici). Per le trasformazioni in text di oggetti generici come object oppure num, i metodi text, print e println di default, definiti su object, provvederanno a generare una stringa generica indicante la classe di appartenenza dell'oggetto (es: "<object>"). Il comportamento delle chiamate artmetiche su lle classi astratte num, real e int non è al momento definito; un comportamento possibile sarebbe la restituzione, da parte di tutte le funzioni, di una medesima istanza generica (vuota) della classe.

[13] Possiamo sempre distinguere il caso di un numero floating point, dato che abbiamo richiesto che il primo carattere di un identificatore non sia numerico.

L'elenco riassuntivo delle chiamate minime da supportare, basate su quanto descritto, è riportato in appendice, nella sezione A.2.

3.14. Ulteriori considerazioni

3.14.1. Garantire la coerenza: il caso dei costruttori

Come abbiamo visto precedentemente, alcune semplici regole ci permettono di costruire un sistema di tipi coerente e controllabile staticamente. Richiamiamole qui, semplificandole per motivi di chiarezza.

Per rendere coerente i tipi dei valori di ritorno, una regola asseriva una generalizzazione di:

- Se due metodi hanno uguale identificatore ed arità, allora corrispondono al medesimo messaggio, e le definizioni devono essere coerenti, cioè se i due metodi sono fun(A1,B1):C1 e fun(A2,B2):C2, e si ha A1 sottoclasse o uguale a A2 e B1 sottoclasse o uguale a B2, ALLORA anche C1 deve essere sottoclasse od uguale a C2.

Ossia, dato un metodo fun(A1,B1):C1, ogni metodo che descrive un caso particolare, più specifico, di tale metodo, deve restituire un valore di ritorno che sia anch'esso una specializzazione del valore di ritorno del primo metodo considerato. L'applicazione della regola ci garantisce che, per ogni combinazione di parametri fornita al messaggio omonimo, sarò in grado di descrivere il valore di ritorno più generico restituito. Più in generale, saremo quindi in grado di stabilire staticamente il tipo di ritorno più generico restituito da una espressione comunque formata .

Un'altra regola garantiva invece che in run time troveremo uno ed un solo metodo più specifico da chiamare per ogni combinazione di tipi effettivi, dinamici, dei parametri di un messaggio, ed era il caso generale di:

- Se in due metodi corrispondenti allo stesso messaggio fun(A1,B1) e fun(A2,B2) si verifica A1 sottoclasse di A2 e B1 sopraclasse di B2, ALLORA deve essere definito anche il metodo fun(A1,B2), pena un errore in fase di compilazione.

Viene perciò imposto di risolvere esplicitamente i casi di ambiguità nelle chiamate, richiedendo una descrizione completa del comportamento che, in corrispondenza di un messaggio, si ha per ogni combinazione di tipi a runtime.

Fin qui tutto bene per i metodi ordinari. Nel caso dei costruttori, tuttavia, le cose si complicano un poco. Supponiamo, infatti, di avere un costruttore per una nuova classe che abbiamo appena definito:

```
!constructorSubClass():  super constructorSuperClass() {...
```

In questo frangente, viene inviato, per l'inizializzazione della sopraclasse, il messaggio constructorSuperClass(). A noi interessa, ovviamente, che il costruttore restituisca un oggetto del tipo corrispondente alla sopraclasse, ma non desideriamo per nessun motivo che l'istanza restituita appartenga ad una sua ulteriore sottoclasse: questo equivarrebbe a dire, infatti, che la sottoistanza corrispondente alla sopraclasse viene inizializzata con qualcosa di diverso da una istanza appartenente strettamente alla sopraclasse stessa.

In questo senso, i costruttori rappresentano un caso particolare rispetto alla filosofia generale del linguaggio, che richiede che in ogni punto ove appare una istanza di una classe possa apparire anche una istanza di una sottoclasse: il valore restituito dal costruttore chiamato deve essere tassativamente del tipo richiesto, e mai di una sottoclasse.

A questo fine, è necessario introdurre delle ulteriori regole, per permetterci di individuare staticamente non solo il tipo più generico di una espressione, ma anche il tipo più specifico, nel caso si tratti di un costruttore; questo ci permetterà di individuare esattamente il tipo restituito.

Questo è un insieme di regole aggiuntive adatto a questo scopo:

- Dati due metodi che si riferiscono allo stesso messaggio, se sono definiti un costruttore costr(A,B):C ed un metodo ordinario fun(A1,B1):C1, ALLORA non devono essere contemporaneamente A1 sottoclasse od uguale di A e B1 sottoclasse od uguale a B.

- Se sono definiti costr(A,B):C e costr(A1,B1):C1, con A1 sottoclasse od uguale di A e B1 sottoclasse od uguale a B, ALLORA C1 deve essere uguale a C.

La prima regola garantisce che nessun metodo ordinario potrà essere più specifico di un costruttore dato; quindi, se invochiamo un costruttore con dei parametri dati, saremo sicuri che nessun metodo più specifico potrà "rubare" il ruolo del costruttore a runtime. Saremo cioè sicuri di invocare un costruttore.

La seconda regola, invece, ci assicura che ogni costruttore più specifico di un costruttore dato dovrà per forza di cose restituire un valore dello stesso tipo (e non di una sottoclasse) del valore del costruttore di partenza. Possiamo quindi determinare staticamente non solo il tipo più generico del costruttore, ma anche il tipo più specifico. Saremo quindi, a questo

punto, sicuri che ogni invocazione di un costruttore restituirà esattamente un oggetto del tipo desiderato.

Nella nostra implementazione sperimentale abbiamo incluso la verifica di entrambe queste regole; è quindi possibile verificarne, se lo si desidera, il corretto funzionamento. Preme comunque sottolineare che tali regole non rappresentano un vincolo per il programmatore, ma esistono solamente per rendere più rigorosa la struttura del linguaggio; durante un utilizzo normale, le regole entreranno in gioco, segnalando una incongruenza, solo nel caso di un utilizzo improprio o comunque incoerente dei metodi. È comunque buona norma tenere distinti, durante la programmazione, i metodi normali dai costruttori; questi ultimi rivestono infatti, come abbiamo visto, un ruolo particolare nel linguaggio, e vanno usati solamente dove ha senso pensare alla creazione ed inizializzazione di una nuova istanza.

3.14.2. Condivisione di istanze ed effetti collaterali

Da un punto di vista implementativo, tutti i linguaggi object oriented conservano le istanze, più o meno in forma di record, in memoria.

La gestione delle variabili, di conseguenza, può essere compiuta solo in due modi: o trattando l'intero record come variabile, oppure utilizzando solamente un puntatore alla struttura che è realmente in memoria.

La prima soluzione, più pulita da un punto di vista concettuale, è in realtà inattuabile da un punto di vista pratico, se non al prezzo di un overhead considerevole; ogni qual volta, infatti, si desiderasse passare la variabile come parametro, oppure assegnare una variabile ad un'altra, si dovrebbe procedere alla duplicazione dell'intero blocco, con una penalità inaccettabile.

Per questo motivo, la stragrande maggioranza dei linguaggi adotta la seconda soluzione: un semplice puntatore. La cosa comporta però uno spiacevole effetto collaterale, di cui è bene essere coscienti: ogni volta che una variabile viene assegnata ad un'altra, si avrà una duplicazione del solo puntatore, il che equivale a dire che il record che descrive le due variabili in memoria sarà il medesimo. Questo effetto di condivisione fra istanze non è particolarmente dannoso in sè, ma può portare, se non si presta attenzione, a dei side effect inattesi.

Si consideri, per esempio, il seguente programma scritto in Java:

```
class sideEffect
{
 long n;
 sideEffect(long v) { n=v;}
 static void dangerous(sideEffect a,sideEffect b)
 {
  System.out.println("b vale "+b.n);
  a.n+=20;
  if (b.n>10)
   System.out.println
      ("Ehi!  La variabile b è cambiata!  Ora è:  "+b.n);
 }
 public static void main(String av[])
 {
  sideEffect a;
  a=new sideEffect(5);
  dangerous(a,a);
 }
}
```

l'output è il seguente:

```
b vale 5
Ehi!  La variabile b è cambiata!  Ora è:  25
```

Come si può vedere, quando il parametro a viene incrementato, anche il parametro b, che ad esso è legato tramite questa forma di sharing, risulta incrementato; il risultato è che una variabile può, se non si presta attenzione, cambiare il proprio valore in modo inaspettato.

Purtroppo non c'è molto che possa essere fatto per ovviare a questo problema, a meno di non discostarsi in modo significativo dal paradigma imperativo, oppure di accettare un overhead pesantissimo; pertanto anche BOH si piega a questo compromesso, e si comporta nel medesimo modo.

D'altronde, anche l'elegantissimo Smalltalk ha esattamente lo stesso comportamento, come si può vedere dalla versione in questo linguaggio dell'esempio, disponibile in appendice.

Dunque, in BOH le istanze sono implementate tramite blocchi di memoria riferiti da puntatori, come avevamo già accennato nella sezione 3.4. Cosa accade, dunque, quando

si esegue una assegnazione? la semantica associata all'istruzione di assegnamento è la seguente:

```
a:=b;
```

significa: "La variabile a non riferisce più l'oggetto precedente; dopo l'assegnamento, fa riferimento allo stesso oggetto cui fa riferimento b."

Quindi, dopo l'assegnamento, a e b diventano due riferimenti allo stesso oggetto; le due variabili seguiranno la stessa sorte finchè una delle due non subirà un'ulteriore assegnamento.

È da notare che l'oggetto cui prima a faceva riferimento registrerà una diminuzione di uno del numero di riferimenti che ad esso vengono fatti. Nel momento in cui nessuna variabile (o campi di altri oggetti) farà più riferimento a tale oggetto, lo si potrà marcare come libero, ed il garbage collector si preoccuperà di deallocarlo automaticamente.

Definita che abbiamo la semantica dell'assegnazione, vorremmo naturalmente che rimanesse valida per tutti i tipi di assegnazione, indipendemente dagli oggetti coinvolti. Questo, però, comporta una possibile penalità in termini di efficienza per i tipi di base. Vediamo perchè.

Se anche i tipi base venissero realmente implementati come oggetti, e si avesse una sezione di codice come:

```
b:=5;   // b è un puntatore associato a "5".
a:=b;   // a e b puntano allo stesso oggetto
inc(b);
```

allora si dovrebbe riscontrare una variazione anche nella variabile a, in conformità a quanto dichiarato sulla condivisione di istanze fra puntatori. La necessità di mantenere, per coerenza, questo comportamento impedirebbe però di conservare gli oggetti appartenenti ai tipi base in registri macchina: se infatti mantenessimo il nostro intero "5" in un registro, anzichè tenervi un puntatore, un incremento coinvolgerebbe solo il registro in causa, e non le eventuali altre variabili che, stando alla semantica che abbiamo introdotto, dovrebbero essere legate allo stesso oggetto.

La soluzione che possiamo adottare è più semplice di quanto si potrebbe pensare: dato che, in BOH, abbiamo scelto che gli unici metodi autorizzati a modificare lo stato interno di un oggetto siano quelli definiti nella classe relativa, e visto che i tipi di base sono definiti nelle librerie di sistema, è sufficiente definire le funzioni che agiscono sui tipi di base in

modo che non alterino *mai* il contenuto interno dei parametri, ma restituiscano sempre un oggetto nuovo come valore di ritorno; per esempio, la funzione inc(), come definita nella libreria standard, non altera il contenuto "interno" del parametro passato, ma restituisce il valore incrementato come nuovo oggetto (fittizio, se l'implementazione usa i registri).

Pertanto, l'unico modo per modificare gli oggetti appartenenti ai tipi base da parte dei programmi utente, è utilizzare l'assegnazione oppure invocare funzioni di libreria; in entrambi i casi non viene persa la semantica che abbiamo definito.

Con questo semplice accorgimento viene quindi mantenuta l'illusione di lavorare sempre con oggetti veri anche nel caso dei tipi base, evitando discrepanze ed asimmetrie nel comportamento, pur consentendo all'implementazione di agire internamente in modo efficiente.

Non è quindi necessario, utilizzando questa strategia, sacrificare all'efficienza la presentazione al programmatore di un ambiente composto interamente da oggetti di comportamento uniforme; ne guadagna, ovviamente, l'usabilità e la semplicità del linguaggio.

Volendo poi affrontare in modo più deciso il problema dei side effects derivante dal passaggio di una medesima istanza al posto di più parametri, una possibile soluzione potrebbe essere quella di associare un lock ad ogni oggetto, elemento comunque utile nell'ipotesi di far evolvere il linguaggio verso il supporto del multithreading.[14]

In tale circostanza, infatti, un tentativo di modificare un oggetto dovrebbe implicare l'applicazione di un lock read/write (esclusivo); il tentativo di applicare un lock read o read/write ad un altro parametro entrerebbe in conflitto con il precedente qualora i due oggetti riferiti fossero in realtà il medesimo.

Naturalmente il meccanismo andrebbe sviluppato ed incorporato nel design del linguaggio in modo opportuno, ma si tratta comunque di un'idea di partenza per una possibile soluzione.

3.14.3. La garbage collection

Come già accennato in precedenza, uno dei meccanismi di supporto necessari a runtime per questo linguaggio è costituito dal garbage collector, che si incaricherà di deallocare automaticamente gli oggetti non più riferiti da alcuna variabile o altro oggetto. Le tecniche note in letteratura per effettuare la garbage collection sono numerosissime (Reference Counting, Mark-Sweep, Copying Collection, algoritmi incrementali etc.), e non se ne farà

[14] L'idea dei lock associati ad ogni oggetto è utilizzata anche in Java, con l'uso della keyword "synchronized". Si veda, ad es. [OW99].

qui una analisi dettagliata.[15] Basti dire che almeno una tecnica deve essere implementata per permettere il corretto funzionamento del linguaggio; non si fanno assunzioni riguardo al particolare meccanismo usato.

Per quanto riguarda il fatto che l'adozione di un garbage collector debba implicare automaticamente una forte inefficienza del linguaggio, ebbene molti ritengono che si tratti di un luogo comune da sfatare. I garbage collectors disponibili attualmente sono considerevolmente efficienti, ed i vantaggi che portano sul dover delegare l'utente ad una gestione manuale della memoria sono così numerosi (sicurezza di non avere puntatori pendenti, garanzia della deallocazione automatica dello spazio non più utilizzato etc), da rendere altamente consigliabile una gestione automatica della memoria, utilizzando appunto la garbage collection (Vedasi ad es. [Gar]). Anche il preconcetto secondo cui l'utilizzo di un garbage collector coinvolga obbligatoriamente fastidiose pause periodiche durante il funzionamento può essere facilmente smentito: sono infatti numerose le tecniche applicabili per realizzare garbage collector real time, pensati appunto per applicazioni in tempo reale e dove quindi i tempi di risposta devono essere certi e predeterminabili. Utilizzando un garbage collector di questo tipo, la gestione dello spazio di memoria viene effettuata in modo incrementale (seppure con alta efficienza), e l'attività risultante non è quindi rilevabile dall'utente finale (ulteriori riferimenti in [New]).

[15] Una bibliografia sconfinata, comprendente oltre 1550 riferimenti, si può trovare in [Jon]; per un panorama delle tecniche più note, si vedano le ottime "GC-FAQ", [Gar]. Per dettagli sulle tecniche di garbage collection utilizzate in Smalltalk/V si può fare riferimento a [Orn96].

4. Una Implementazione Sperimentale

Non abbiamo cercato di implementare tutte le caratteristiche che abbiamo descritto per il linguaggio, vista la complessità che riveste la realizzazione concreta di un compilatore completo; ci siamo quindi focalizzati sulla sperimentazione delle caratteristiche più importanti: l'utilizzo delle generic functions e dei multimetodi, l'applicazione delle regole per il controllo dell'ambiguità e della definizione dei costruttori, l'utilizzo di tipi statici e dinamici ed il relativo type checking, l'implementazione di un opportuno dispatcher in runtime adatto all'individuazione dinamica dei multimetodi idonei tramite la disambiguazione su più parametri simultaneamente. Abbiamo cercato inoltre, per rendere l'implementazione del linguaggio il più possibile fruibile, di implementare le peculiari caratteristiche sintattiche del linguaggio, fra cui gli operatori definiti dall'utente e l'utilizzo versatile del punto per identificare il primo operando di una funzione.

Abbiamo invece messo in secondo piano l'eredità multipla, in quanto le tecniche di trattamento, come abbiamo visto, non sono che una variazione di quelle usate per i multimetodi, mentre l'implementazione concreta è resa difficoltosa dalla complessità delle strutture dati che bisogna sintetizzare; abbiamo inoltre trascurato la suddivisione del sorgente in packages multipli, in quanto non essenziale ai fini della sperimentazione del linguaggio. Non è stato data infine grande rilevanza all'efficienza nel codice prodotto, preferendo invece puntare ad avere un prototipo funzionante in un tempo contenuto.

In questo capitolo, dunque, verrà descritto un compilatore sperimentale, riferito come "implementazione preliminare", di un sottoinsieme piuttosto ampio di BOH, realizzato al fine di sperimentare con mano le funzionalità ed il comportamento del linguaggio.

Allo stato attuale, tale compilatore presenta soprattutto limitazioni riguardo la suddivisione in package dei sorgenti, l'eredità multipla, la gestione della visibilità di classi e metodi (pubbliche/private), la mancanza di alcune strutture di controllo non essenziali, una implementazione parziale dei tipi quad e uquad, ed una gestione molto inefficiente della memoria.

Sono invece disponibili, e perfettamente funzionanti, gli operatori definiti dall'utente,

le espressioni ed i tipi primitivi, il type checking statico, le componenti, i vettori, i costrutti while {...}, if...elsif...else e for, i contesti annidati, le variabili locali, il dispatcher, un subset utilizzabile della libreria standard e praticamente tutto il resto.

A nostro giudizio, quindi, l'Implementazione Preliminare rappresenta in ogni caso un ottimo punto di partenza, perfettamente funzionante, per sperimentare concretamente il linguaggio e verificarne le caratteristiche.

È perfettamente verosimile pensare di utilizzare questa prima implementazione per realizzare un ulteriore compilatore scritto nativamente in linguaggio BOH.

4.1. Funzionamento generale

L'Implementazione Preliminare consiste in una serie di coppie di sorgenti lex/yacc che trasformano, in fasi successive, il codice sorgente iniziale in un sorgente in altro linguaggio (nel nostro caso Ansi C), pronto per la compilazione nativa.

La scelta di compilare in codice C, anzichè produrre direttamente codice eseguibile, è dettata principalmente dai seguenti motivi:

- rapidità di realizzazione. Il tempo richiesto per realizzare un compilatore di questo tipo è stato decisamente inferiore a quello richiesto per la realizzazione di un compilatore completo; non risultano necessarie infatti le parti di ottimizzazione e di produzione del codice finale, demandate al compilatore del linguaggio ospite.

- indipendenza dall'architettura. Producendo un codice C conforme alle specifiche ANSI si ottiene un codice compilabile ovunque esista un compilatore C adatto (ossia sulla stragrande maggioranza delle architetture disponibili oggigiorno).

È da notare che il codice sorgente risultante non utilizza affatto, se non per quanto riguarda inessenziali funzioni di supporto, le funzionalità object oriented ed altre caratteristiche tipiche del C++; il codice prodotto utilizza il C come una specie di macroassembler, utilizzandone le caratteristiche di alto livello veramente al minimo. Neppure le strutture di controllo di BOH (test, cicli etc.) vengono mappate sulle corrispondenti C; viene invece fatto ampio uso dell'istruzione "goto" (disponibile in C), come verrebbe fatto per la produzione di codice a tre indirizzi od altro equivalente linguaggio o rappresentazione intermedia.

Nulla proibisce di modificare quindi l'implementazione preliminare per produrre direttamente codice intermedio utilizzabile con i compilatori GNU,[1] utilizzando quindi nel modo migliore l'intero backend, ottimizzatore compreso, offerto da tale popolarissima famiglia di compilatori, oppure anche il bytecode della macchina virtuale Java (descritto in [Sun95]).

[1] Si faccia riferimento alla documentazione in linea del gcc, file gcc.texinfo, oppure /usr/info/gcc* (Linux Redhat 5.1), in cui il formato intermedio utilizzato (RTL) è illustrato accuratamente. A dire il vero, l'opzione consigliata dagli autori del gcc per l'interfacciamento con un nuovo front end è quella di utilizzare la rappresentazione interna ad albero, e passarla quindi alle fasi successive del gcc: "The proper way to interface GNU CC to a new language front end is with the "tree" data structure.", file /usr/info/gcc.info-15.gz.

4.2. Le fasi della traduzione

ll codice sorgente viene tradotto in fasi successive, qui di seguito descritte. Si noterà subito che molte fasi sono in realtà accorpabili in una fase sola, risparmiando tempo durante la scansione dell'input. Dal momento che però l'efficienza non era fra gli obiettivi principali dell'implementazione preliminare, si è preferito mantenere le vare fasi distinte, in modo da rendere più chiara la sequenza di trasformazioni.

4.2.1. Fase 0: preprocessing iniziale.

In questa fase il sorgente in ingresso viene preprocessato, in modo da trasformare tutte le parti del sorgente in caratteri minuscoli, eccettuate quelle comprese nelle stringhe di testo. Simultaneamente, tutti i simboli utilizzabili negli identificatori vengono tradotti in caratteri maiuscoli, secondo il seguente schema:

```
‘       ->      A
?       ->      F
_       ->      K
%       ->      P
~       ->      B
>       ->      G
-       ->      L
$       ->      Q
|       ->      C
<       ->      H
*       ->      M
#       ->      R
\       ->      D
+       ->      I
&       ->      N
@       ->      S
/       ->      E
=       ->      J
^       ->      O
!       ->      T
```

Dopo questa trasformazione, tutti i lexeme corrispondenti agli identificatori saranno costituiti, per le fasi successive, da soli caratteri maiuscoli e minuscoli. Questo ne semplifica ed omogeneizza il trattamento e la successiva trasformazione nei corrispondenti identificatori C che compariranno nel sorgente prodotto alla fine dell'elaborazione.

La fase 0 è implementata da lex0.l, contenuto nella sottodirectory Code0; il codice lex viene utilizzato in modo stand alone, senza un programma yacc abbinato.

4.2.2. Fase 1: ricerca degli operatori ed eliminazione dei commenti

Durante la fase 1 vengono ricercate le definizione degli operatori, e viene costruita internamente una tabella comprendente nomi, priorità ed associatività degli operatori trovati. A questa vengono aggiunte le definizioni contenute nel file "includes/std-ops", che contiene gli operatori definiti nel package system.

A questo riguardo va notato che l'implementazione preliminare permette di utilizzare un unico package alla volta: la clausola "uses" dell'intestazione viene in realtà ignorata, ed al package in fase del test vengono comunque messi a disposizione gli operatori, le classi ed i metodi che sarebbero normalmente definiti dal package "system".

Una volta completato l'elenco degli operatori, viene preparata una coppia di sorgenti per la fase successiva, partendo da una coppia di skeleton ("lex2.proto" e "yacc2.proto"), e costruendo dinamicamente i sorgenti "Code2/lex2.l" e "Code2/yacc2.y". L'idea è che l'utility yacc è comunque in grado di effettuare il parsing di un sorgente contenente operatori infissi con diversi livelli di priorità, data la grammatica ed alcune regole di disambiguazione in caso di conflitti.[2] Ebbene, una volta estratta la lista degli operatori, viene semplicemente costruita al volo una grammatica idonea e le regole relative, ottenendone così un programma yacc (ed il relativo lex) ad hoc per il programma dato.

Come funzione aggiuntiva, durante questa fase vengono eliminati i commenti di linea ed a blocco, controllando il livello di annidamento.

La fase 1 è realizzata da lex1.l e yacc1.y, contenuti nella sottodirectory Code1.

Il sorgente lex1.l non contiene particolarità di rilievo, ad eccezione forse della gestione della profondità di annidamento dei commenti tramite la variabile Nesting.

Nel codice di yacc1.y si può notare la grammatica piuttosto essenziale: durante il parsing, mentre gli operatori vengono analizzati, tutto il resto viene semplicemente ignorato e ricopiato in uscita inalterato. Saranno le fasi successive, in cui è descritta la grammatica completa, a rilevare eventuali errori sintattici. La funzione

```
int oprec_compare(oprec *first,oprec *second)
```

si occupa di confrontare due operatori, per verificare quale debba avere la precedenza sugli altri. Se due operatori hanno uguale livello di priorità, hanno la precedenza i binari

[2] Cfr. bison, documentazione in linea; in particolare il file /usr/info/bison.info-4.gz (distribuzione Linux RedHat 5.1).

sugli unari; fra due binari o due unari, hanno la precedenza quelli che associano a sinistra, poi quelli a destra ed infine i non associativi.

La routine dumpOpTab() è la parte centrale del programma: ormai riempito il vettore degli operatori opV, vengono costruiti i file lex2.l e yacc2.y. Per quanto riguarda lex2.l, vengono ricopiate inalterate le righe dallo skeleton lex2.proto fino a riconoscere il tag speciale "$$$<here>$$$". A questo punto, per ogni identificatore distinto utilizzato come operatore, viene aggiunta la riga:

```
<INITIAL>"id"    reflex(return _x_id;)
```

e quindi viene ricopiata la parte finale dello skeleton. La descrizione delle funzionalità del lex2.l così ottenuto verrà fatto entro breve, nella documentazione della prossima fase.

Analogamente viene costruito il file yacc2.y partendo da yacc2.proto; per ogni identificatore di operatore vengono aggiunte le righe:

```
%%token _x_id
%%type <str> _x_id
```

quindi vengono generate le regole grammaticali necessarie. La generazione è un po' complessa, ma l'idea di base è che vengono costruite semplicemente delle regole, per esempio per un operatore infisso binario del tipo:

```
expr:   expr _x_id expr
```

seguendo una descrizione intuitiva della grammatica. Naturalmente, se le definizioni di tutti gli operatori venissero semplicemente aggiunte una dietro l'altra, yacc non potrebbe sapere che a certi operatori va data precedenza su certi altri, e verrebbero generati una gran quantità di conflitti. Fortunatamente, yacc mette a disposizione uno strumento assai potente per risolvere situazioni di questo genere senza obbligare l'utilizzatore a riscrivere interamente la grammatica: la direttiva %prec, infatti, consente di specificare dei livelli di precedenza di alcune regole rispetto ad altre, consentendo quindi a yacc di cavarsi d'impaccio. Dato però che gli operatori possono essere utilizzati anche come identificatori ordinari, sono necessarie delle regole speciali per forzare la precedenza degli operatori in determinati casi. Esempio: se abbiamo gli operatori:

```
operator aaa x 100;
operator x bbb 150;
```

e poi nel codice si ha:

```
c:=aaa bbb;
```

si avrebbe un conflitto che yacc non è in grado di risolvere correttamente. Vengono quindi aggiunte ulteriori regole esplicite per risolvere anche questi casi. Per avere una idea dei file prodotti da lex1.l/yacc1.y, ossia un esempio di file lex2.l/yacc2.y, si vedano le stampe degli esempi in appendice.

Si noti la regola speciale inserita per il punto, che, agli effetti del parsing, ha l'aspetto di un operatore; difatti, se non venisse specificata una priorità anche per esso, si avrebbero conflitti nel parser. La priorità di default assegnata al punto, se utilizzato per invocare funzioni o per riferire componenti di una istanza, è 700.

4.2.3. Fase 2: risoluzione degli operatori.

Vengono qui utilizzati i sorgenti prodotti nella fase precedente, memorizzati nella sottodirectory Code2.

In questa fase vengono decodificate le espressioni contenenti operatori, e ne viene effettuata la trasformazione in espressioni munite di parentesi. Al termine di questa procedura, tutte le espressioni sono ricondotte alla forma di chiamate annidate di funzioni, in che ne rende più semplice il parsing e la successiva trasformazione in codice intermedio (in C, nel nostro caso). Allo stesso tempo, anche gli accessi a campi di istanze effettuati tramite il punto e le costanti floating point vengono trasformate nella medesima forma. Ad esempio, il codice:

```
a:=3+4*5-6/2-3;
b.c:=a.fun(x,y).test-3.25;
```

viene trasformato in:

```
a:=-(-(+(3,*(4,5)),/(6,2)),3);
c(b):=-(test(fun(a,x,y)),25(3));
```

Passando all'analisi dei sorgenti, nel file lex2.l (lex2.proto) due sono i punti da considerare con maggior attenzione: il trattamento delle costanti floating point e la macro "reflex". Le costanti in virgola mobile vengono accettate infatti in una varietà di formati diversi, ossia tutte quelle già citate nel capitolo 3.5, più tutte le variazioni ottenibili sostituendo il

punto decimale con una coppia di parentesi utilizzando le regole che abbiamo visto. Ebbene, la funzione normalizeNum() riduce il numero di formati possibili, eliminando il punto decimale; i formati che utilizzano le parentesi, anche se inconsueti, sono quelli in realtà utilizzati internamente in tutte le fasi successive.

Per l'elenco completo dei formati accettati, si faccia riferimento ai commenti inclusi nel sorgente.

Per quanto riguarda la macro "reflex", bisogna ricordare il funzionamento del lex, e precisamente il meccanismo di pattern matching: in caso di dubbio, lex cerca sempre di effettuare il matching con la stringa più lunga possibile. Questo, però, conduce inevitabilmente a considerevoli problemi in caso si incontrino delle espressioni contenenti operatori. Per esempio, se "and" è un operatore infisso binario, l'espressione "xandy" sarebbe riconosciuta da lex come un identificatore unico, e l'unico modo per forzare il riconoscimento sarebbe separare le varie parti usando spazi. Scopo della macro reflex è appunto quello di forzare lex a non riconoscere l'identificatore in una sola operazione, ma, ad ogni carattere, verificare se per caso si è giunti all'inizio di una sottostringa che identifica un operatore. In pratica, si procede carattere per carattere; ad ogni passo, se non si incontra un identificatore noto il carattere viene aggiunto ad una stringa usata come "accumulatore" dell'identificatore in fase di riconoscimento; se viceversa si incontra un operatore, i caratteri corrispondenti all'operatore vengono rimessi nel buffer di input (con "yyless(0)") e viene restituita la stringa letta fino a quel punto, in qualità di identificatore. Alla successiva invocazione dell'analizzatore lessicale, il buffer dell'identificatore sarà vuoto (test "!strcmp(lastID,"")"), e verrà quindi riconosciuto finalmente l'operatore incontrato.

Questo modo di procedere, pur se perfettamente funzionale, e più che idoneo ad una implementazione di test, presenta purtroppo alcuni problemi: una volta definito un operatore, non sarà più possibile utilizzare nessun identificatore che contenga come sottostringa l'operatore definito. Per esempio, avendo definito "and", non sarà possibile utilizzare come nome di variabile "sand", perchè lex cercherà sempre di scomporlo in un identificatore più un operatore. Per aggirare il problema, la soluzione migliore sarebbe forse quella di abbandonare interamente lex, e di costruire un analizzatore lessicale ad hoc, di comportamento simile a quello costruito con la macro "reflex" citata, ma in grado di dare priorità agli identificatori noti rispetto agli operatori. Ossia, sarebbe necessario un analizzatore lessicale "intelligente", in grado di variare l'automa a stati finiti che guida il riconoscimento dei token in modo incrementale, a mano a mano che si entra e si esce dai contesti. Per esempio:

```
c:=panda;       // panda non è un identificatore noto
                // -> scomposto in p and a
{
 panda:=41;     // non possono esserci operatori sulla LHS
                // -> panda è un nuovo id
                // l'automa a stati finiti, e di conseguenza
                // l'analizzatore lessicale, viene modificato
                // in modo da riconoscere "panda" come token.
 d:=panda;      // panda viene riconosciuto come identificatore
}               // fine del contesto -> utilizzando uno "stack
                // di automi", viene recuperato l'automa
                // precedente
f:=panda;       // riconosciuto nuovamente come p and a
```

Sarebbe un interessantissimo oggetto di studio il verificare come un automa a stati finiti si modifichi incrementalmente aggiungendo un nuovo pattern da riconoscere; l'implementazione di un algoritmo efficiente per realizzare la trasformazione costituirebbe certamente un punto di partenza per la realizzazione di un analizzatore lessicale ottimale per il linguaggio BOH. A questo riguardo, dei riferimenti utili possono essere [HKR90] e [HKR92].

4.2.4. Fase 3: analisi dei metodi e delle classi

La fase 3 è essenzialmente una fase di controllo semantico; risolto il problema degli operatori, infatti, si può passare all'analisi dei messaggi e dei metodi, si può verificare la gerarchia delle classi e ricavare la struttura interna delle istanze, compilando l'elenco delle componenti con i relativi tipi. Come funzione collaterale, vengono eliminati durante la scansione del codice spazi, tabulazioni e ritorni di carrello ridondanti.

Durante il parsing, dunque, viene compilato l'elenco delle classi e dei metodi e componenti definite al loro interno, nonchè l'elenco dei metodi globali. Nel file lex3.l non sono presenti elementi particolari, mentre nel file yacc3.y, possiamo notare la funzione addMethHeader(), che aggiunge un nuovo metodo e definisce, se necessario, il messaggio relativo; oltre a questo, viene anche verificato che tutti i metodi con stesso nome ed arità abbiano la stessa visibilità (o tutti pubblici o tutti privati), che non si stia cercando di definire costruttori al di fuori di una classe e che non ci siano duplicati. La funzione loadLibraryTypes() si preoccupa di precaricare, prima dell'inizio del parsing vero e proprio, gli elenchi di messaggi, metodi e classi, definiti nella libreria standard. La funzione dumpMethods() stampa l'elenco delle intestazioni dei metodi, e mentre fa questo verifica che tutti i tipi dei parametri, ed il valore di ritorno, appartengano a classi esistenti ed accessibili; analogamente

dumpClasses() stampa l'elenco delle classi (con le relative componenti), e verifica che tutti i tipi che compaiono (tipi delle componenti e la sopraclasse) esistano e siano disponibili.

Il risultato principale di questa fase, al di là delle minime trasformazioni applicate all'input, è una apposita tabella descrittiva di metodi e classi, che utilizza un linguaggio libero dal contesto creato appositamente; la descrizione del formato si può trovare fra i commenti di yacc3.y. Il nome del file contenente la tabella è, di default, "test.out3.def".

4.2.5. Fase 4: produzione del codice intermedio

Il file contenente la tabella prodotta nell'ultima fase viene aggiunto all'inizio del resto del sorgente, insieme al file "library.def" che contiene le definizioni standard di libreria (metodi e classi), ed il risultato (file "test.outx") viene dato in pasto alla coppia di programmi derivanti da lex4.l e yacc4.y, che costituiscono la parte vera e propria di traduzione del sorgente iniziale in un codice intermedio; nel nostro caso, viene prodotto del codice C ANSI, praticamente pronto per la compilazione.

Nel file lex4.l si possono notare due elementi di spicco: la riconversione dei numeri floating point nella consueta notazione con il punto decimale (al fine di utilizzarne la stringa risultante nel codice C in corso di produzione), ed il trattamento delle costanti intere, quad compresi, utilizzando solo long, che si suppone siano, come accade frequentemente nelle architetture disponibili oggigiorno, in grado di conservare almeno un valore a 32 bit. Questa scelta, malgrado nel C ANSI sia previsto un tipo "long long", è stata fatta per semplificare la generazione di codice a 32 bit, nel caso l'implementazione preliminare dovesse essere modificata per produrre un codice intermedio diverso, dotato di tipi primitivi lunghi al massimo 32 bit.

Il file yacc4.y, cuore di tutto il processo di compilazione, effettua concretamente l'operazione di traduzione basandosi sulla grammatica che descrive la sintassi del linguaggio. Procediamo ora ad una analisi delle sue funzionalità.

In una prima fase vengono ricostruite in memoria le tabelle di messaggi, metodi e classi (con le componenti relative) generate durante la fase precedente. Per fare questo viene estesa la grammatica del linguaggio in modo da includere una piccola grammatica aggiuntiva, incaricata di effettuare il parsing delle tabelle. Tale grammatica compare verso l'inizio del file yacc4.y, e si può individuare faclmente dal momento che tutti i nonterminali sono del tipo spXxxx. Completato il caricamento delle tabelle, il programma ha tutte le informazioni necessarie per iniziare dunque, finalmente, la produzione del codice.

La traduzione del sorgente viene effettuata nel seguente modo: per ogni metodo definito

viene composta una funzione C, e per ogni classe viene assemblata una definizione di struttura. Le funzioni C costruite hanno la forma:

```
void *_m_nomeMessaggio_tipoParam1_tipoParam2...tipoParamN (
        _x_tipoParam1 nomeParam1_n,...,
        _x_tipoParamN nomeParamN_n)
```

Dal momento che gli identificatori BOH possono ora contenere solo caratteri in [0-9a-zA-T], l'underscore viene utilizzato come carattere speciale per evitare clash fra traduzioni di identificatori diversi. I nomi dei parametri, rispetto agli originali identificatori BOH, hanno come suffisso un numero unico, che viene utilizzato nella gestione dei contesti per eliminare conflitti in caso di ridefinizioni dello stesso identificatore in contesti diversi. Per esempio:

```
a:=5;        // trasformato in a_1
{
 b:="ciao";  // questo è b_2
}{
 b:=7.25;    // nuovo contesto, diventa b_3
}
b:=6;        // questo è b_1
```

Il nome della funzione viene calcolato sulla base del nome del messaggio e dei tipi dei parametri (e sul loro numero), in modo da ottenerne una signature univoca.

Se il metodo BOH corrispondente non ritorna alcun valore, la funzione viene ugualmente dichiarata come void *, e verrà restituito un puntatore NULL fittizio al termine della funzione; il motivo è da ricercarsi nell'implementazione del dispatcher (si veda a questo riguardo la sezione "Il supporto runtime", più avanti), che tratterà sempre funzioni che restituiscono un puntatore a void.

Proprio a questo riguardo, si noti che il valore di ritorno è indicato come puntatore generico; la gestione dei tipi BOH viene effettuata seguendo la logica interna del programma BOH corrispondente, e viene interamente ignorato il type checking nativo del C. Sono frequenti infatti le conversioni da un puntatore ad un'altro di tipo diverso tramite dei cast; al fine del codice prodotto, quello che è rilevante è solo il fatto di avere un puntatore generico (come d'altronde avverrebbe nel caso della compilazione in codice macchina, o di un codice intermedio generico).

La generazione delle intestazioni dei metodi viene effettuata dalle funzioni setupSignature(), beginMethodBody() e beginSuperMethodBody(). Quest'ultima, in particolare,

genera le intestazioni per i costruttori secondo una forma leggermente diversa: anzichè restituire il valore del puntatore alla nuova istanza creata, viene accettato un parametro extra iniziale che riceverà il puntatore all'istanza in fase di inizializzazione, che deve essere già allocata.

Il motivo di questa differenza risiede nel fatto che i costruttori vengono utilizzati sì per la creazione di nuove istanze, ma anche per l'inizializzazione delle sottoistanze corrispondenti alle sopraclassi (cfr. clausola "super" nelle dichiarazioni di classi). Se un metodo costruttore restituisse un puntatore ad una nuova istanza creata, sarebbe inutilizzabile per questa seconda funzione.

Facciamo un esempio:

```
!padre:  super object
{
 a:long;
 !padre() :  super object() {a:=4;}
}

!figlia:  super padre
{
 b:long;
 !figlia() :  super padre() {b:=6;}
}
```

Ogni istanza della classe figlia conterrà la componente b, più una sottoistanza corrispondente alla classe padre, la quale, a sua volta, conterrà la componente a più la sottoistanza corrispondente ad object e così via. Durante l'inizializzazione della istanza della classe figlia, tramite il metodo figlia(), dovrà essere invocato il metodo padre(), come specificato dalla clausola super; il record corrispondente alla istanza di figlia sarà allocato però in una volta sola, con la dimensione comprendente tutta la nidificazione di istanze e sottoistanze necessarie; il costruttore padre() (e quelli da lui invocati) non dovranno dunque allocare ulteriore memoria, e men che meno restituire un puntatore.

Allocando quindi il blocco di memoria subito prima di invocare figlia(), e passando ai costruttori il puntatore al nuovo blocco di memoria ottenuto, il problema è banalmente risolto: i costruttori non effettueranno mai internamente l'allocazione dell'istanza di cui si chiede la creazione/inizializzazione, ma utilizzeranno il parametro aggiuntivo come puntatore ad una istanza già allocata.

Per quanto riguarda le classi, queste vengono trasformate in strutture, dividendo la parte

che contiene sottoistanze e dati da quella che contiene dati per la gestione delle istanze stesse, secondo il seguente schema:

```
typedef struct _data_classname {
  struct _data_superclass super;
  void *_x_field1;
  void *_x_field2;
  void *_x_field3;
  ...
};
typedef struct _x_classname {
  short refNum;
  MTCclass *mytype;
  struct _data_classname me;
};
```

Le istanze vengono poi allocate come variabili C di tipo _x_classname, che quindi conterrano un refNum (attualmente inutilizzato), il puntatore al record che descrive la classe, ed identifica quindi il tipo della istanza, ed il record "me" di tutte le componenti, contenute in strutture _data_classname annidate, seguendo la gerarchia delle classi. Si noti come le componenti siano implementate tramite puntatori (generici) ad ulteriori istanze.

La parte di compilazione vera e propria viene effettuata tramite una traduzione guidata dalla sintassi; la trasformazione è quindi distribuita in tutte le produzioni della grammatica. Si può facilmente desumere la modalità di traduzione di determinate parti del codice consultando nel sorgente yacc4.y la produzione relativa.

Si può osservare come la traduzione delle strutture di controllo venga fatta non già utilizzando le corrispondenti strutture C, ma usando solo ed esclusivamente istruzioni "goto", allo stesso modo di quanto avverrebbe nella produzione di un codice intermedio generico, dove siano presenti, come frequente, solamente istruzioni di salto.

Per quanto riguarda le funzioni di supporto, una particolare menzione va fatta per la routine messageOK(), che effettua un controllo accurato sulla correttezza della definizione dei vari metodi corrispondenti ad un messaggio. Si ricordi che un messaggio, con i relativi metodi, è identificato dalla combinazione di nome ed arità.

I controlli che vengono dunque effettuati sono, nell'ordine:

1. Un messaggio deve avere almeno un metodo corrispondente. Se la cosa non si verifica, probabilmente si è verificato un errore interno del compilatore (per esempio, potrebbe essere inconsistente il file "library.def").

2. Un messaggio di arità zero deve avere uno ed un solo metodo corrispondente. Se ce n'è più di uno, siamo di fronte ad una definizione duplicata.

3. Viene poi effettuato il controllo della coerenza delle definizioni dei metodi, secondo i criteri descritti nei capitoli iniziali di questo lavoro e le ulteriori regole speciali da applicarsi nel caso dei costruttori descritte nel capitolo 3.14.1. Tali regole sono comunque riportate come commento nel sorgente.

4. Infine vengono cercate le eventuali coppie di metodi che darebbero luogo ad ambiguità nella risoluzione a run time, e viene stampato l'elenco dei metodi mancanti. Viene inoltre controllata l'eventuale presenza di definizioni duplicate.

Un'altra routine rilevante è la findStaticCall(), che si incarica di trovare, accertato che, grazie alle regole citate, è sempre possibile farlo, il tipo più generico restituito da una determinata invocazione. In pratica, viene cercato il "tipo statico" di una espressione, il tipo più generale che un messaggio potrà restituire data una certa combinazione di parametri.

È da notare anche il gruppo di chiamate initSymbolTable(), findSTlastContextEntry(), findSTentry(), addSTentry(), openSTcontext() e closeSTcontext(), che implementano la symbol table, con il consueto meccanismo a stack utilizzato abitualmente per la compilazione dei linguaggi struttrati.

L'implementazione della symbol table non è certamente fra le più efficienti, essendo basata su liste e su elementi allocati dinamicamente, ma è facile, verificata la funzionalità del compilatore, sostituirne l'implementazione con una migliore conservando la medesima interfaccia.

Anche altrove, a dire il vero, l'implementazione preliminare abusa di liste semplici, notoriamente poco efficienti; la scelta è stata fatta al fine di contenere i tempi di sviluppo del prototipo, pur ottenendo una implementazione il più possibile completa.

Una eventuale versione riveduta dell'implementazione dovrà certamente tenere in debito conto anche i problemi di efficienza.

Come osservazione finale, si può notare come in alcune regole si faccia riferimento ad un "optionalAmpersand"; si tratta in effetti di una feature attualmente non implementata.

4.2.6. Fase 5: compilazione finale

Il risultato della fase 4 è un sorgente C pronto per la compilazione, contenente una traduzione del programma sorgente originale.

Prima di poter procedere alla compilazione, è necessario, nel caso si utilizzi il compilatore gcc, procedere ad una passata intermedia, al fine di ovviare ad una limitazione di quel

particolare compilatore. Difatti, il gcc si rifiuta di compilare codice che contenga goto che scavalchino dichiarazioni di variabili, come accade nel caso del codice prodotto dalla fase 4.

Al fine di ovviare a questo inconveniente, viene effettuato un postprocessing del sorgente, al fine di raggruppare tutte le dichiarazioni di variabili all'inizio di ogni funzione. Questa funzione viene svolta dal programma di supporto "mix", contenuto nella subdirectory "helper". Si noti che, utilizzando altri compilatori (per es. il CodeWarrior della Metrowerks), questa modifica del sorgente non è necessaria e può essere evitata.

Ottenuto finalmente il sorgente modificato, viene invocato il compilatore C, con il quale vengono assemblati il codice tradotto (test.out4), l'implementazione della libreria standard (library.c) ed il supporto runtime (runtime.cpp). Ciò che ne si ottiene è un singolo eseguibile stand alone, di nome "result", il quale costituisce la traduzione finale in codice macchina del sorgente BOH di partenza, ossia l'ambito risultato di tutto il processo di compilazione.

4.3. Il supporto runtime

Le funzioni di supporto necessarie durante la fase di runtime sono incluse staticamente durante la compilazione, e sono contenute, come accennato, nel file "runtime.cpp", presente nella directory "Code5".

Scorrendo tale file, si può vedere l'implementazione di un albero generico (class tree), da cui vengono ricavate una sottoclasse adatta per la memorizzazione delle classi (class MTCclass). Analogamente, viene definita una lista generica (class list) e due sue sottoclassi per registrare i messaggi (class MTCmessage) ed i metodi (class MTCmethod).

Tali strutture dati verranno poi riempite dalle funzioni setupMTCclasses() e setupMTC-messages(), costruite a partire dal sorgente BOH iniziale e che compaiono net file test.out4; fatto ciò, come si può vedere nel corpo di main(), in fondo al sorgente, viene invocato il dispatcher, lanciando il messaggio "main" di arità zero, che costituisce l'entry point del programma BOH.

Per quanto riguarda il dispatcher, questo viene implementato con la routine dispatch-GeneralMTCmessage(), che, basandosi sui tipi effettivi posseduti dai parametri a runtime, cerca il metodo migliore da invocare per quella specifica invocazione. Una volta trovato il metodo del caso, viene effettuata la chiamata utilizzandone il puntatore al codice; si noti come venga aggirato interamente il controllo dei tipi dei parametri, utilizzando ad hoc il meccanismo di cast del C. Per esempio, nel caso di un metodo di arità tre, l'invocazione si presenta come:

```
((void*(*)(void*,void*,void*))best->code())(p0,p1,p2);
```

Il che suona più o meno come: "ricava il puntatore alla funzione associata al metodo best, convertilo in un puntatore a funzione che restituisce un void* dati tre argomenti void*, e quindi invocalo con i tre parametri p0,p1 e p2".

Ricordiamo che, in C, il tipo "void*" si comporta come un puntatore generico, e che una variabile od un parametro di tipo void* accetta un qualunque puntatore.

4.4. La libreria standard (package “system”)

Come abbiamo già visto, nell’implementazione preliminare la clausola "uses" del package viene ignorata, e viene messa sempre a disposizione la libreria standard, come se fosse scritto sempre "uses system".

L’insieme di chiamate implementate nella libreria standard finora è solo un subset di quelle suggerite precedentemente; verranno ora illustrate quelle disponibili, e verrà dato uno sguardo alla loro realizzazione interna, contenuta nei file ’library.c" e "library.def" (oltre che "std-ops").

Sono attualmente implementati i seguenti metodi:

```
nl().
object()                // costruttore
=(object,object):bool   // confronto fra puntatori
print(object)
println(object)
not(bool):bool
and(bool,bool):bool
or(bool,bool):bool
=(bool,bool):bool
<>(bool,bool):bool
print(bool)
println(bool)
num()                   // costruttore
+(text,text):text
readText():text
print e println su text
```

Inoltre

```
+, -, *, /, <, >, <=, >=, <>, =, print e println
```

definiti su

```
byte, ubyte, word, uword, long, ulong, float e double,
```

nonchè su quad ed uquad (non completamente implementati).

Non sono implementate le classi int, real, byteInt, wordInt, longInt e quadInt, ma tutti i tipi numerici vengono fatti discendere direttamente da num; la classe class non è inoltre disponibile in quanto tale.

Completare la libreria standard non è operazione difficile; il subset finora implementato, comunque, è sufficiente per una prima sperimentazione delle caratteristiche del linguaggio.

È da notare che ogni modifica del file library.c deve essere rispecchiata nei due file library.def e std-ops; la sintassi seguita dal file library.def è descritta nei commenti del file yacc3.y, mentre il file std-ops consiste in un insieme di righe del tipo:

```
priority arity type name
```

dove priority è il valore numerico della priorità, compreso fra 1 e 999, arity vale 1 per gli operatori unari e 2 per quelli binari, type vale 'n' per gli operatori non associativi, 'r' per quelli che associano a destra (o per gli operatori unari prefissi) ed 'l' per quelli che associano a sinistra (o per gli operatori unari postfissi). Il nome deve essere indicato come risulta dalla rimappatura dei caratteri indicati nel capitolo che descrive il primo preprocessing, in questo stesso manuale ("Fase 0: preprocessing iniziale").

4.4.1. Limitazioni dell'Implementazione Preliminare

Come già accennato, l'implementazione preliminare costituisce uno strumento introduttivo alla programmazione in BOH, ed implementa solo un subset, peraltro piuttosto esteso, delle funzionalità del linguaggio completo.

In particolare sono presenti le seguenti limitazioni:

- non è supportata la compilazione modulare: è possibile compilare un unico package che importa le definizioni del package di sistema. Non è possibile chiamare tutte le funzioni definite nel package tramite una shell interattiva, ma l'unico entry point del codice è costituito dal metodo main(), di arità zero.

- L'eredità multipla non è implementata.

- gli oggetti di tipo text non possono essere di lunghezza arbitraria, ma possono solo contenere al massimo 512 caratteri; l'unico set di caratteri effettivamente collaudato è quello ISO-8859-1 (ASCII esteso con caratteri dell'Europa Occidentale); non dovrebbero comunque sussistere problemi nell'utilizzare altre variazioni ISO-8859.

- sono al massimo definibili 400 operatori; i metodi non possono avere più di 10 parametri.

- la gestione della visibilità di classi e metodi (pubblici/privati) non è stata interamente verificata; dal momento che non è stata realizzata la compilazione modulare, questa feature è implementata solo parzialmente.
- sono implementate le strutture if...elsif...else, la while {...} ed il ciclo for; non sono invece ancora realizzati il do {...} while ed il selettore case.
- non sono al momento supportate le variabili globali.
- il limite maggiore è probabilmente l'assenza di un garbage collector; nel codice eseguibile finale gli oggetti, anche quelli temporanei, vengono allocati ma mai deallocati. Questo implica che durante l'esecuzione il codice continua a consumare memoria, e non è quindi possibile realizzare codice che prosegua l'esecuzione a tempo indefinito. È comunque possibile adattare facilmente l'implementazione in modo da utilizzare una delle librerie C di gestione automatica della memoria disponibili liberamente.

4.5. Miglioramenti da apportare

Non è difficile modificare questa implementazione in modo da renderla adatta anche come tool di sviluppo reale. Ecco alcune delle caratteristiche che possono essere completate o migliorate senza grande sforzo:

- Una modifica consigliabile è certamente modificare ed integrare la libreria standard in modo da ampliare la gamma di funzioni disponibili. È già stata descritta la sintassi dei file relativi (library.c, library.def, std-ops). Tali modifiche non comportano particolari difficoltà: si tratta solo di scrivere, seguendo il modello delle funzioni già presenti, le definizioni mancanti.

- La modifica più importante da applicare sarebbe certamente l'implementazione di un garbage collector. Come detto, si può utilizzare una delle libreria preconfezionate reperibili in rete, oppure realizzare un garbage collector dinamico scritto su misura.. Per una descrizione delle tecniche generali più utilizzate, si rimanda a [Gar].

Con queste semplici migliorie, l'Implementazione Preliminare diventerebbe a tutti gli effetti un compilatore utilizzabile nel mondo reale, anche per scrivere applicazioni complesse. Possono però essere implementate ulteriori modifiche:

- Nel quadro del completamento delle funzioni di libreria, andrebbe implementata anche, oltre alle altre classi mancanti, la classe "class". In particolare, sarebbe opportuno unificare il record descrittivo della classe class ed il record MTCclass, in modo che il puntatore al tipo, all'interno dell'istanza, punti effettivamente ad un oggetto di tipo class.

- Sarebbe auspicabile un miglioramento della diagnostica d'errore, e, ove possibile, l'applicazione di meccanismi di error recovery (allo stato attuale, per scoprire il punto in cui, nel sorgente, si è verificato l'errore, è necessario consultare il file di lavoro in uscita dalla fase che ha segnalato il problema, e vedere dove si è interrotto il processo di compilazione).

- L'implementazione delle variabili globali può comportare qualche problema, dal momento che la loro definizione può apparire in un punto qualsiasi del programma BOH, ma nel codice C è necessario che le variabili siano note prima del loro utlizzo. Questo rende difficoltosa la generazione del codice in modo guidato dalla sintassi, in quanto può accadere che le informazioni necessarie per la traduzione siano disponibili dopo che la variabile è già stata utilizzata, e ne è quindi già stato necessario

conoscere il tipo. Una possibile soluzione consiste nell'effettuare un postprocessing del sorgente dopo la fase tre in modo da raggruppare tutte le definizioni di variabili globali all'inizio del sorgente; in tale modo si potrà essere sicuri, durante la traduzione, che la definizione preceda l'utilizzo. Si noti che, nel file yacc4.y, la traduzione delle variabili globali non è implementata. Dal momento che le variabili globali vanno inizializzate prima della prima chiamata al codice (prima di main(), al momento del caricamento del package), la routine C principale contenuta nel file runtime.cpp dovrà, prima di effettuare il dispatching di main(), invocare una routine tipo initGlobalVar(), che si occuperà di chiamare, a turno, tutte le inizializzazioni delle diverse variabili globali.

- Volendo migliorare infine l'efficienza dell'implementazione, si può pensare di sostituire il dispatcher attuale con uno basato su tabelle, come descritto nella sezione "Considerazioni sull'efficienza", più avanti in questo stesso manuale. Per aumentare infine l'efficienza in modo drastico, si possono seguire le indicazioni relative al trattamento dei tipi fornite nella sezione "Considerazioni sull'efficienza", non chiamando quindi le funzioni di libreria per le operazioni sui tipi primitivi, ma includendo direttamente il codice relativo all'interno del sorgente.

4.6. Aspetti pratici

4.6.1. Distribuzione ed utilizzo

L'implementazione preliminare viene distribuita come un unico file tar, compresso con gzip. Dalla decompressione risulta una directory principale, di nome "BOH", che contiene un certo numero di sottodirectory, che abbiamo già incontrato durante la descrizione dell'implementazione. In particolare, le sottodirectory Code0..Code4 contengono i sorgenti di lex e yacc per le varie fasi della compilazione, in Code5 si trova il file "runtime.cpp", che contiene il supporto runtime per il programma BOH, in includes si trovano vari file di supporto e la libreria standard, in helper si trova il sorgente del programma che effettua il postprocessing fra la fase 4 e la fase 5.

Oltre a tutte queste sottodirectory, sono presenti nella directory BOH alcuni sorgenti di esempio ed alcuni script di shell Unix che vengono usati per controllare il processo di compilazione, oltre ad ulteriori funzioni di utilità.

Vediamo un caso concreto: avendo un file contenente il sorgente BOH che si desiderano compilare, è sufficiente dare il comando "boh nomefile", eventualmente preceduto dal path completo dove si trova il comando se la directory che lo contiene non è già stata aggiunta al path di sistema. La prima volta che il comando viene eseguito verrà eseguita la compilazione degli eseguibili che effettuano le varie fasi della compilazione. Verrà stampanta una diagnostica dell'esecuzione delle varie fasi, ed al termine verrà prodotto il file "result", contenente il risultato finale della compilazione, ossia l'eseguibile standalone. A fini di debugging, verranno lasciati, nella directory contenente il compilatore, i file "result.symbolic" e "result.source" che contengono rispettivamente una rappresentazione della scomposizione delle espressioni in chiamate elementari, ed il sorgente C corrispondente.

Durante la compilazione verranno anche prodotti diversi file temporanei, che verranno cancellati a mano mano che la procedura continua. Se qualcosa dovesse andare storto, si potrà trovare un file test.outn, contenente l'output della compilazione fino al punto dell'errore, il che può facilitare la diagnostica. Se la normalizzazione dei caratteri speciali dovesse dare problemi di interpretazione del file, lo script "dog" può essere può essere di aiuto nell'effettuare la mappatura opposta, trasformando cioè i caratteri maiuscoli nei simboli originari.

Lo script "boh_debug" è del tutto equivalente a "boh", ma non cancella i file temporanei prodotti durante la compilazione; questi ultimi possono essere di valido aiuto durante il debugging del compilatore.

Lo script "clean" provvede, qualora fosse necessario, ad una pulizia dei file esegui-

bili e dei file temporanei eventualmente prodotti. Lo script "autopack" ricostruisce, nella directory dove compare anche la directory BOH, un file tar gzipped contenente il kit di distribuzione, comprendente le modifiche ai sorgenti che fossero state apportate nel frattempo.

Tutti i file "test.*" sono programmi di esempio, scritti in BOH, che possono essere usati per effettuare alcune prove sul linguaggio.

Lo script "de_iso_8859_1", infine, può essere utilizzato qualora si dovesse fare il porting su un ambiente in cui i caratteri accentati non siano supportati: ricopia l'intera gerarchia di directory e di file in una directory di nome "converted", allo stesso livello della directory BOH, in cui da ogni file sono stati eliminati i caratteri accentati, e sostituiti con coppie di caratteri ASCII.

Ulteriori file contenenti programmi di test scritti in vari linguaggi di programmazione, e riportati in appendice, sono contenuti in un secondo file tar, di nome langxxxxxxxx.tgz, che, decompresso, produce una directory di nome "lang".

Questa documentazione è allineata con la versione "06110220" dei due file.

4.6.2. Informazioni sulla portabilità

L'Implementazione Preliminare è stata scritta utilizzando un computer portatile PowerBook 280c Apple™, con il compilatore C/C++ CodeWarrior Pro della MetroWerks™, ed è stata ultimata utilizzando Linux RedHat 5.1. Gli archivi finali, disponibili su richiesta, contengono il sorgente e gli esempi di programmazione e sono utilizzabili con Linux su qualunque architettura; dovrebbero comunque essere portabili su altri sistemi operativi senza particolari difficoltà, a patto di disporre delle utilities flex e bison (implementazioni GNU di lex e yacc).

Per la compilazione delle parti in C, è stato utilizzato il compilatore gcc, disponibile su una gran varietà di piattaforme; neppure su questo fronte, quindi, dovrebbero esserci problemi particolari di portabilità.

Ulteriori richieste di informazioni e commenti concernenti l'implementazione od il linguaggio possono essere inviati all'indirizzo e-mail: "toni@beer.com".

5. Conclusioni

In questo lavoro abbiamo mostrato come le generic functions ed i multimetodi possano essere impiegati per risolvere in modo concreto alcuni problemi che emergono durante la progettazione dei linguaggi object oriented; come abbiamo infatti mostrato, tramite tali strumenti ed un opportuno studio formale possiamo:

- evitare i problemi collegati all'utilizzo simultaneo di risoluzione statica tramite overloading e dispatching dei messaggi a runtime (cap. 2.1 e 2.2)
- avere maggiore flessibilità nell'utilizzo dei tipi di ritorno dei metodi (cap. 2.3.4)
- effettuare un type checking statico, anche con i multimetodi, eliminando la possibilità che in esecuzione non esista un metodo adatto in corrispondenza di un messaggio (cap. 2.3, ed in particolare cap. 2.3.1 e 2.3.2).
- eliminare le ambiguità nella ricerca dei metodi durante il dispatching di messaggi a runtime tramite una analisi statica (cap. 2.3.3).
- introdurre l'eredità multipla in modo rigoroso, definendo tecniche per riconoscere e segnalare staticamente le condizioni di ambiguità invece di utilizzare regole implicite, difficili da ricordare, per effettuare la disambiguazione (cap. 2.4.2.3)

Abbiamo inoltre introdotto la definizione di un linguaggio sperimentale teso a verificare l'applicabilità concreta di tali meccanismi (cap. 3), concepito però in modo tale da ammettere una implementazione efficiente (cap. 2.5) pur conservando una complessiva semplicità di utilizzo, e tale da presentare all'utilizzatore un ambiente composto esclusivamente da oggetti di comportamento omogeneo (cap. 3.14.2 e 2.5.2).

Di questo abbiamo quindi realizzato una implementazione concreta, realmente funzionante, in grado di compilare un subset significativo del linguaggio definito (cap. 4, i sorgenti completi in Appendice), mettendo alla prove le tecniche più rilevanti fra quelle descritte.

Grazie a tale implementazione, è stato infine possibile mostrare, tramite esempi reali (elencati in Appendice), la praticabilità delle tecniche illustrate, e come quindi i multimetodi possano essere fattivamente impiegati quale valida alternativa alle soluzioni classiche nell'ambito dei linguaggi object oriented tipizzati.

Ulteriori sviluppi

La metodologia illustrata nel corso del presente lavoro, ed il linguaggio che ne è scaturito, possono essere una base interessante per ulteriori ricerche.

Ad esempio, si può pensare di estendere il meccanismo di costruzione dei grafi in modo da contemplare anche i parametri funzione, pur mantenendo i vantaggi derivanti dal type checking statico. Per quanto riguarda il linguaggio, sarebbe interessante verificare come si comporti nei confronti del problema della "fragile base class" ([Pes95b], [MS97], [IBM]), che assume nuove dimensioni nel caso dei multimetodi, stabilendo se si rendano necessarie modifiche significative della struttura del linguaggio per evitare tale problema. Altri argomenti di interesse possono risiedere nelle estensioni del sistema idonee a supportare il multithreading usando, ad esempio, i lock alla maniera di Java ([OW99]), oppure nell'esplorarne l'utilizzo nell'ambito dei sistemi persistenti e/o distribuiti ([ABC$^+$83], [KM97], [KM], [KCC$^+$97]), verificando quali nuove situazioni comporti l'utilizzo dei multimetodi.

A. Appendice: sorgenti ed esempi.

A.1. Confronto fra linguaggi

A.1.1. SideEffect in Smalltalk

```
"
""
""  Dimostrazione di side effect indesiderato in Smalltalk.
""  La routine dangerous scopre che uno dei parametri
""  si incrementa inaspettatamente.
""
"

Object  subclass:  #sideEffect
        instanceVariableNames:  'n'
        classVariableNames:  ''
        poolDictionaries:  ''
        category:  nil !

!sideEffect class methodsFor:  '!
new:  v
 ^((super new) init:  v)
!!

!sideEffect methodsFor:  '!
init:  v
 n:=v
!
val
 ^n
!
dangerous:  b
 'b vale ' xprint.
```

```
 (b val) printNl.

 n:=n+20.

 ((b val) > 20) ifTrue:  [
   'Ehi!  La variabile b è cambiata!  Ora è:  '  xprint.
   (b val) printNl
 ]
!!
"
"" Una stampa di stringhe personalizzata
"
!String methodsFor:  'customPrinting'!
xprint
 self do:  [ :char | stdout nextPut:  char ]
!!

"
"" E adesso verifichiamo il side-effect!
"
 |a|
  a:=sideEffect new:  5.
  a dangerous:  a
!
```

Output:

```
Execution begins...
b vale 5
Ehi!  La variabile b è cambiata!  Ora è:  25
2587 byte codes executed
```

A.1.2. SideEffect in Java

```
/*
  Dimostrazione di side effect indesiderato in Java.
  La routine dangerous scopre che uno dei parametri
  si incrementa inaspettatamente.
*/

class sideEffect
{
 long n;

 sideEffect(long v) { n=v; }

 static void dangerous(sideEffect a,sideEffect b)
 {
  System.out.println("b vale "+b.n);

  a.n+=20;

  if (b.n>10)
   System.out.println("Ehi!  La variabile b è cambiata!  Ora è:  "+b.n);
 }

 public static void main(String av[])
 {
  sideEffect a;
  a=new sideEffect(5);
  dangerous(a,a);
 }
}
```

Output:

```
b vale 5
Ehi!  La variabile b è cambiata!  Ora è:  25
```

A.1.3. SideEffect in BOH

```
/*
  Dimostrazione di side effect indesiderato in BOH.
  La routine dangerous scopre che uno dei parametri
  si incrementa inaspettatamente.
*/
side:  uses system {

!sideEffect:super object
{
 n:long;

 !sideEffect(v:long):  super object() { sideEffect.n:=v; }

 !dangerous(a,b:sideEffect)
 {
  "b vale ".print;
  b.n.println;

  a.n:=a.n+20;

  if b.n>10 {
   print("Ehi!  La variabile b è cambiata!  Ora è:  ");
   b.n.println;
  }
 }
}

!main()
{
 a:=sideEffect(5);
 dangerous(a,a);
}
}
```

Output:

```
b vale 5
Ehi!  La variabile b è cambiata!  Ora è:  25
```

A.1.4. Identificatori BOH: set di caratteri esteso

```
lexicon :  uses system
{

!  ~B%#@@$?  ():long
{
 ~B%#@@$?  := 15;
}

main()
{
 println(~B%#@@$?());
}

}
```

Output:

```
15
```

A.1.5. Overloading vs. generic functions: Java

```
class figlia extends padre {}

class padre
{
 void direct(padre b) {
  System.out.println(" Invocato padre - padre");
 }

 void direct(figlia b) {
  System.out.println(" Invocato padre - figlia");
 }

 void indirect(padre b) {
  direct(b);
 }

 public static void main(String av[])
 {
  padre p=new padre();
  figlia f=new figlia();

  System.out.print(p); System.out.print(p); p.direct(p);
  System.out.print(p); System.out.print(f); p.direct(f);
  System.out.print(p); System.out.print(f); p.indirect(f);
 }
}
```

Output:

```
padre@80caf4a padre@80caf4a Invocato padre - padre
padre@80caf4a figlia@80caf4c Invocato padre - figlia
padre@80caf4a figlia@80caf4c Invocato padre - padre
```

A.1.6. Overloading vs. generic functions: BOH

```
genericVsOverload:  uses system
{
!padre:  super object {!padre():  super object() {}}
!figlia:  super padre {!figlia():  super padre() {}}


direct(a:padre,b:padre)   { println(" Invocato padre - padre"); }

direct(a:padre,b:figlia)  { println(" Invocato padre - figlia"); }

indirect(a:padre,b:padre) {
 direct(a,b);
}

main()
{
 p:=padre();
 f:=figlia();

 p.print; p.print; p.direct(p);
 p.print; f.print; p.direct(f);
 p.print; f.print; p.indirect(f);
}
}
```

Output:

```
<padre><padre> Invocato padre - padre
<padre><figlia> Invocato padre - figlia
<padre><figlia> Invocato padre - figlia
```

A.1.7. Metodi con tipi restituiti diversi in Java

```
class figlia extends padre
{
 figlia msg() {
  System.out.println(" Invocato msg di figlia");
  return (new figlia());
 }
}

class padre
{
 padre msg() {
  System.out.println(" Invocato msg di padre");
  return (new padre());
 }

 public static void main(String av[])
 {
  padre p=new padre();
  figlia f=new figlia();
  padre k;
  k=p.msg();
  k=f.msg();
 }
}
```

Output del compilatore:

```
retVal.java:2:  Method redefined with different return type:  figlia msg() was padre
msg()
 figlia msg() {
        ^
1 error
```

A.1.8. Metodi con tipi restituiti diversi in BOH

```
retVal:  uses system
{
 !padre:  super object {!padre():  super object() {}}
 !figlia:  super padre {!figlia():  super padre() {}}

msg(a:padre):padre {
 println(" Invocato msg di padre");
 msg:=a;
}

msg(a:figlia):figlia {
 println(" Invocato msg di figlia");
 msg:=a;
}

padre_o_figlia(a:padre) {
 print (" Padre o figlia, applicato a ");
 println (a);
}

speciale_per_figlia(a:figlia) {
 print (" Speciale per figlia, applicato a ");
 println (a);
}

main()
{
 p:=padre();
 f:=figlia();

 p.print;
 p.msg.padre_o_figlia;      // tipo statico di p.msg:  padre
 .nl;
 f.print;
 f.msg.speciale_per_figlia; // tipo statico di f.msg:  figlia
}
}
```

Output:

```
<padre> Invocato msg di padre
 Padre o figlia, applicato a <padre>

<figlia> Invocato msg di figlia
 Speciale per figlia, applicato a <figlia>
```

A.1.9. Trattamento dell'ambiguità in C++

```
class ambig {};
class ambig2:ambig {};

void one(ambig a,ambig2 b) {}
void one(ambig2 a,ambig b) {}

main(){
  ambig2 a=*(new ambig2);
  one(a,a);
}
```

Output del compilatore:

```
ambig.cpp:  In function ‘int main()’:
ambig.cpp:9:  call of overloaded ‘one’ is ambiguous
ambig.cpp:4:  candidates are:  one(ambig, ambig2)
ambig.cpp:5:                  one(ambig2, ambig)
```

A.1.10. Trattamento dell'ambiguità in Java

```
class ambig {}
class ambig2 extends ambig {

 static void one(ambig a,ambig2 b) {}
 static void one(ambig2 a,ambig b) {}

 static void main(){
  ambig2 a=new ambig2();
  one(a,a);
 }
}
```

Output del compilatore:

```
ambig.java:9:  Reference to one is ambiguous.  It is defined in
     void one(ambig2, ambig) and void one(ambig, ambig2).
  one(a,a);
     ^
1 error
```

A.1.11. Trattamento dell'ambiguità in BOH

```
ambigPack:  uses system {

 !ambig:  super object {!ambig():  super object(){}}
 !ambig2:  super ambig {!ambig2():  super ambig(){}}

 one(a:ambig,b:ambig2) {}
 one(a:ambig2,b:ambig) {}

 main(){
  a:=ambig2();
  one(a,a);
 }
}
```

Output del compilatore:

```
...
Phase4:
  processing...
Uhm..  definizione mancante:  one(ambig2,ambig2)..
Ci sono stati errori nella verifica dei messaggi.  Esco disgustato.

Error during compilation.
Bailing out.
```

A.2. Funzioni minime di libreria suggerite

Object:

```
object():object         // costruttore
=(object,object):bool   // uguaglianza (fra puntatori, per object)
class(object):class     // classe di appartenenza
print(object)           // stampa senza newline
println(object)         // stampa con newline
```

Num:

```
num():num               // costruttore
+(num,num):num          // somma
-(num,num):num          // differenza
*(num,num):num          // prodotto
/(num,num):num          // divisione
=(num,num):num          // uguale (numerico)
<>(num,num):num         // diverso
<(num,num):bool         // minore
<=(num,num):bool        // minore o uguale
>(num,num):bool         // maggiore
>=(num,num):bool        // maggiore o uguale
```

Int:

```
int():int               // costruttore
and(int,int):int        // bitwise and
or(int,int):int         // bitwise or
not(int):int            // bitwise not
asl(int,long):int       // shift left aritmetico
lsl(int,long):int       // shift left logico
asr(int,long):int       // shift right aritmetico
lsr(int,long):int       // shift right aritmetico
mod(int,int):int        // modulo (resto)
inc(int):int            // incremento di uno
dec(int):int            // decremento di uno
```

Real:

```
real():real             // costruttore
trunc(real):real        // troncamento
round(real):real        // arrotondamento all'intero più vicino
sin(real):real          // seno
cos(real):real          // coseno
tan(real):real          // tangente
atn(real):real          // arcotangente
log(real):real          // logaritmo naturale
exp(real):real          // e elevato al parametro
pow(real,real):real     // esponenziazione
*pi(real):real          // moltiplica per pi greco
NAN(real):bool          // Not A Number (test)
```

Bool:
```
and(bool,bool):bool     // and logico
or(bool,bool):bool      // or logico
not(bool):bool          // not logico
```

Byte:
```
byte(wordInt):byte      // parte bassa del parametro (costruttore)
byteh(wordInt):byte     // parte alta del parametro (costruttore)
byte(num):byte          // conversione (costruttore)
byte(text):byte         // conversione (costruttore)
readByte():byte         // lettura da console
```

Ubyte:
```
byte(wordInt):ubyte     // parte bassa del parametro (costruttore)
byteh(wordInt):ubyte    // parte alta del parametro (costruttore)
byte(num):ubyte         // conversione (costruttore)
byte(text):ubyte        // conversione (costruttore)
readUByte():ubyte       // lettura da console
```

Word:
```
word(longInt):word      // parte bassa del parametro (costruttore)
wordh(longInt):word     // parte alta del parametro (costruttore)
word(num):word          // conversione (costruttore)
word(text):word         // conversione (costruttore)
readWord():word         // lettura da console
```

Uword:
```
uword(longInt):uword    // parte bassa del parametro (costruttore)
uwordh(longInt):uword   // parte alta del parametro (costruttore)
uword(num):uword        // conversione (costruttore)
uword(text):uword       // conversione (costruttore)
readUWord():uword       // lettura da console
```

Long:
```
long(quadInt):long      // parte bassa del parametro (costruttore)
longh(quadInt):long     // parte alta del parametro (costruttore)
long(num):long          // conversione (costruttore)
long(text):long         // conversione (costruttore)
readLong():long         // lettura da console
```

Ulong:
```
ulong(quadInt):ulong    // parte bassa del parametro (costruttore)
ulongh(quadInt):ulong   // parte alta del parametro (costruttore)
ulong(num):ulong        // conversione (costruttore)
ulong(text):ulong       // conversione (costruttore)
readULong():ulong       // lettura da console
```

Quad:
```
quad(num):quad          // conversione (costruttore)
```

```
quad(text):quad         // conversione (costruttore)
readQuad():quad         // lettura da console
```

Uquad:

```
uquad(num):uquad        // conversione (costruttore)
uquad(text):uquad       // conversione (costruttore)
readUQuad():uquad       // lettura da console
```

Float:

```
float(num):float        // conversione (costruttore)
float(text):float       // conversione (costruttore)
readFloat():float       // lettura da console
```

Double:

```
double(num):double      // conversione (costruttore)
double(text):double     // conversione (costruttore)
readDouble():double     // lettura da console
```

Text:

```
+(text,text):text       // somma fra stringhe
+(text,object):text     // implementato come +(text,text(object))
+(object,text):text     // implementato come +(text(object),text)
text(object):text       // conversione in stringa (costruttore)
readText():text         // lettura da console
len(text):long          // numero di caratteri nel testo
                        // (se applicabile)
```

generiche:

```
nl()            // stampa un newline
panic(text)     // interrompe l'esecuzione
                   visualizzando il parametro.
```

A.3. Sorgenti dell'Implementazione Preliminare

A.3.1. File: BOH/Code0/lex0.l

```
%{
//
// code0.c - Antonio Cunei
//
// normalizza il testo in ingresso; i caratteri alfabtici vengono
// resi minuscoli, mentre i caratteri speciali vengono trasformati
// in caratteri alfabetici maiuscoli.  In questo modo viene semplificata
// la trasformazione del codice in forma di strutture C.
//
//-------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <locale.h>
#ifdef __MWERKS__
 #include <sioux.h>
 #include <console.h>
#endif
#include <string.h>

inline char normalize(int i)
{
  switch (i) {
   case ' ' :  return 'A';
   case '~' :  return 'B';
   case '|' :  return 'C';
   case '\\' :  return 'D';
   case '/' :  return 'E';
   case '?'  :  return 'F';
   case '>' :  return 'G';
   case '<' :  return 'H';
   case '+' :  return 'I';
   case '=' :  return 'J';
   case '_' :  return 'K';
   case '-' :  return 'L';
   case '*' :  return 'M';
   case '&' :  return 'N';
   case '^' :  return 'O';
   case '%' :  return 'P';
```

```
   case '$' :  return 'Q';
   case '#' :  return 'R';
   case '@' :  return 'S';
   case '!'  :  return 'T';
   default :  return tolower(i);
  }
 }
%}
%s          StrState
%%
<INITIAL>"\""           ECHO;BEGIN(StrState);
<INITIAL>.               |
<INITIAL>\n             putchar(normalize(*yytext));

<StrState>"\"\""        ECHO;
<StrState>"\""          ECHO;BEGIN(INITIAL);
<StrState>.              |
<StrState>\n            ECHO;
%%
int yywrap(void)
{
 return 1;
}

main(int ac,char *av[])
{
 short i;

#ifdef __MWERKS__
 FILE *fi,*fo;

 if ((fo=freopen("::test.out0","w",stdout)) == NULL) {
  printf ("Can't redirect stdout\n");
  exit (1);
 }
 if ((fi=freopen("::test","r",stdin)) == NULL) {
  printf ("Can't redirect stdin\n");
  exit (1);
 }
// ac=ccommand(&av);
 SIOUXSettings.autocloseonquit=FALSE;
 SIOUXSettings.asktosaveonclose=FALSE;
 SIOUXSettings.showstatusline=FALSE;
 SIOUXSettings.columns=132;
```

```
 SIOUXSettings.rows=48;
 SIOUXSettings.toppixel=40;
 SIOUXSettings.leftpixel=5;
 setlocale(LC_ALL,"");
#endif

 yylex();
}
```

A.3.2. File: BOH/Code1/lex1.l

```
%{
//
// lex1.l - Antonio Cunei
//

//  Dopo la normalizzazione:
//--------------------------
//   '~|\/?><+=_-*&^%$#@!
//   ABCDEFGHIJKLMNOPQRST

#include "y.tab.h"
int Nesting=0;
%}
SIMPLENUM           [1-9][0-9]{0,2}
%s                  StrState LongComm
%%
<INITIAL>"\""           ECHO;BEGIN(StrState);
<StrState>"\"\""        ECHO;
<StrState>"\""          ECHO;BEGIN(INITIAL);
<StrState>.              |
<StrState>\n            ECHO;

<INITIAL>"EM"           Nesting++; BEGIN(LongComm);
<LongComm>"EM"          Nesting++;
<LongComm>"ME"          if (!(--Nesting)) BEGIN(INITIAL);
<LongComm>.              |
<LongComm>\n            ;

<INITIAL,LongComm>"EE".*$       ;

<INITIAL>"operator" return OPERATOR;
<INITIAL>"uses"     return USES;
<INITIAL>","            return COMMA;
<INITIAL>":"            return COLON;
<INITIAL>"."            return DOT;
<INITIAL>";"            return SEMICOLON;
<INITIAL>"{"            return CURLY_OPEN;
<INITIAL>"}"            return CURLY_CLOSE;
<INITIAL>{SIMPLENUM}    return SIMPLENUM;
<INITIAL>[a-z0-9ABCDEFGHIJKLMNOPQRST]+  return ID;
```

```
%%
int yywrap(void)
{
 return 1;
}
```

A.3.3. File: BOH/Code1/yacc1.l

```
%{
//
//  Yacc1.y - Antonio Cunei
//
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef __MWERKS__
 #include <sioux.h>
 #include <console.h>
 #include "::includes:boh.h"
 #include <unix.h>
#else
 #include "../includes/boh.h"
#endif
#include <string.h>

#ifdef __MWERKS__
 #define lex2_proto_file  "::includes:lex2.proto"
 #define lex2_file        "::Code2:lex2.l"
 #define yacc2_proto_file "::includes:yacc2.proto"
 #define yacc2_file       "::Code2:yacc2.y"
 #define std_ops_file     "::includes:std-ops"
#else
 #define lex2_proto_file  "../includes/lex2.proto"
 #define lex2_file        "../Code2/lex2.l"
 #define yacc2_proto_file "../includes/yacc2.proto"
 #define yacc2_file       "../Code2/yacc2.y"
 #define std_ops_file     "../includes/std-ops"
#endif


extern char yytext[];
extern int yylex(void);
typedef union
{
 int    numval;
 string str;
} YYSTYPE;
```

```
FILE *f,*f1;

const char opleft='l';
const char opright='r';
const char opnonassoc='n';

typedef char optype;

struct oprec {
 short level; // da 1 a 999
 optype type;
 char name[StrLen];
 short arity;
};


void dumpOp(oprec *o)
{
#ifdef DEBUG
 fprintf(stderr,"Level %3d, type %s, arity %d, operator:  %s\n",
 o->level,(o->type==opright?"   right":(o->type==opleft?"    left":"nonassoc")),
 o->arity,o->name);
#endif
}


const short MAXOP = 400;

oprec opV[MAXOP];
short opnum;

%}
%start package
%token OPERATOR USES COMMA COLON CURLY_OPEN CURLY_CLOSE DOT
%token SEMICOLON ID SIMPLENUM
%type <str> id
%type <numval> simplenum
%%
package :  id { printf ("%s",$1); } COLON { printf (":"); } USES
          { printf ("uses"); } idlist CURLY_OPEN { printf ("{"); }
          pckbody CURLY_CLOSE { printf ("}\n"); } ;
id      :  ID { strncpy($$,yytext,StrLen); } ;
simplenum   :  SIMPLENUM { $$=atoi(yytext); } ;
idlist  :  | idlist1 id { printf("%s",$2); } ;
```

```
    idlist1 :  | idlist1 id COMMA { printf("%s,",$2); } ;
    pckbody :  rubbishlist ;
    oper    :  OPERATOR id id id simplenum SEMICOLON
               {
                 if (strcmp($2,"x")&&strcmp($2,"xx")) YYERROR; // se div.  da x e div.  da
xx
                 if (strcmp($4,"x")&&strcmp($4,"xx")) YYERROR;
                 if ((!strcmp($2,"xx"))&&(!strcmp($4,"xx"))) YYERROR; // se entrambi = xx
                 if ((!strcmp($3,"xx"))||(!strcmp($3,"x"))) YYERROR; // se ug.  a x o xx
                 if ((!strcmp($2,"xx"))&&(!strcmp($4,"x"))) opV[opnum].type=opleft; else
                 if ((!strcmp($2,"x"))&&(!strcmp($4,"xx"))) opV[opnum].type=opright; else
                 if ((!strcmp($2,"x"))&&(!strcmp($4,"x"))) opV[opnum].type=opnonassoc;
                 strncpy(opV[opnum].name,$3,StrLen);
                 opV[opnum].level=$5;
                 opV[opnum].arity=2;
                 dumpOp(&opV[opnum]);
                 if (opnum++>=MAXOP)
                 {
                  fprintf (stderr,"Too many operator definitions!\n");
                  exit (10);
                 }
               }
              | OPERATOR id id simplenum SEMICOLON
               {
                 if (strcmp($2,"x")&&strcmp($3,"x")) YYERROR; // se entrambi <> "x"
                 if ((!strcmp($2,"x"))&&(!strcmp($3,"x"))) YYERROR; // se entrambi ="x"
                 if ((!strcmp($2,"xx"))||(!strcmp($3,"xx"))) YYERROR; // se almeno uno="xx"
                 if (!strcmp($2,"x")) {
                  opV[opnum].type=opleft;
                  strncpy(opV[opnum].name,$3,StrLen);
                 } else if (!strcmp($3,"x")) {
                  opV[opnum].type=opright;
                  strncpy(opV[opnum].name,$2,StrLen);
                 }
                 opV[opnum].level=$4;
                 opV[opnum].arity=1;
                 dumpOp(&opV[opnum]);
                 if (opnum++>=MAXOP)
                 {
                  fprintf (stderr,"Too many operator definitions!\n");
                  exit (10);
                 }
               } ;
```

```
    rubbish :  CURLY_OPEN { printf("{"); } nooperrubbishlist CURLY_CLOSE { printf("}");
}
            | id { printf("%s",$1); }
            | simplenum { printf("%d",$1); }
            | SEMICOLON { printf(";"); }
            | COLON { printf(":"); }
            | DOT { printf("."); }
            | COMMA { printf(","); }
            | oper ;
    rubbishlist :  | rubbishlist rubbish ;
    nooperrubbish   :  CURLY_OPEN { printf("{"); } nooperrubbishlist
                       CURLY_CLOSE { printf("}"); }
            | id { printf("%s",$1); }
            | simplenum { printf("%d",$1); }
            | SEMICOLON { printf(";"); }
            | DOT { printf("."); }
            | COLON { printf(":"); }
            | COMMA { printf(","); } ;
    nooperrubbishlist   :  | nooperrubbishlist nooperrubbish ;
    %%
    int yyerror(const char *msg)
    {
     (void) fprintf(stderr,"%s\n",msg);
     return (0);
    }
    //
    // a parità di livello prima binari poi unari.  A parità, left, poi right, poi nonassoc.
    int oprec_compare(oprec *first,oprec *second)
    {
     short f,s;
     if (first->level<second->level) return -1;
     if (first->level>second->level) return 1;
     if (first->arity<second->arity) return 1;
     if (first->arity>second->arity) return -1;
     switch (first->type) {
      case opleft    :  f=3; break; //più alto->più forte
      case opright   :  f=2; break;
      case opnonassoc:  f=1; break;
     }
     switch (second->type) {
      case opleft    :  s=3; break;
      case opright   :  s=2; break;
      case opnonassoc:  s=1; break;
     }
```

```
 if (f<s) return -1;
 if (f>s) return 1;
 return 0;
}
//
void checkOps()
{
 // se nomi= ed arita'= -> errore.
 short i,j;
 // inefficiente, potrei usare il qsort e fare una passata sola.
 // ma per il momento può bastare.  Frattanto uso i vettori.
 for (i=0;i<opnum-1;i++)
 for (j=i+1;j<opnum;j++) {
  if ((!strcmp(opV[i].name,opV[j].name))&&(opV[i].arity==opV[j].arity)) {
   fprintf(stderr,"\n..  but operator \"%s\" (arity %d) is declared twice.\n",
            opV[i].name,opV[i].arity);
   exit(10);
  }
 }
/* qsort(opV,opnum,sizeof(oprec),
 (int(*)(const void *,const void*))oprec_compare);
 fprintf(stderr,"Operators defined so far:\n");
 for (i=0;i<opnum;i++) dumpOp(&opV[i]);
*/
}

int cmpOpName(oprec *first,oprec *second)
{
 return strcmp(first->name,second->name);
}

void dumpOpTab()
{
 int i,j,done,k,i1;
 char buf[StrLen];
 int dotPrinted=0;

 fprintf(stderr,"\n Lex section...  \n\n");
 if (!(f=fopen(lex2_proto_file,"r"))) {
  fprintf(stderr,"\n Cannot open lex2.proto!  \n\n");
  exit(10);
 }
 if (!(f1=fopen(lex2_file,"w"))) {
  fprintf(stderr,"\n Cannot open lex2.l!  \n\n");
```

```
 exit(10);
}
while (fgets(buf,StrLen,f)) {
 if (!strcmp(buf,"$$$<here>$$$\n")) break;
 fputs(buf,f1);
}
qsort(opV,opnum,sizeof(oprec),
      (int(*)(const void *,const void*))cmpOpName);
for (i=0;i<opnum;i++) {
 if ((i==0)||(strcmp(opV[i].name,opV[i-1].name))) {
  fprintf(f1,"<INITIAL>\"%s\"    reflex(return _x_%s;)\n",opV[i].name,opV[i].name);
 }
}
while (fgets(buf,StrLen,f)) {
 fputs(buf,f1);
}
fclose(f); fclose(f1);
//
fprintf(stderr,"\n Yacc section...  \n\n");

if (!(f=fopen(yacc2_proto_file,"r"))) {
 fprintf(stderr,"\n Cannot open yacc2.proto!  \n\n");
 exit(10);
}
if (!(f1=fopen(yacc2_file,"w"))) {
 fprintf(stderr,"\n Cannot open yacc2.y!  \n\n");
 exit(10);
}
while (fgets(buf,StrLen,f)) {
 if (!strcmp(buf,"$$$<here>$$$\n")) break;
 fputs(buf,f1);
}
for (i=0;i<opnum;i++) {
 if ((i==0)||(strcmp(opV[i].name,opV[i-1].name))) {
  fprintf(f1,"%%token _x_%s\n",opV[i].name);
 }
}
if (opnum>0) {
 fprintf(f1,"%%type <str>");
 for (i=0;i<opnum;i++) {
  if ((i==0)||(strcmp(opV[i].name,opV[i-1].name))) {
   fprintf(f1," _x_%s",opV[i].name);
  }
 }
```

```
 fprintf(f1,"\n");
}
fprintf(f1,"\n%%left _z_Z_z_");
qsort(opV,opnum,sizeof(oprec),
      (int(*)(const void *,const void*))oprec_compare);
for (i=0;i<opnum;i++) {
 if ((!dotPrinted) && (opV[i].level>DOTpriority)) {
  dotPrinted=1;
  fprintf (f1,"\n\n%%left DOT\n");
 }
 if ((i==0)||((opV[i].level!=opV[i-1].level)||(opV[i].type!=opV[i-1].type)
      ||(opV[i].arity!=opV[i-1].arity))) {
  fprintf(f1,"\n%s _y_%d%c%d",
  (opV[i].type==opright?"%right":(opV[i].type==opleft?"%left":"%nonassoc")),
  opV[i].level,opV[i].type,opV[i].arity);
 }
 done=0; for (k=0;k<i;k++) if (!strcmp(opV[k].name,opV[i].name)) { done=1; break; }
 if (!done) fprintf(f1," _x_%s",opV[i].name);
}
 if (!dotPrinted) {
  fprintf (f1,"\n\n%%left DOT\n");
 }
fprintf (f1,"\n%%%%\n");
for (i=0;i<opnum;i++) {
 if ((i==0)||((opV[i].level!=opV[i-1].level)||(opV[i].type!=opV[i-1].type)
      ||(opV[i].arity!=opV[i-1].arity))) j=i;
 if (opV[i].arity==1) {
  if (opV[i].type==opright)
   fprintf(f1,"expr:   _x_%s expr {sprintf($$,\"%s(%%s)\",$2);} %%prec _y_%d%c%d ;\n",
       opV[i].name,opV[i].name,opV[j].level,opV[j].type,opV[j].arity);
  else {
   fprintf(f1,"expr:   expr _x_%s {sprintf($$,\"%s(%%s)\",$1);} %%prec _y_%d%c%d ;\n",
       opV[i].name,opV[i].name,opV[j].level,opV[j].type,opV[j].arity);
// caso speciale:  devo costruire le regole per un op.unario
// prefix+op.unario postfix se la prec.  del secondo >
// NB: qui sono ordinati per precedenza, e i binari prima degli unari, e i left
// prima dei right.  Quindi per op1 x e x op2 con pari priorità,
// la regola extra NON viene generata.
/////
// per levare i conflitti reduce/reduce, commentare questo pezzetto.
    for (i1=0;i1<i;i1++) {
     if ((opV[i1].arity==1)&&(opV[i1].type==opright)) {
      fprintf(f1,"expr:   _x_%s",opV[i1].name);
      fprintf(f1," _x_%s ",opV[i].name);
```

```
          fprintf(f1,"{sprintf($$,\"%s(%s)\");} %%prec _y_%d%c%d ;\n",
            opV[i].name,opV[i1].name,opV[j].level,opV[j].type,opV[j].arity);
         }
        }
    // fin qui.
    /////
       }
       } else {
        fprintf(f1,
         "expr:   expr _x_%s expr {sprintf($$,\"%s(%%s,%%s)\",$1,$3);} %%prec _y_%d%c%d
;\n",
         opV[i].name,opV[i].name,opV[j].level,opV[j].type,opV[j].arity);
    // caso speciale:  devo costruire le regole per un op.unario prefix+op.binario
    // se la prec.  del secondo >
    // NB: qui sono ordinati per precedenza, e i binari prima degli unari.
    // COMUNQUE, per il modo in cui è realizzato il parsing in yacc2.y,
    // riduce sempre prima gli unari prima dei binari;
    // aggiungere regole esplicite porta solo a conflitti di parsing inutili.
        for (i1=0;i1<i;i1++) {
         if ((opV[i1].arity==1)&&(opV[i1].type==opright)) {
          fprintf(f1,"expr:   _x_%s",opV[i1].name);
          fprintf(f1," _x_%s expr ",opV[i].name);
          fprintf(f1,"{sprintf($$,\"%s(%s,%%s)\",$3);} %%prec _y_%d%c%d ;\n",
            opV[i].name,opV[i1].name,opV[j].level,opV[j].type,opV[j].arity);
         }
        }
       }
      }
      if (opnum>0) {
       qsort(opV,opnum,sizeof(oprec),
             (int(*)(const void *,const void*))cmpOpName);
       fprintf(f1,"everyOp :  ");
       for (i=0;i<opnum;i++) {
        if ((i==0)||(strcmp(opV[i].name,opV[i-1].name))) {
        if(i!=0) fprintf(f1," %%prec _z_Z_z_ |\n\t\t");
        fprintf(f1,"_x_%s",opV[i].name);
        }
       }
       fprintf(f1," %%prec _z_Z_z_;\n");
      }
     while (fgets(buf,StrLen,f)) {
      fputs(buf,f1);
     }
     fclose(f); fclose(f1);
```

```
}
//
void main(int ac,char *av[])
{
 extern int yyparse();
 string buf;

#ifdef __MWERKS__
 FILE *fi,*fo;

// ac=ccommand(&av);
 SIOUXSettings.autocloseonquit=FALSE;
 SIOUXSettings.asktosaveonclose=FALSE;
 SIOUXSettings.showstatusline=FALSE;
 SIOUXSettings.columns=132;
 SIOUXSettings.rows=48;
 SIOUXSettings.toppixel=40;
 SIOUXSettings.leftpixel=5;
 setlocale(LC_ALL,"");

 if ((fo=freopen("::test.out1","w",stdout)) == NULL) {
  fprintf (stderr,"Can't redirect stdout\n");
  fflush(stdout);
  exit (10);
 }
 if ((fi=freopen("::test.out0","r",stdin)) == NULL) {
  fprintf (stderr,"Can't redirect stdin\n");
  fflush(stdout);
  exit (10);
 }
#endif

 opnum=0;

 if (!(f=fopen(std_ops_file,"r"))) {
  fprintf (stderr,"Warning:  file \"std-ops\" not found.\n");
 } else {
  fprintf (stderr,"Reading in \"std-ops\"...\n");
  while (!(feof(f))) {
   fscanf(f,"%d %d %c %s",&opV[opnum].level,
     &opV[opnum].arity,&opV[opnum].type,opV[opnum].name);
   if (!(strcmp("",opV[opnum].name))) break;
   dumpOp(&opV[opnum]);
   opnum++;
```

```
  }
  fprintf(stderr,"--\n");
  fclose(f);
 }

 if (yyparse()) {
  fprintf (stderr,"Sorry, there's a syntax error.\n");
  exit(10);
 } else {
  fprintf (stderr,"\nOperator scan completed successfully.\n");
  checkOps();
  dumpOpTab();
//
#ifdef __MWERKS__
  fclose (fi);
#endif
 }
}
```

A.3.4. File: BOH/includes/lex2.proto

```
%{
//
// lex2.proto - Antonio Cunei
//

//  Dopo la normalizzazione:
//--------------------------
//   '~|\/?><+=_-*&^%$#@!
//   ABCDEFGHIJKLMNOPQRST

#ifdef __MWERKS__
 #include "::includes:boh.h"
 #include <unix.h>
#else
 #include "../includes/boh.h"
#endif
#include "y.tab.h"
#include <string.h>
char lastID[StrLen]="";
char lastStr[StrLen]="";
int Nesting=0;
#define reflex(action)  if (!strcmp(lastID,"")) \
           { strcpy(yylval.str,yytext); action} \
          else { \
           yyless(0);{strcpy(yylval.str,lastID);strcpy(lastID,""); return ID; \
          }}

#define reflex2(action)  if (!strcmp(lastID,"")) \
           {action} \
          else { \
           REJECT; \
          }

char *normalizeNum(char *s);

%}
D1          [0-9]+
D0          [0-9]*
IDL         [a-z0-9A-T]
QL          [bwlq]
FQL         [ef]
```

```
NUMINT          {D1}[u]?{QL}
INT             {D1}
NUMB            ({D1}{FQL}|{D1}{FQL}{D1}|{D1}{FQL}"L"{D1}|{D1}{FQL}"L("{D1}")")
NUMFP           ({INT}"."{INT}|{INT}"("{INT}")"|{INT}"."{NUMB}|{NUMB}"("{INT}")"|{NUMB})

%s          StrState
%%
<INITIAL>"\""           reflex(strcpy(lastStr,yytext);BEGIN(StrState);)
<StrState>"\"\""        strcat(lastStr,yytext);
<StrState>"\""          {strcat(lastStr,yytext);strcpy(yylval.str,lastStr);
                         BEGIN(INITIAL);return STRING;}
<StrState>.              |
<StrState>\n            strcat(lastStr,yytext);

<INITIAL>"operator"     reflex(return OPERATOR;)
<INITIAL>"uses"         reflex(return USES;)
<INITIAL>","            reflex(return COMMA;)
<INITIAL>":"            reflex(return COLON;)
<INITIAL>";"            reflex(return SEMICOLON;)
<INITIAL>"{"            reflex(return CURLY_OPEN;)
<INITIAL>"}"            reflex(return CURLY_CLOSE;)
<INITIAL>({NUMINT}|{INT})   reflex2(strcpy(yylval.str,yytext); return NUM;)
<INITIAL>{NUMFP}        reflex2(strcpy(yylval.str,normalizeNum(yytext)); return NUM;)

<INITIAL>"super"        reflex(return SUPER;)
<INITIAL>".."           reflex(return DOTDOT;)
<INITIAL>":J"           reflex(return ASSIGNMENT;)
<INITIAL>"if"           reflex(return IF;)
<INITIAL>"else"         reflex(return ELSE;)
<INITIAL>"elsif"        reflex(return ELSIF;)
<INITIAL>"while"        reflex(return WHILE;)
<INITIAL>"const"        reflex(return CONST;)
<INITIAL>"for"          reflex(return FOR;)
<INITIAL>"["            reflex(return SQUARE_OPEN;)
<INITIAL>"]"            reflex(return SQUARE_CLOSE;)
<INITIAL>"N"            reflex(return AMPERSAND;)
<INITIAL>"T"            reflex(return BANG;)
<INITIAL>"("            reflex(return PAR_OPEN;)
<INITIAL>")"            reflex(return PAR_CLOSE;)
<INITIAL>"."            reflex(return DOT;)

$$$<here>$$$
```

```
<INITIAL>{IDL}              strcat(lastID,yytext);
<INITIAL>[ \t\r\v\f\n]      reflex(ECHO;)
<INITIAL>.                   return ERROR;

%%
int yywrap(void)
{
 return 1;
}

// normalize:  riduce i numeri FP a formato standard.
// i formati possibili sono:
//
//   999e
//   999e99
//   999.9      -> 9(999)
//   999.9e     -> 9e(999)
//   999e-99        -> 999e-(99)
//   999.9e99       -> 9e99(999)
//   999.9e-99      -> 9e-(99)(999)
//   999.9e-(99)    -> 9e-(99)(999)
//   9e-99(999)     -> 9e-(99)(999)
//   9(999)
//   9e(999)
//   999e-(99)
//   9e99(999)
//   9e-(99)(999)

string buf2;

char *normalizeNum(char *s)
{
 string buf;
 char *q,*p=s;

 while (*p) {
  if (*p=='.')  break;
  p++;
 }
 if (*p=='.')  {
  *p++='\0';
  strcpy(buf,p);
  strcat(buf,"(");
  strcat(buf,s);
```

```
  strcat(buf,")");
 } else strcpy(buf,s);

// ora buf contiene qualcosa del tipo B(A), A(A) oppure B
// con A ::= [0-9]+ e B ::= Ae|AeA|Ae-A|Ae-(A)

 p=buf;
 while (*p) {
  if (*p=='L') break;
  p++;
 }

// visto che il lex ha gia' fatto il pattern matching,
// c'e' sempre almeno un carattere dopo "L" ("-");
// per il short circuit, se *p=='\0', non viene
// valutato *(p+1).

 if ((*p=='L') && (*(p+1)!='(')) {
  *p++='\0';
  strcpy(buf2,buf);
  strcat(buf2,"L(");
  q=p;
  while (*p) {
   if (*p=='(') {
    break;
   }
   p++;
  }
  strncat(buf2,q,p-q);
  strcat(buf2,")");
  strcat(buf2,p);
 } else strcpy(buf2,buf);

// ora buf2 contiene qualcosa del tipo B(A), A(A) oppure B
// con A ::= [0-9]+ e B ::= Ae|AeA|Ae-(A)

 return buf2;
}
```

A.3.5. File: BOH/includes/yacc2.proto

```
%{
//
// yacc2.proto - Antonio Cunei
//
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef __MWERKS__
 #include <sioux.h>
 #include <console.h>
 #include "::includes:boh.h"
#else
 #include "../includes/boh.h"
#endif
#include <string.h>

extern char yytext[];
extern int yylex(void);
%}
%union
{
 string str;
};
%token OPERATOR USES COMMA COLON SEMICOLON CURLY_OPEN CURLY_CLOSE NUM ID
%token SUPER DOTDOT DOT ASSIGNMENT IF ELSE ELSIF WHILE SQUARE_OPEN FOR
%token SQUARE_CLOSE AMPERSAND BANG PAR_OPEN PAR_CLOSE STRING ERROR CONST
%start package
%type <str> ID NUM expr idList idList1 usesList id everyOp
%type <str> STRING superList idListOne type nonVectType vectDim
%type <str> paramDefList1 methHeader methReturnVal varDef varDecl
%type <str> mayBeNegativeNum exprListTwo globVarDecl index indexList field
%token _x_L

%right PAR_OPEN SQUARE_OPEN

$$$<here>$$$

id :  ID | everyOp; // default:  $1 -> $$

idList      :  {strcpy($$,"");}| idList1 id {sprintf($$,"%s%s",$1,$2);} ;
```

```
idList1     :  {strcpy($$,"");}| idList1 id COMMA {sprintf($$,"%s%s,",$1,$2);};

package     :  id COLON
               usesList CURLY_OPEN {printf("%s :  %s {",$1,$3);}
               packageBody CURLY_CLOSE {puts("}\n");} ;
usesList    :  {strcpy($$,"");}| USES idList {sprintf($$,"uses %s",$2);};

packageBody :  | packageBody packageItem;
packageItem :  globalVar | classDef | methDef ;

globalVar   :  globVarDecl SEMICOLON {printf("%s;",$1);};
globalVar   :  CONST globVarDecl SEMICOLON {printf("const %s;",$2);};

globVarDecl :  id ASSIGNMENT expr {sprintf($$,"%s:J%s",$1,$3);};
varDecl     :  globVarDecl
        | field ASSIGNMENT expr {sprintf($$,"%s:J%s",$1,$3);}
        | field indexList ASSIGNMENT expr {sprintf($$,"%s%s:J%s",$1,$2,$4);};

index       :  SQUARE_OPEN expr SQUARE_CLOSE {sprintf($$,"[%s]",$2);};
indexList   :  index | indexList index {sprintf($$,"%s%s",$1,$2);};


expr        :  PAR_OPEN expr PAR_CLOSE {sprintf($$,"(%s)",$2);};
expr        :  field ;
expr        :  NUM | id | STRING ; // default:  $1 -> $$
expr        :  id PAR_OPEN PAR_CLOSE
               {sprintf($$,"%s()",$1);};
field       :  id PAR_OPEN expr PAR_CLOSE
               {sprintf($$,"%s(%s)",$1,$3);};
expr        :  id PAR_OPEN exprListTwo PAR_CLOSE
               {sprintf($$,"%s(%s)",$1,$3);};
expr        :  field indexList
               {sprintf($$,"%s%s",$1,$2);};
field       :  expr DOT id
               {sprintf($$,"%s(%s)",$3,$1);};
expr        :  expr DOT id PAR_OPEN expr PAR_CLOSE
               {sprintf($$,"%s(%s,%s)",$3,$1,$5);};
expr        :  expr DOT id PAR_OPEN exprListTwo PAR_CLOSE
               {sprintf($$,"%s(%s,%s)",$3,$1,$5);};
expr        :  expr DOT id PAR_OPEN PAR_CLOSE
               {sprintf($$,"%s(%s)",$3,$1);};
expr        :  DOT id
               {sprintf($$,"%s()",$2);};
```

```
    exprListTwo :  expr COMMA expr {sprintf($$,"%s,%s",$1,$3);}
                | exprListTwo COMMA expr {sprintf($$,"%s,%s",$1,$3);} ;

    //
    classDef    :  BANG id COLON superList CURLY_OPEN {printf ("T %s :  %s {",$2,$4);}
                          classTail {printf("}");};
    classDef    :  id COLON superList CURLY_OPEN {printf ("%s :  %s {",$1,$3);}
                          classTail {printf("}");};
    superList   :  {strcpy($$,"");}| SUPER idList {sprintf($$,"super %s",$2);};
    classTail   :  varDefList methDefList CURLY_CLOSE | varDefList CURLY_CLOSE ;
    varDefList  :  | varDefList varDef SEMICOLON {printf("%s ;",$2);};
    varDef      :  idListOne COLON type {sprintf ($$,"%s:%s ",$1,$3);} ;
    idListOne   :  id | idListOne COMMA id {sprintf($$,"%s,%s",$1,$3);};

    type        :  nonVectType
                | type SQUARE_OPEN vectDim SQUARE_CLOSE {sprintf($$,"%s[%s]",$1,$3);};
    nonVectType :  id ;

    mayBeNegativeNum:  NUM | _x_L NUM {sprintf($$,"L(%s)",$2);} ;
    vectDim     :  {strcpy($$,"");}| mayBeNegativeNum
                | mayBeNegativeNum DOTDOT mayBeNegativeNum {sprintf($$,"%s..%s",$1,$3);};

    methDefList :  methDef | methDefList methDef ;

    methDef         :  BANG id methHeader CURLY_OPEN {printf("T %s %s {",$2,$3);}
                       methBody CURLY_CLOSE {printf("}");};
    methDef         :  id methHeader CURLY_OPEN {printf("%s %s {",$1,$2);}
                       methBody CURLY_CLOSE {printf("}");};
    paramDefList1   :  varDef | paramDefList1 COMMA varDef {sprintf($$,"%s,%s",$1,$3);}
;
    methHeader      :  PAR_OPEN paramDefList1 PAR_CLOSE
                       methReturnVal {sprintf($$,"(%s)%s",$2,$4);};
    methHeader      :  PAR_OPEN PAR_CLOSE methReturnVal {sprintf($$,"()%s",$3);};
    methReturnVal   :  {strcpy($$,"");}| COLON nonVectType {sprintf($$,":%s",$2);}
            | COLON SUPER expr {sprintf($$,":super %s",$3);}
            | COLON SUPER exprListTwo {sprintf($$,":super %s",$3);};

    methBody        :  varDefList cmdList | varDefList;
    cmdList         :  cmd | cmdList cmd ;
    cmd             :  varDecl SEMICOLON {printf("%s;",$1);}
                    | expr SEMICOLON {printf("%s;",$1);}
                    | expr AMPERSAND SEMICOLON {printf("%s &;",$1);}
                    | expr AMPERSAND id SEMICOLON {printf("%s N %s ;",$1,$3);}
                    | CONST varDecl SEMICOLON {printf("const %s;",$2);}
```

```
                | ifHead cmdList CURLY_CLOSE {printf("}");}
                | ifHead cmdList CURLY_CLOSE ELSE CURLY_OPEN {printf("} else {");}
                   cmdList CURLY_CLOSE {printf("}");}
                | CURLY_OPEN {printf("{");} cmdList CURLY_CLOSE {printf("}");}
                | WHILE expr CURLY_OPEN {printf("while %s {",$2);}
                   cmdList CURLY_CLOSE {printf("}");}
                | FOR {printf("for");} cmd expr SEMICOLON {printf("%s;",$4);}
                   cmd CURLY_OPEN
                  {printf("{");} cmdList CURLY_CLOSE {printf("}");} ;
ifHead             :  IF expr CURLY_OPEN {printf("if %s {",$2);}
                | ifHead cmdList CURLY_CLOSE ELSIF expr CURLY_OPEN
                   {printf("} elsif %s {",$5);};

%%
int yyerror(const char *msg)
{
 (void) fprintf(stderr,"%s\n",msg);
 return (0);
}
//
void main(int ac,char *av[])
{
 extern int yyparse();
 string buf;

#ifdef __MWERKS__
 FILE *fi,*fo;

// ac=ccommand(&av);
 SIOUXSettings.autocloseonquit=FALSE;
 SIOUXSettings.asktosaveonclose=FALSE;
 SIOUXSettings.showstatusline=FALSE;
 SIOUXSettings.columns=132;
 SIOUXSettings.rows=48;
 SIOUXSettings.toppixel=40;
 SIOUXSettings.leftpixel=5;
 setlocale(LC_ALL,"");

 if ((fo=freopen("::test.out2","w",stdout)) == NULL) {
  fprintf (stderr,"Can't redirect stdout\n");
  fflush(stdout);
  exit (1);
 }
 if ((fi=freopen("::test.out1","r",stdin)) == NULL) {
```

```
  fprintf (stderr,"Can't redirect stdin\n");
  fflush(stdout);
  exit (1);
 }
#endif

 if (yyparse()) {
  fprintf (stderr,"Sorry, there's a syntax error.\n");
  exit(10);
 } else {
  fprintf (stderr,"Operator decoding pass completed successfully.\n");
//
#ifdef __MWERKS__
  fclose (fi);
//  if (remove("::test.out1"))
//   fprintf (stderr,"Warning:  Cannot remove temporary file.\n");
#endif

 }
}
```

A.3.6. File: BOH/includes/defs34.h

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef __MWERKS__
 #include <sioux.h>
 #include <console.h>
 #include "::includes:boh.h"
#else
 #include "../includes/boh.h"
#endif
#include <string.h>

const short maxDimNum=5;

struct Dim {
 long start,end;
};

struct Type {
 string name;
 short dimNum;
 Dim dim[maxDimNum];
};

struct VarDef {
 string name;
 Type *type;
 struct VarDef *next;
};

struct ClassDef {
 string name,super; // uno solo
 short Library;
 short Public;
 VarDef *vars;
 struct ClassDef *next;
 short scanned;
};
//
const short maxParams=10;
```

```
// Val ritorno sempre presente (ev.  "" oppure "super")
// "" per i metodi non funzionali; "super" per le inizializzazioni.
struct Method {
 string par[maxParams];
 ClassDef *Class; // oppure NULL

 string ret;
 struct Method *next;
};

// un messaggio; nome+arity.
struct Message {
 string name;
 short arity;
 short Public;
 struct Method *meths;
 struct Message *next;
};

//////////
//////////
//
// Per la parte di produzione del codice:
//

struct exprType {
 string str;
 string type;
 short simple;
};

//--
// un elenco di parametri:  nomi e tipi
//
struct paramListType {
 short numParams;
 string paramNames[maxParams];
 string paramTypes[maxParams];
};

//--
// un elenco di indici
//
```

```
struct indexListType {
 short numIndexes;
 string indexes[maxDimNum];
};

//--
//
// per il parsing
//
struct quadVal {
 unsigned long high,low;
};

typedef enum {K_byte,K_ubyte,K_word,K_uword,
          K_long,K_ulong,K_quad,K_uquad} intKind;
```

A.3.7. File: BOH/Code3/lex3.l

```
%{
//
// lex3.l - Antonio Cunei
//
#ifdef __MWERKS__
 #include "::includes:defs34.h"
#else
 #include "../includes/defs34.h"
#endif
#include "y.tab.h"
#define reflex(action)  { ECHO; action }

string lastStr="";
string lastID="";
%}
D1          [0-9]+
D0          [0-9]*
IDL                     [a-z0-9A-T]
QL                      [bwlq]
FQL                     [ef]

NUMINT                  {D1}[u]?{QL}
INT                     {D1}
NUMB                    ({D1}{FQL}|{D1}{FQL}{D1}|{D1}{FQL}"L("{D1}")")
NUMFP                   ({INT}"("{INT}")"|{NUMB}"("{INT}")"|{NUMB})

NUM         ({NUMINT}|{INT}|{NUMFP})

%s          StrState
%%
<INITIAL>"\""           strcpy(lastStr,yytext);BEGIN(StrState);
<StrState>"\"\""        strcat(lastStr,yytext);
<StrState>"\""          {strcat(lastStr,yytext);strcpy(yylval.str,lastStr);
                         BEGIN(INITIAL);printf("%s",lastStr);return STRING;}
<StrState>.              |
<StrState>\n            strcat(lastStr,yytext);

<INITIAL>"uses"         reflex(return USES;)
<INITIAL>","            reflex(return COMMA;)
<INITIAL>":"            reflex(return COLON;)
<INITIAL>";"            reflex(return SEMICOLON;)
```

```
<INITIAL>"{"            reflex(return CURLY_OPEN;)
<INITIAL>"}"            reflex(return CURLY_CLOSE;)
<INITIAL>{NUM}          reflex(strcpy(yylval.str,yytext);return NUM;)

<INITIAL>"super"        reflex(return SUPER;)
<INITIAL>".."           reflex(return DOTDOT;)
<INITIAL>":J"           reflex(return ASSIGNMENT;)
<INITIAL>"if"           reflex(return IF;)
<INITIAL>"else"         reflex(return ELSE;)
<INITIAL>"elsif"        reflex(return ELSIF;)
<INITIAL>"while"        reflex(return WHILE;)
<INITIAL>"const"        reflex(return CONST;)
<INITIAL>"for"          reflex(return FOR;)
<INITIAL>"["            reflex(return SQUARE_OPEN;)
<INITIAL>"]"            reflex(return SQUARE_CLOSE;)
<INITIAL>"N"            reflex(return AMPERSAND;)
<INITIAL>"T"            reflex(return BANG;)
<INITIAL>"("            reflex(return PAR_OPEN;)
<INITIAL>")"            reflex(return PAR_CLOSE;)

<INITIAL>{IDL}+         {if (!strncmp(yytext,"___",3)) return ERROR;
                else {strcpy(yylval.str,yytext);printf("%s",yytext);return ID;}}
<INITIAL>[ \t]+         printf(" ");
<INITIAL>[ \t]*[\r\v\f\n]+([ \t]*[\r\v\f\n]+)*[ \t]*    printf("\n");
<INITIAL>.               return ERROR;

%%
int yywrap(void)
{
 return 1;
}
```

A.3.8. File: BOH/Code3/yacc3.y

```
%{
//
// yacc3.y - Antonio Cunei
//
#ifdef __MWERKS__
 #include "::includes:defs34.h"
 #define defFile "::test.out3.def"
 #define libDefFile "::includes:library.def"
#else
 #include "../includes/defs34.h"
 #define defFile "../test.out3.def"
 #define libDefFile "../includes/library.def"
#endif
//
extern char yytext[];
extern int yylex(void);

Method currentMethod;
short currentMethodParNum;

ClassDef *ClassDefList=NULL;
ClassDef *CurrentClass=NULL;

Message *MessageList=NULL;

#define MaxLibraryTypes 50

string libraryTypes[MaxLibraryTypes];

int fail(const char *msg);
%}
%union
{
 string str;
 long   num;
 VarDef *VarDefPtr;
 Type *type;
 Dim dim;
};
%token USES COMMA COLON SEMICOLON CURLY_OPEN CURLY_CLOSE NUM ID
%token SUPER DOTDOT ASSIGNMENT IF ELSE ELSIF WHILE SQUARE_OPEN FOR
```

```
%token SQUARE_CLOSE AMPERSAND BANG PAR_OPEN PAR_CLOSE STRING ERROR CONST
%start package
%type <str> superList id NUM ID
%type <num> num mayBeNegativeNum parNameList1
%type <VarDefPtr> varDef varNameListOne
%type <type> nonVectType type
%type <dim> vectDim
%%
package     :  id COLON usesList CURLY_OPEN packageBody CURLY_CLOSE {};
id          :  ID ;
usesList    :  | USES idList ;
idList      :  | idList1 ;
idList1     :  id {} | idList1 COMMA id ;

packageBody :  | packageBody packageitem ;
packageitem :  globalVar | classDef | methDef ;


globalVar       :  globVarDecl SEMICOLON ;
globalVar       :  CONST globVarDecl SEMICOLON ;

index           :  SQUARE_OPEN expr SQUARE_CLOSE ;
indexList       :  index | indexList index ;

globVarDecl     :  id ASSIGNMENT expr {} ;

classDef    :  BANG id COLON superList CURLY_OPEN
                      { CurrentClass=addClassDef($2,$4,1); }
                      classTail { CurrentClass=NULL; };
classDef    :  id COLON superList CURLY_OPEN
                      { CurrentClass=addClassDef($1,$3,0); }
                      classTail { CurrentClass=NULL; };

superList   :  {strcpy($$,"");}| SUPER id {strncpy($$,$2,StrLen);};

classTail   :  classVarDefList CURLY_CLOSE | classVarDefList methDefList CURLY_CLOSE
;
classVarDefList :
        | classVarDefList varDef SEMICOLON {VarDefAppend(CurrentClass->vars,$2);} ;
varDef      :  varNameListOne COLON type {SetVarListType($1,$3);$$=$1;};
varNameListOne  :  id {$$=SetupVar($1);}
            | varNameListOne COMMA id {$$=SetupVar($3); $$->next=$1;};
//
type        :  nonVectType
```

```
            | type SQUARE_OPEN vectDim SQUARE_CLOSE {BumpDim($1,&$3); $$=$1;};
    nonVectType :  id {$$=SetupType($1);};
    mayBeNegativeNum :  num | id PAR_OPEN num
                  PAR_CLOSE {if (strcmp("L",$1)) {fail("Invalid array index.");} $$=-$3;}
;
    vectDim     :  {fail("Flexible vectors not allowed..Sorry.");}
                | mayBeNegativeNum {if ($1<1) fail("Dimension is less than 1.");
                      $$.start=0; $$.end=$1-1;}
                | mayBeNegativeNum DOTDOT mayBeNegativeNum
       {if ($3<$1) fail("lower bound greater than upper bound."); $$.start=$1; $$.end=$3;};
    num         :  NUM { $$=getNum($1);};

    varDecl         :  globVarDecl
                    | field ASSIGNMENT expr {} ;
                    | field indexList ASSIGNMENT expr {} ;

    expr        :  field {} ;
    expr        :  PAR_OPEN expr PAR_CLOSE ;
    expr        :  NUM {}| id {}| STRING {};
    expr        :  id PAR_OPEN PAR_CLOSE {};
    field       :  id PAR_OPEN expr PAR_CLOSE {};
    expr        :  id PAR_OPEN exprList2 PAR_CLOSE {};
    expr        :  field indexList {};
    //expr      :  field SQUARE_OPEN expr SQUARE_CLOSE {};
    exprList2   :  expr COMMA expr | exprList2 COMMA expr ;


    methDefList :  methDef | methDefList methDef ;

    ////

    methDef         :  BANG id methHeader {currentMethod.Class=CurrentClass;}
                       CURLY_OPEN methTail {addMethHeader($2,1);} ;
    methDef         :  id methHeader {currentMethod.Class=CurrentClass;}
                       CURLY_OPEN methTail {addMethHeader($1,0);} ;
    // i parametri possono essere solo oggetti.
    paramDeclList   :  | paramDeclList1 ;
    paramDeclList1  :  paramDecl | paramDeclList1 COMMA paramDecl ;
    paramDecl       :  parNameList1 COLON id {if (currentMethodParNum+$1>maxParams)
                       {fail("Too many parameters.");}
                      { short kk; for (kk=0;kk<$1;kk++)
                       strncpy(currentMethod.par[currentMethodParNum++],$3,StrLen);}} ;
    parNameList1    :  id {$$=1;} | parNameList1 COMMA id {$$=$1+1;};
```

```
    methHeader      :  PAR_OPEN {currentMethodParNum=0;} paramDeclList
                        PAR_CLOSE methReturnVal ;
    methReturnVal   :  {strcpy(currentMethod.ret,"");}
                    | COLON id {strncpy(currentMethod.ret,$2,StrLen);}
                    | COLON SUPER expr {strcpy(currentMethod.ret,"super");};
                       // una expr sola, in qs.  versione..

    methVarDeclList :  | methVarDeclList methVarDecl SEMICOLON ;
    methVarDecl     :  methVarNameListOne COLON type ;
    methVarNameListOne  :  id {}| methVarNameListOne COMMA id ;

    methTail        :  methVarDeclList cmdList CURLY_CLOSE | methVarDeclList CURLY_CLOSE
;
    cmdList         :  cmd | cmdList cmd ;
    cmd             :  varDecl SEMICOLON | expr SEMICOLON
                    | expr AMPERSAND SEMICOLON
                    | expr AMPERSAND id SEMICOLON
                    | WHILE expr CURLY_OPEN cmdList CURLY_CLOSE
                    | ifHead cmdList CURLY_CLOSE
                    | ifHead cmdList CURLY_CLOSE ELSE CURLY_OPEN cmdList CURLY_CLOSE
                    | CURLY_OPEN cmdList CURLY_CLOSE
                    | FOR cmd expr SEMICOLON cmd CURLY_OPEN
                      cmdList CURLY_CLOSE ;
    ifHead          :  IF expr CURLY_OPEN
                    | ifHead cmdList CURLY_CLOSE ELSIF expr CURLY_OPEN ;


    %%
    //-----------------------------------------------
    //
    //  Costruzione delle strutture dati
    //
    //-----------------------------------------------
    //
    ClassDef *addClassDef(char *name,char *super,short Pub)
    {
     ClassDef *p=ClassDefList;

     if (!p) p=ClassDefList=new ClassDef;
     else {
      while (p->next) p=p->next;
      p->next=new ClassDef;
      p=p->next;
     }
```

```
 strncpy(p->name,name,StrLen);
 strncpy(p->super,super,StrLen);
 p->Public=Pub;
 p->next=NULL;
 p->vars=NULL;
 p->scanned=0;
// fprintf(stderr,"Class definition:  \"%s\", superclass \"%s\", %s.\n",
//         name,super,Pub?"public":"private");

 return p;
}
//
long getNum(char *St)
{
 // solo interi (al max con 1 o 2 caratteri (di cui il primo 'u') di trailing.)
 long res=0;
 long pos=0;
 long sign=1;

 if (St[0]=='-') {
  sign=-1;
  pos++;
 }
 if(!isdigit(St[pos])) fail("Illegal number");
 while (isdigit(St[pos])) {
  res=res*10+St[pos++]-'0';
 }
 if (St[pos]=='\0') return sign*res;
 if (St[pos]=='u') {
  if (St[pos+1]=='\0') return sign*res;
  if (!isalpha(St[pos+1])) fail("Illegal number");
  if (St[pos+2]=='\0') return sign*res;
  fail("Illegal number");
 }
 if (!isalpha(St[pos])) fail("Illegal number");
 if (St[pos+1]=='\0') return sign*res;
 fail("Illegal number");
}
//
void VarDefAppend(VarDef *&start,VarDef *it)
{
 VarDef *p=start;

 if (!p) start=it; else {
```

```
  while (p->next) p=p->next;
  p->next=it;
 }
}
//
VarDef *SetupVar(char *it)
{
 VarDef *vd=new VarDef;
 strncpy (vd->name,it,StrLen);
 vd->type=NULL;
 vd->next=NULL;
 return vd;
}
//
Type *SetupType(char *name)
{
 Type *tp=new Type;
 strncpy (tp->name,name,StrLen);
 tp->dimNum=0;
 return tp;
}
//
void BumpDim(Type *tp,Dim *dim)
{
 if (tp->dimNum==maxDimNum-1) fail("Too many dimensions for this array.");
 tp->dim[tp->dimNum++]=*dim;
}
//
void SetVarListType(VarDef *vd,Type *tp)
{
 while (vd) {
  vd->type=tp;
  vd=vd->next;
 }
}
//
void addMethHeader(char *name,short Pub)
{
 Message *m;
 Method *mt;
 short i;short found;
 // aggiunge:  currentMethod, di arità currentMethodParNum, al messaggio
 // name (che ev.  crea).  deve duplicare currentMethod in un nuovo Method.
 // currentMethodParNum -> arity.  Qui può verificare la coerenza di funzione.
```

```
if (!currentMethod.Class) {
// per prima cosa controlla se cerco di definire un metodo super globale (illegale)
 if (!strcmp(currentMethod.ret,"super"))
  fail("Cannot define \"super\" method outside a class definition.");
} else {
// deve controllare che torni super <-> solo se il super della classe è non nullo.
// NB: TUTTE le classi, a parte object, dovrebbero avere almeno una sopraclasse, in
// modo che tutti gli oggetti (istanze di sottoclassi di object) possano essere
// trattati nello stesso modo.
 if ((!strcmp(currentMethod.Class->super,""))&&(!strcmp(currentMethod.ret,"super")))
 {
  fail("Cannot define \"super\" method:  class has no superclass.");
 }
}
// verifichiamo se pubblico/privato è uguale agli altri con medesimo nome.
m=MessageList;

while (m) {
 if (!strcmp(m->name,name)) break;
 m=m->next;
}
// ha lo stesso valore di "pubblicità"?
if (m) {
 if (m->Public!=Pub) fail("Should be private instead of public or viceversa.");
}

//
// dapprima cerca il messaggio con la relativa arità.
m=MessageList;
while (m) {
 if ((!strcmp(m->name,name))&&(m->arity==currentMethodParNum)) break;
 m=m->next;
}
if (!m) { // se non c'è crealo.
 m=new Message;
 strncpy(m->name,name,StrLen);
 m->arity=currentMethodParNum;
 m->meths=NULL;
 m->next=MessageList;
 m->Public=Pub;
 MessageList=m;
}
// ci siamo.  Cerca questa combinazione di parametri.
```

```
 // se la trova, errore.  Infatti:  se messaggio non funzionale,
 // è una duplicazione; se funzione e ritorno uguale, duplicazione,
 // sennò errore lo stesso.
 mt=m->meths;
 while (mt) {
  found=1;
  for (i=0;i<currentMethodParNum;i++)
   if (strcmp(mt->par[i],currentMethod.par[i])) {found=0;break;}
  if (found) fail("Method redefined.");
  mt=mt->next;
 }
 mt=new Method;
 *mt=currentMethod;
 mt->next=m->meths;
 m->meths=mt;
/*
 fprintf(stderr,"Method definition:  %s(",name);
 for (i=0;i<currentMethodParNum;i++) {
  if (i!=0) fprintf(stderr,",");
  fprintf(stderr,"%s",currentMethod.par[i]);
 }
 fprintf(stderr,")");
 if (strcmp(currentMethod.ret,"")) fprintf(stderr,":%s",currentMethod.ret);
 if (currentMethod.Class)
  fprintf(stderr," defined in class \"%s\"",currentMethod.Class->name);
 else
  fprintf(stderr," global");
 fprintf(stderr,Pub?", public\n":", private\n");
*/
}
//
int yyerror(const char *msg)
{
 (void) fprintf(stderr,"%s\n",msg);
 return (0);
}
//
int fail(const char *msg)
{
 fflush(stdout);
 fprintf(stderr,"\n%s\n",msg);
 exit (10);
}
//
```

```
ClassDef *findClass(char *name);
// si ferma con errore se la classe non esiste/ non è pubblica.
void checkLegalClass(char *name,short Public)
{
 ClassDef *c;
 short i;
  // cerca il tipo.
  c=findClass(name);
  if (c) {
   if (Public && !c->Public) {
    fprintf(stderr,"\"%s\" ",name);
    fail("public class expected.");
   }
  } else {
   i=0;
   while (strcmp("",libraryTypes[i])) {
    if (!strcmp(name,libraryTypes[i])) return;
    i++;
   }
   fprintf(stderr,"\"%s\" ",name);
   fail("type id unknown.");
  }
}

//-----------------------------------------------
//
// Load dell'elenco delle classi di libreria
//
void loadLibraryTypes()
{
 FILE *f;
 string buf;
 char *p,*p1;
 short i=0;

 if (!(f=fopen(libDefFile,"r"))) {
  fprintf(stderr,"\n Cannot open library.def!  \n\n");
  exit(10);
 }
 while (fgets(p=buf,StrLen,f)) {
  if (*p++=='T') {
   while (*p==' ') p++;
   p1=p;
   while ((*p!=' ')&&(*p!='T')&&(*p)) p++;
```

```
   *p='\0';
   if (i>=MaxLibraryTypes-2) {
    fprintf(stderr,"\n Cannot handle more than %d classes in standard library\n",
            MaxLibraryTypes-2);
    exit(10);
   }
   strcpy(libraryTypes[i++],p1);
  }
 }
 strcpy(libraryTypes[i],"");

 fclose(f);
}
//----------
// Funzione di supporto.
void intConv(char *s,long l)
{
 if (l>=0L)
  sprintf(s,"%ld",l);
 else {
  if (l==-2147483648)
   strcpy(s,"L(2147483648)");
  else
   sprintf(s,"L(%ld)",-l);
 }
}

//-----------------------------------------------
//
//  Dump (& check) delle strutture dati
//
//-----------------------------------------------
//
/*

 Ecco la mini-sintassi:

 Dichiarazione classe:

  BANG nameclasse BANG listasopraclassi AMPERSAND BANG  // -> pubblica, di libreria
  BANG nameclasse BANG listasopraclassi AMPERSAND COLON //-> privata, di libreria
  BANG nameclasse BANG listasopraclassi BANG        // -> pubblica
  BANG nameclasse BANG listasopraclassi COLON       //-> privata
```

```
  a cui segue zero o piu' volte le def.  var:
  nomevar COLON tipo SEMICOLON

  con tipo:
  id oppure tipo[ lower DOTDOT upper ]

  A queste seguono le def.  messaggi:

  CURLY_OPEN nomemessaggio COMMA arity BANG         // -> pubblico
  CURLY_OPEN nomemessaggio COMMA arity COLON        // -> privato

  a cui seguono zero o più volte le def.  metodi:

  listaParametri COLON belongingclass COLON returnclass CURLY_CLOSE

  (returnclass e belongingclass possono non essere presenti)
  e per finire

  AMPERSAND

*/
void dumpVarVector(VarDef *v)
{
 VarDef *vx=v;
 VarDef *w=v;
 short i;
 string buf1,buf2;

 while (v) {
  checkLegalClass(v->type->name,0);
  // controlliamo le doppie dichiarazioni.
  w=vx;
  while (w!=v) {
   if (!strcmp(v->name,w->name)) {
    fprintf(stderr,"\"%s\" ",v->name);
    fail("variable redefined.");
   }
   w=w->next;
  }
//
  printf("%s :  %s",v->name,v->type->name);
  for (i=0;i<v->type->dimNum;i++) {
   intConv(buf1,v->type->dim[i].start);
   intConv(buf2,v->type->dim[i].end);
```

```
   printf("[%s..%s]",buf1,buf2);
  }  printf(";\n");
//
  v=v->next;
 }
}
//
ClassDef *findClass(char *name)
{
 ClassDef *c=ClassDefList;

 while (c) {
  if (!strcmp(name,c->name)) return c;
  c=c->next;
 }
 return NULL;
}

void dumpOneClass(ClassDef *c)
{
 ClassDef *s;
 short i=0,found=0;

 if (c) {
  if (!c->scanned) {
   if (strcmp("",c->super)) {
    if (!(s=findClass(c->super))) {
     while (strcmp("",libraryTypes[i])) {
      if (!strcmp(c->super,libraryTypes[i])) {
    found=1;
    break;
      }
      i++;
     }
     if (!found) {
      fprintf(stderr,"\"%s\" ",c->super);
      fail("class not found");
     }
    }
    dumpOneClass(s);
   }
//
   printf("T %s T %s %c\n",c->name,c->super,c->Public?'T':':');
   dumpVarVector(c->vars);
```

```
//
   c->scanned=1;
  }
 }
}

void dumpClasses()
{
 ClassDef *c=ClassDefList;

 while (c) {
  c->scanned=0;
  c=c->next;
 }
 c=ClassDefList;
 while (c) {
  dumpOneClass(c);
  c=c->next;
 }
}
//
void dumpMethods()
{
// deve controllare che tutte le classi param esistano, e siano pubbliche
// se il metodo è pubblico.
 Message *m=MessageList;
 Method *mt;
 short i;

 while(m) {
  mt=m->meths;
  printf("{ %s , %d %c\n",m->name,m->arity,m->Public?'T':':');

  while (mt) {
   if (strcmp("",mt->ret) && strcmp("super",mt->ret))
    checkLegalClass(mt->ret,m->Public);
   for (i=0;i<m->arity;i++) checkLegalClass(mt->par[i],m->Public);
   //
   for (i=0;i<m->arity;i++) {
    if (i!=0) printf(",");
    printf("%s",mt->par[i]);
   }
   if (i!=0) printf(" ");
   printf(":");
```

```
   if (mt->Class) printf(" %s",mt->Class->name);
   printf(" :");
   if (strcmp(mt->ret,"")) printf(" %s",mt->ret);
   printf(" }\n");
   //
   mt=mt->next;
  }
  m=m->next;
 }
 printf("N\n");
}
//
void prepareMTCclasses();
void prepareMTCmethodsAndMessages();

//
void main(int ac,char *av[])
{
 extern int yyparse();
 char buf[StrLen];

 FILE *fi,*fo;

#ifdef __MWERKS__

// ac=ccommand(&av);
 SIOUXSettings.autocloseonquit=FALSE;
 SIOUXSettings.asktosaveonclose=FALSE;
 SIOUXSettings.showstatusline=FALSE;
 SIOUXSettings.columns=132;
 SIOUXSettings.rows=48;
 SIOUXSettings.toppixel=40;
 SIOUXSettings.leftpixel=5;
 setlocale(LC_ALL,"");

 if ((fo=freopen("::test.out3","w",stdout)) == NULL) {
  fprintf (stderr,"Can't redirect stdout\n");
  fflush(stdout);
  exit (1);
 }
 if ((fi=freopen("::test.out2","r",stdin)) == NULL) {
  fprintf (stderr,"Can't redirect stdin\n");
  fflush(stdout);
  exit (1);
```

```
 }
#endif

 if ((fo=freopen("../test.out3","w",stdout)) == NULL) {
  fprintf (stderr,"Can't redirect stdout\n");
  fflush(stdout);
  exit (1);
 }
 if ((fi=freopen("../test.out2","r",stdin)) == NULL) {
  fprintf (stderr,"Can't redirect stdin\n");
  fflush(stdout);
  exit (1);
 }

 loadLibraryTypes();

 if (yyparse()) {
  fprintf (stderr,"Sorry, there's an error, somewhere..\n");
  exit(10);
 } else {
  fflush(stdout);
  fprintf (stderr,"Scan completed.  Checking classes...\n\n");

  if ((fo=freopen(defFile,"w",stdout)) == NULL) {
   fprintf (stderr,"Can't redirect stdout\n");
   fflush(stdout);
   exit (1);
  }

  dumpClasses();
  dumpMethods();
  fprintf (stderr,"\nPhase 3 successfully completed.\n");
  fprintf (stderr,"Class hierarchy and method headers checked.\n");
//

 }
}
```

A.3.9. File: BOH/Code4/lex4.l

```
%{
//
// lex4.l - Antonio Cunei
//
#ifdef __MWERKS__
 #include "::includes:defs34.h"
#else
 #include "../includes/defs34.h"
#endif
#include "y.tab.h"
#define reflex(action)  { /* ECHO;*/ action }


void theIntNum(char*,quadVal*,intKind);
void fail(const char *);
char *theFloatingPoint(char *s);

string lastStr="";
string lastID="";

// NB: in questa fase vengono "riassorbiti" nei token i
// "meno", che fin qui sono stati trattati come operatori.

%}
D1          [0-9]+
D0          [0-9]*
IDL         [a-z0-9A-T]

BYTE                    {D1}"b"|"L("{D1}"b)"
UBYTE                   {D1}"ub"
WORD                    {D1}"w"|"L("{D1}"w)"
UWORD                   {D1}"uw"
LONG                    {D1}"l"?|"L("{D1}[l]?")"
ULONG                   {D1}"u""l"?
QUAD                    {D1}"q"|"L("{D1}"q)"
UQUAD                   {D1}"uq"

NUME                    ({D1}"e"|{D1}"e"{D1}|{D1}"eL("{D1}")")
NUMF            ({D1}"f"|{D1}"f"{D1}|{D1}"fL("{D1}")")

UDOUBLE         ({D1}"("{D1}")"|{NUME}"("{D1}")"|{NUME})
```

```
UFLOAT                  ({NUMF}"("{D1}")"|{NUMF})

DOUBLE                  ({UDOUBLE}|"L("{UDOUBLE}")")
FLOAT                   ({UFLOAT}|"L("{UFLOAT}")")

TTRUE                   "true"
TFALSE                  "false"

%s          StrState
%%
<INITIAL>"\""           strcpy(lastStr,yytext);BEGIN(StrState);
<StrState>"\"\""        strcat(lastStr,"\\\"");
<StrState>"\\"          strcat(lastStr,"\\\\");
<StrState>"\""          {strcat(lastStr,yytext);strcpy(yylval.str,lastStr);
                          BEGIN(INITIAL);return STRING;}
<StrState>.              |
<StrState>\n            strcat(lastStr,yytext);

<INITIAL>"uses"         reflex(return USES;)
<INITIAL>","            reflex(return COMMA;)
<INITIAL>":"            reflex(return COLON;)
<INITIAL>";"            reflex(return SEMICOLON;)
<INITIAL>"{"            reflex(return CURLY_OPEN;)
<INITIAL>"}"            reflex(return CURLY_CLOSE;)
<INITIAL>{BYTE}         reflex(theIntNum(yytext,&(yylval.qV),K_byte);return BYTE;)
<INITIAL>{UBYTE}        reflex(theIntNum(yytext,&(yylval.qV),K_ubyte);return UBYTE;)
<INITIAL>{WORD}         reflex(theIntNum(yytext,&(yylval.qV),K_word);return WORD;)
<INITIAL>{UWORD}        reflex(theIntNum(yytext,&(yylval.qV),K_uword);return UWORD;)
<INITIAL>{LONG}         reflex(theIntNum(yytext,&(yylval.qV),K_long);return LONG;)
<INITIAL>{ULONG}        reflex(theIntNum(yytext,&(yylval.qV),K_ulong);return ULONG;)
<INITIAL>{QUAD}         reflex(theIntNum(yytext,&(yylval.qV),K_quad);return QUAD;)
<INITIAL>{UQUAD}        reflex(theIntNum(yytext,&(yylval.qV),K_uquad);return UQUAD;)
<INITIAL>{TTRUE}        reflex(return TTRUE;)
<INITIAL>{TFALSE}       reflex(return TFALSE;)

<INITIAL>{FLOAT}        {reflex(strcpy(yylval.str,theFloatingPoint(yytext));
                           strcat(yylval.str,"f");return FLOAT;)}
<INITIAL>{DOUBLE}       {reflex(strcpy(yylval.str,theFloatingPoint(yytext));
                           return DOUBLE;)}

<INITIAL>"super"        reflex(return SUPER;)
<INITIAL>".."           reflex(return DOTDOT;)
<INITIAL>":J"           reflex(return ASSIGNMENT;)
<INITIAL>"if"           reflex(return IF;)
```

```
<INITIAL>"else"         reflex(return ELSE;)
<INITIAL>"elsif"        reflex(return ELSIF;)
<INITIAL>"while"        reflex(return WHILE;)
<INITIAL>"const"        reflex(return CONST;)
<INITIAL>"for"          reflex(return FOR;)
<INITIAL>"["            reflex(return SQUARE_OPEN;)
<INITIAL>"]"            reflex(return SQUARE_CLOSE;)
<INITIAL>"N"            reflex(return AMPERSAND;)
<INITIAL>"T"            reflex(return BANG;)
<INITIAL>"("            reflex(return PAR_OPEN;)
<INITIAL>")"            reflex(return PAR_CLOSE;)

<INITIAL>{IDL}+         {if (!strncmp(yytext,"___",3)) return ERROR;
                          else {strcpy(yylval.str,yytext);return ID;}}
<INITIAL>[ \t\r\v\f\n]  {}
<INITIAL>.               return ERROR;

%%
int yywrap(void)
{
 return 1;
}


void theIntNum(char* s,quadVal *qv,intKind k)
{
 unsigned long d=0,c=0,b=0,a=0;
 unsigned short neg=0;

 if (*s=='L') neg=1;

 while (*s && !(*s>='0' && *s <='9')) s++;
 while (*s && (*s>='0' && *s <='9')) {

  d*=10;
  d+=(*s++-'0');
  c*=10;
  if (d&0xffff0000) {
   c+=((d&0xffff0000)>>16);
   d&=0x0000ffff;
  }
  b*=10;
  if (c&0xffff0000) {
   b+=((c&0xffff0000)>>16);
```

```
  c&=0x0000ffff;
 }
 a*=10;
 if (b&0xffff0000) {
  a+=((b&0xffff0000)>>16);
  b&=0x0000ffff;
 }
 if (a&0xfffe0000)
  fail ("Numeric constant overflow");
}
if (neg) {
 d=~d; c=~c; b=~b; a=~a;
 d++;

 if (~(d|0x0000ffff)) {
  c++;
  d|=0xffff0000;
 }
 if (~(c|0x0000ffff)) {
  b++;
  c|=0xffff0000;
 }
 if (~(b|0x0000ffff)) {
  a++;
  b|=0xffff0000;
 }
 if (~(a|0x0000ffff)) {
  fail ("Negative numeric constant overflow");
 }
}
switch (k) {
 case K_ubyte:  if (a|b|c|(d>>8)) fail ("UBYTE constant overflow"); break;
 case K_uword:  if (a|b|c) fail ("UWORD constant overflow"); break;
 case K_ulong:  if (a|b) fail ("ULONG constant overflow"); break;
 case K_uquad:  if (a&0xffff0000) fail ("UQUAD constant overflow"); break;
 case K_byte:  if (!neg) {
        if (a|b|c|(d>>7)) fail ("Positive BYTE constant overflow");
      } else {
        if  (~(a&b&c&((d>>7)|0xfe000000))) fail ("Negative BYTE constant overflow");
      } break;
 case K_word:  if (!neg) {
        if (a|b|c|(d>>15)) fail ("Positive WORD constant overflow");
      } else {
        if  (~(a&b&c&((d>>15)|0xfffe0000))) fail ("Negative WORD constant overflow");
```

```
      } break;
  case K_long:  if (!neg) {
        if (a|b|(c>>15)) fail ("Positive LONG constant overflow");
      } else {
        if  (~(a&b&((c>>15)|0xfffe0000))) fail ("Negative LONG constant overflow");
      } break;
  case K_quad:  if (!neg) {
        if (a>>15) fail ("Positive QUAD constant overflow");
      } else {
        if  (~((a>>15)|0xfffe0000)) fail ("Negative QUAD constant overflow");
      } break;
  default :  ;
 }
 qv->high=a<<16|(b&0x0000ffff);
 qv->low=c<<16|(d&0x0000ffff);
}


string buf;

char *theFloatingPoint(char *s)
{
// s[] contiene qualcosa del C oppure L(C), dove C puo'
// essere del tipo B(A), A(A) oppure B
// con A ::= [0-9]+ e B ::= Ae|AeA|Ae-(A)
//
// la cosa va riconvertita in una stringa che abbia
// l'aspetto di un numero floating point abituale
//
char *p,*q;

strcpy(buf,"");
if (*s=='L') {
 strcat(buf,"-");
 s[strlen(s)-1]='\0';   // elimino ')'
 s+=2;  // salto "L("
}

// ora s[] e' B(A), A(A), oppure B;
// se e' A(A) non compare nessun "e" oppure "d"

p=s;
while (*p) {
 if ((*p=='e') || (*p=='f')) {
```

```
// gestione del caso B(A) oppure B
 *p='e'; // per cominciare.
 // dopo "e" posso avere:  niente, "L(" oppure numero:
 if (*(p+1)=='\0') {
  strcat(buf,s);
  strcat(buf,"0"); // in modo da comporre 999e0
  return buf;
 }
 p++;

 if (*p!='L') { // puo' esserci solo una
  // coppia di parentesi, per la parte intera
  // della mantissa
  while (*p)
  {
   if (*p=='(') {
    break;
   }
   p++;
  }
  if (*p) {
   *p++='\0';

   strncat(buf,p,strlen(p)-1);
   strcat(buf,".");
   strcat(buf,s);
  } else {
   strcat(buf,s);
  }
  if (buf[strlen(buf)-1]=='e')
   strcat(buf,"0");
  return(buf);
 }
 // c'e' sicuramente una coppia di parentesi
 // per l'esponente negativo; poi, forse,
 // la parte intera della mantissa.
 *p++='-';
 while (*p!=')') { // la trovo sicuramente, pattern del lex
  *p=*(p+1);
  p++;
 }
 *(p-1)='\0';
 *p++='\0';
 // ora:  s corrisponde a B; p ad (A) (se c'e')
```

```
  if (*p) {
   strncat(buf,p+1,strlen(p)-2); // salto le parentesi
   strcat(buf,".");
   strcat(buf,s);
  } else strcat(buf,s);
  return(buf);
 }

 p++;
}
// gestione del caso A(A)

 p=s;
 while (*p)
 {
  if (*p=='(') {
   *p='\0';
   break;
  }
  p++;
 }

 if (*p) {
  fail("Internal error in converion of floating point number");
 }
 p++;
 strncat(buf,p,strlen(p)-1);
 strcat(buf,".");
 strcat(buf,s);
 return(buf);
}
```

A.3.10. File: BOH/Code4/yacc4.y

```
%{
//
// yacc4.y - Antonio Cunei
//
#ifdef __MWERKS__
 #include "::includes:defs34.h"
#else
 #include "../includes/defs34.h"
#endif
//
extern char yytext[];
extern int yylex(void);

//--
// Per la parte di rilettura dei metodi
ClassDef *ClassDefList=NULL;
ClassDef *CurrentClass=NULL;

// e' anche il messaggio corrente durante la rilettura dei metodi.
Message *MessageList=NULL;
Message *thisMessage=NULL;
Method *currentMethod=NULL;
short currentMethodParNum;

//--
// Per la generazione del codice
string paramNames[maxParams];
string paramTypes[maxParams];
string methName;
string retType;
short numParams;
short numTypedParams;
short methRetValInited;

string thisClass;
string thisSuper;

void prepareMTCclasses();
void prepareMTCmethodsAndMessages();
ClassDef *findClass(char *name);
```

```
// per le Symbol Table
short uniq[200];

//--
//

void fail(const char *msg);

%}
%union
{
 string str;
 VarDef *VarDefPtr;
 Type *type;
 Dim dim;
 paramListType *plt;
 indexListType *intp;
 quadVal qV;
//
 exprType expp;
};
%token USES COMMA COLON SEMICOLON CURLY_OPEN CURLY_CLOSE ID
%token SUPER DOTDOT ASSIGNMENT IF ELSE ELSIF WHILE SQUARE_OPEN FOR
%token SQUARE_CLOSE AMPERSAND BANG PAR_OPEN PAR_CLOSE STRING ERROR CONST
%token BYTE UBYTE WORD UWORD LONG ULONG QUAD UQUAD TTRUE TFALSE FLOAT DOUBLE
%start package
%type <str> id ID spSuperClassList STRING index ifHead ifHeadPre
%type <str> whileHead whilePre forHead forPre FLOAT DOUBLE
%type <plt> exprList exprList1
%type <VarDefPtr> spVar
%type <type> nonVectType type
%type <dim> vectDim
%type <expp> expr expParam
%type <intp> indexList
%type <qV> BYTE UBYTE WORD UWORD LONG ULONG QUAD UQUAD
%%
//
// recupero delle strutture classi/variabili/messaggi/metodi
//
specialHeader   :  spClassMessageList AMPERSAND ;

spClassMessageList  :  | spClassMessageList spClass | spClassMessageList spMessage ;

spClass         :  BANG id BANG spSuperClassList AMPERSAND BANG // libreria !
```

```
        { CurrentClass=addClassDef($2,$4,1,1); } spVarDefList { CurrentClass=NULL; } ;
    spClass         :  BANG id BANG spSuperClassList AMPERSAND COLON // libreria !
        { CurrentClass=addClassDef($2,$4,0,1); } spVarDefList { CurrentClass=NULL; } ;
    spClass         :  BANG id BANG spSuperClassList BANG
        { CurrentClass=addClassDef($2,$4,1,0); } spVarDefList { CurrentClass=NULL; } ;
    spClass         :  BANG id BANG spSuperClassList COLON
        { CurrentClass=addClassDef($2,$4,0,0); } spVarDefList { CurrentClass=NULL; } ;
    spSuperClassList    :  {strcpy($$,"");} | id {strncpy($$,$1,StrLen);};

    spVarDefList    :  | spVarDefList spVarDef ;

    spVarDef        :  spVar COLON type SEMICOLON
                        {SetVarListType($1,$3); VarDefAppend(CurrentClass->vars,$1);} ;

    spVar           :  id {$$=SetupVar($1);} ;


    spMessage       :  CURLY_OPEN id COMMA LONG BANG
                       { addMessage($2,(long)($4.low),1); } spMethList ;
    spMessage       :  CURLY_OPEN id COMMA LONG COLON
                       { addMessage($2,(long)($4.low),0); } spMethList ;

    spMethList      :  | spMeth spMethList ;
    spMeth          :  {addMethod(); currentMethodParNum=0; } spParList COLON
                         spMethClass COLON spRetClass CURLY_CLOSE ;

    spMethClass     :  { currentMethod->Class=NULL; } ;
    spMethClass     :  id { currentMethod->Class=findClass($1); } ;

    spRetClass      :  {strcpy(currentMethod->ret,"");}
                    | id {strncpy(currentMethod->ret,$1,StrLen);}
                    | SUPER {strncpy(currentMethod->ret,"super",StrLen);} ;

    spParList       :  | spParList1 ;
    spParList1      :  spPar | spParList1 COMMA spPar ;
    spPar           :  id { strncpy(currentMethod->par[currentMethodParNum++],$1,StrLen);
} ;
    //
    package         :  specialHeader {
                        checkMessages();
                        prepareMTCclasses();
                        initSymbolTable();
                      } normalPackage { prepareMTCmethodsAndMessages(); };
```

```
    //
    // -----------
    //  Parsing del sorgente, ora siamo pronti per produrre del codice

    normalPackage   :  id COLON usesList
                       CURLY_OPEN { strcpy (thisClass,"");} packageBody CURLY_CLOSE ;
    id          :  ID ;

    usesList    :  | USES idList ;
    idList      :  | idList1 ;
    idList1     :  id | idList1 COMMA id ;

    packageBody :  | packageBody packageItem ;
    packageItem :  globalVar | classDef | methDef ;

    // manca ancora il caso CONST id ASSIGNMENT expr SEMICOLON
    globalVar   :  id ASSIGNMENT expr SEMICOLON {
                    STentry *st;
                    ClassDef *cd;

              printf("\t\t\t\t\t\t\t\t\t//  %s_%d:=%s  // %s\n",$1,uniq[0],$3.str,$3.type);
              printf("%s_%d=",$1,uniq[0]);
              printDispatchMTCmessage($3.str,$3.type,dispNormal);
              printf(";\n");

              if (findSTlastContextEntry($1)) {
               fprintf(stderr,"La variabile globale \"%s\" è definita due volte\n",$1);
               fail("");
              }
              st=addSTentry();
              if (!(cd=findClass($3.type))) {
               fprintf(stderr,"Il tipo \"%s\" non esiste??  Errore nella definizione\n",$1);
               fprintf(stderr," della variabile globale \"%s\"\n",$1);
               fail("");
              }
              st->type=cd;
              strcpy(st->name,$1);
              st->entryType=st_glob_var;
             };


    classDef    :  BANG id COLON superList CURLY_OPEN { strcpy (thisClass,$2);} classTail
;
    classDef    :  id COLON superList CURLY_OPEN { strcpy (thisClass,$1);} classTail ;
```

```
superList   :  | SUPER id {strcpy (thisSuper,$2);};

classTail   :  classVarDeclList CURLY_CLOSE
                { strcpy (thisClass,""); strcpy (thisSuper,"");}
            | classVarDeclList methDefList CURLY_CLOSE
                { strcpy (thisClass,""); strcpy (thisSuper,"");};
classVarDeclList    :  | classVarDeclList varDecl SEMICOLON ;
varDecl     :  varNameListOne COLON type ;
varNameListOne  :  id | varNameListOne COMMA id ;

//
type        :  nonVectType
            | type SQUARE_OPEN vectDim SQUARE_CLOSE {BumpDim($1,&$3); $$=$1;};
nonVectType :  id {$$=SetupType($1);};
vectDim     :  {fail("Flexible vectors not allowed..Sorry.");}
            | LONG {if ((long)($1.low)<1) fail("Dimension is less than 1.");
                $$.start=0; $$.end=$1.low-1;}
            | LONG DOTDOT LONG {if ((long)($3.low)<(long)($1.low))
                fail("lower bound greater than upper bound.");
                $$.start=(long)($1.low); $$.end=(long)($3.low);};



// NB!!  i record puntati da exprList e exprList1 vengono allocati qui e deallocati
// nelle produzioni che utilizzano exprList

exprList    :  {$$=new paramListType; $$->numParams=0;}| exprList1 {$$=$1;};
exprList1   :  {$$=new paramListType;$$->numParams=0;} expParam {
                     strcpy($$->paramNames[$$->numParams],$2.str);
                     strcpy($$->paramTypes[$$->numParams],$2.type);
                     $$->numParams++;
                     }
            | exprList1 COMMA expParam {
                     $$=$1;
                     strcpy($$->paramNames[$$->numParams],$3.str);
                     strcpy($$->paramTypes[$$->numParams],$3.type);
                     $$->numParams++;
                     };

expParam    :  expr
                  {
                   if ($1.simple) {
                    sprintf($$.str,"%s",$1.str);
```

```
                    } else {
                     string ss;
                     getTemp(ss);
                     printf("\t\t\t\t\t\t\t\t\t//  %s:=%s  // %s\n",ss,$1.str,$1.type);
                     printf ("_x_%s *%s=",$1.type,ss);
                     printDispatchMTCmessage($1.str,$1.type,dispNormal);
                     printf(";\n");
                     sprintf($$.str,"%s",ss);
                    }
                    strcpy($$.type,$1.type);
                   } ;

// NB!!  i record puntati da indexList vengono allocati qui e deallocati nelle produzioni
// che utilizzano indexList:

index   :  SQUARE_OPEN expr SQUARE_CLOSE
                  {
                    if (strcmp($2.type,"long")) {
                     fail("L'indice del vettore non è di tipo long");
                    }
                    if ($2.simple) {
                      strcpy($$,$2.str);
                    } else {
                     string ss;
                     getTemp(ss);
                     printf("\t\t\t\t\t\t\t\t\t//  %s:=%s  // %s\n",ss,$2.str,$2.type);
                     printf ("_x_%s *%s=",$2.type,ss);
                     printDispatchMTCmessage($2.str,$2.type,dispNormal);
                     printf(";\n");
                     strcpy($$,ss);
                    }
                   } ;

indexList   :  index {$$=new indexListType;
                      strcpy($$->indexes[0],$1);
                      $$->numIndexes=1;
                      }
            | indexList index {
                      $$=$1;
                      strcpy($$->indexes[$$->numIndexes],$2);
                      $$->numIndexes++;
                      };
```

```
expr        :      LONG {
                   string ss;
                   getTemp(ss);
                   printf("\t\t\t\t\t\t\t\t//  %s:=%ld  // long\n",ss,$1.low);
                   printf ("_x_long *%s=new_long(%ld);\n",ss,$1.low);
                   strcpy($$.str,ss);
                   strcpy($$.type,"long");
                   $$.simple=1;
                  }
               | ULONG {
                   string ss;
                   getTemp(ss);
                   printf("\t\t\t\t\t\t\t\t//  %s:=%lu // ulong\n",ss,$1.low);
                   printf ("_x_ulong *%s=new_ulong(%lu);\n",ss,(unsigned long)($1.low));
                   strcpy($$.str,ss);
                   strcpy($$.type,"ulong");
                   $$.simple=1;
                  }
               | QUAD {
                   string ss;
                   getTemp(ss);
                   printf("\t\t\t\t\t\t\t\t//  %s:=%ld:%lu  // long\n",
                     ss,(long)$1.low,(unsigned long)$1.high);
                   printf ("_x_quad *%s=new_quad(%ld,%lu);\n",
                     ss,(long)$1.high,(unsigned long)$1.low);
                   strcpy($$.str,ss);
                   strcpy($$.type,"quad");
                   $$.simple=1;
                  }
               | UQUAD {
                   string ss;
                   getTemp(ss);
                   printf("\t\t\t\t\t\t\t\t//  %s:=%lu:%lu // ulong\n",
                     ss,(unsigned long)$1.high,(unsigned long)$1.low);
                   printf ("_x_uquad *%s=new_uquad(%lu,%lu);\n",
                     ss,(unsigned long)$1.high,(unsigned long)$1.low);
                   strcpy($$.str,ss);
                   strcpy($$.type,"ulong");
                   $$.simple=1;
                  }
               | FLOAT {
                   string ss;
                   getTemp(ss);
                   printf("\t\t\t\t\t\t\t\t//  %s:=%s // float\n",ss,$1);
```

```
       printf ("_x_float *%s=new_float(%s);\n",ss,$1);
       strcpy($$.str,ss);
       strcpy($$.type,"float");
       $$.simple=1;
      }
| DOUBLE {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t\t//  %s:=%s // double\n",ss,$1);
       printf ("_x_double *%s=new_double(%s);\n",ss,$1);
       strcpy($$.str,ss);
       strcpy($$.type,"double");
       $$.simple=1;
      }
| UBYTE {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t\t//  %s:=%lu // ubyte\n",ss,$1.low);
       printf ("_x_ubyte *%s=new_ubyte(%lu);\n",ss,(unsigned long)($1.low));
       strcpy($$.str,ss);
       strcpy($$.type,"ubyte");
       $$.simple=1;
      }
| UWORD {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t\t//  %s:=%lu // uword\n",ss,$1.low);
       printf ("_x_uword *%s=new_uword(%lu);\n",ss,(unsigned long)($1.low));
       strcpy($$.str,ss);
       strcpy($$.type,"uword");
       $$.simple=1;
      }
| WORD {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t\t//  %s:=%ld // word\n",ss,$1.low);
       printf ("_x_word *%s=new_word(%ld);\n",ss,$1.low);
       strcpy($$.str,ss);
       strcpy($$.type,"word");
       $$.simple=1;
      }
| BYTE {
       string ss;
       getTemp(ss);
```

```
       printf("\t\t\t\t\t\t\t\t//  %s:=%ld // byte\n",ss,$1.low);
       printf ("_x_byte *%s=new_byte(%ld);\n",ss,$1.low);
       strcpy($$.str,ss);
       strcpy($$.type,"byte");
       $$.simple=1;
      }
| TTRUE {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t//  %s:=true // bool\n",ss);
       printf ("_x_bool *%s=bool_true();\n",ss);
       strcpy($$.str,ss);
       strcpy($$.type,"bool");
       $$.simple=1;
      }
| TFALSE {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t//  %s:=false // bool\n",ss);
       printf ("_x_bool *%s=bool_false();\n",ss);
       strcpy($$.str,ss);
       strcpy($$.type,"bool");
       $$.simple=1;
      }
| id {
    short level;

    STentry *st=findSTentry($1,&level);
    if (!st) {
     fprintf(stderr,"La variabile \"%s\" è sconosciuta\n",$1);
     fail("");
    }
    sprintf($$.str,"%s_%d",$1,level);
    $$.simple=1;
    strcpy($$.type,st->type->name);
   }
| STRING {
       string ss;
       char *p;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t//  %s:=%s  // text\n",ss,$1);
       printf ("_x_text *%s=new_text(%s);\n",ss,$1);
       strcpy($$.str,ss);
       strcpy($$.type,"text");
```

```
                  $$.simple=1;
                 } ;
expr        :  PAR_OPEN expr PAR_CLOSE { strcpy($$.str,$2.str);
                                         $$.simple=$2.simple;
                                         strcpy($$.type,$2.type);} ;
expr        :  id PAR_OPEN exprList PAR_CLOSE
    {
     short i;
     Method *meth;
     findStaticCallResult result;
     result=findStaticCall($1,$3,&meth);
     if (result==findStaticCallOK)
     {
      // se super:  trattamento particolare:  l'invocazione prevede un parametro
      // extra, che è quello che normalmente sarebbe il valore di ritorno!
      // la forma della traduzione di un metodo super non restituisce alcun
      // valore, ma accetta quello che dovrebbe essere il suo valore di ritorno
      // come parametro, nel suo primo argomento
      if (!strcmp(meth->ret,"super")) {
       string ss;
       getTemp(ss);
       printf("\t\t\t\t\t\t\t\t//  %s:=NewObject \"%s\"\n",ss,meth->Class->name);

       /**/ printf("_x_%s *%s=new _x_%s;\n",meth->Class->name,ss,meth->Class->name);
       /**/ printf("%s->mytype=_y_%s;\n",ss,meth->Class->name);

       strcpy($$.str,ss);
       $$.simple=1;
       strcpy($$.type,meth->Class->name);
       sprintf(ss,"%s(%s",$1,$$.str);
       for (i=0;i<$3->numParams;i++) {
        strcat(ss,",");
        strcat(ss,$3->paramNames[i]);
       }
       strcat(ss,")");
       printf("\t\t\t\t\t\t\t\t//  %s\n",ss);
       printDispatchMTCmessage(ss,"",dispSuper);
       printf(";\n");
      } else {
       $$.simple=0;
       strcpy($$.type,meth->ret);
       strcpy($$.str,$1);
       strcat($$.str,"(");
       for (i=0;i<$3->numParams;i++) {
```

```
        if (i) strcat($$.str,",");
        strcat($$.str,$3->paramNames[i]);
       }
       strcat($$.str,")");
      }
     }
     else
     {
      VarDef *field;
      if ($3->numParams!=1)
       findStaticCallHandleError($1,$3,result); // se non un parametro
      // abbiamo:  $3->paramNames[0] di tipo $3->paramTypes[0] (simple o meno)
      // di cui dobbiamo controllare se esiste il campo $1
      // controlliamo se siamo in un metodo della classe giusta.
      if (!strcmp("",thisClass)) findStaticCallHandleError($1,$3,result);
      if (strcmp($3->paramTypes[0],thisClass)) findStaticCallHandleError($1,$3,result);
      // potrebbe essere.  Vediamo se esiste il campo giusto.
      field=findField(thisClass,$1);
      if (!field) findStaticCallHandleError($1,$3,result);
      // vediamo ora se si tratta di un vettore.
      // Se si -> errore (OK nella prossima prod.)
      if (field->type->dimNum) {
       fprintf(stderr,"La componente \"%s\" della classe %s e' un vettore.\n",
                      field->name,thisClass);
       fail("");
      }
      // OK!!  Componiamo la stringa ed usciamo.
      sprintf($$.str,"%s->me._x_%s",$3->paramNames[0],$1);
      strcpy($$.type,field->type->name);
      $$.simple=0;
     }
     // le informazioni per la chiamata sono state estratte; il record può
     // essere rimosso
     delete $3;} ;

expr        :  id PAR_OPEN exprList PAR_CLOSE indexList
    {
     // id DEVE essere un selettore di campo, exprList un elemento solo.
     short i;
     Method *meth;
     findStaticCallResult result;
     result=findStaticCall($1,$3,&meth);
     if (result==findStaticCallOK)
     {
```

```
 fprintf(stderr,"L'invocazione sembra un accesso a una componente vettore,\n");
 fail("ma un metodo con nome uguale ne rende impossibile l'accesso.");
}
else
{
 VarDef *field;
 if ($3->numParams!=1)
  findStaticCallHandleError($1,$3,result); // se non un parametro
 // abbiamo:  $3->paramNames[0] di tipo $3->paramTypes[0] (simple o meno)
 // di cui dobbiamo controllare se esiste il campo $1
 // controlliamo se siamo in un metodo della classe giusta.
 if (!strcmp("",thisClass)) findStaticCallHandleError($1,$3,result);
 if (strcmp($3->paramTypes[0],thisClass))
  findStaticCallHandleError($1,$3,result);
 // potrebbe essere.  Vediamo se esiste il campo giusto.
 field=findField(thisClass,$1);
 if (!field) findStaticCallHandleError($1,$3,result);
 // vediamo ora se si tratta di un vettore.  Se no -> errore
 if (!field->type->dimNum) {
  fprintf(stderr,"La componente \"%s\" della classe %s non e' un vettore.\n",
                 field->name,thisClass);
  fail("");
 }
 // Verifichiamo che sia stato specificato il giusto numero di indici.
 if ($5->numIndexes!=field->type->dimNum) {
  fprintf(stderr,"Sono stati specificati %ld",(long)($5->numIndexes));
  fprintf(stderr,
   " indici, ma il campo \"%s\" \ndella classe %s ha dimensione %ld.\n",
   field->name,thisClass,(long)(field->type->dimNum));
  fail("");
 }
 // OK!!  Componiamo la stringa ed usciamo.
 sprintf($$.str,"%s->me._x_%s",$3->paramNames[0],$1);
 for (i=0;i<$5->numIndexes;i++) {
  strcat($$.str,"[");
  strcat($$.str,$5->indexes[i]);
  strcat($$.str,"->me.me");

  if (field->type->dim[i].start==0)
   strcat($$.str,"]");
  else {
   string ttemp;
   sprintf(ttemp,"- %ld]",field->type->dim[i].start);
   strcat($$.str,ttemp);
```

```
       }
      }
      $$.simple=0;
      strcpy($$.type,field->type->name);
     }
     // le informazioni per la chiamata sono state estratte; il record può
     // essere rimosso
     delete $3;
     delete $5;
    } ;

methDefList :  methDef | methDefList methDef ;

////
methDef         :  BANG id {strncpy(methName,$2,StrLen);
          methRetValInited=numParams=numTypedParams=0; openSTcontext();}
          methHeader CURLY_OPEN methTail;
methDef         :  id {strncpy(methName,$1,StrLen);
          methRetValInited=numParams=numTypedParams=0;
          openSTcontext();} methHeader CURLY_OPEN methTail;
// i parametri possono essere solo oggetti.
// il valore di ritorno viene creato *prima* dell'invocazione
paramDeclList   :  | paramDeclList1 ;
paramDeclList1  :  paramDecl | paramDeclList1 COMMA paramDecl ;
paramDecl       :  parNameList1 COLON id
               {
                STentry *st;
                ClassDef *cd;

                if (!(cd=findClass($3))) {
                 fprintf(stderr,
                  "Il tipo \"%s\" non esiste??  Errore nella definizione\n",$3);
                 fail("");
                }

                while (numTypedParams<numParams) {
                 if (findSTlastContextEntry(paramNames[numTypedParams])) {
                  fprintf(stderr,
         "Il parametro \"%s\" è definito due volte\n",paramNames[numTypedParams]);
                  fail("");
                 }
                 strcpy(paramTypes[numTypedParams],$3);
                 st=addSTentry();
                 st->type=cd;
```

```
                st->parNum=numTypedParams;
                strcpy(st->name,paramNames[numTypedParams++]);
                st->entryType=st_param;
               }
              };
parNameList1    :  id {strncpy(paramNames[numParams++],$1,StrLen);}
             | parNameList1 COMMA id {strncpy(paramNames[numParams++],$3,StrLen);};
methHeader      :  PAR_OPEN paramDeclList PAR_CLOSE methReturnVal;
methReturnVal   :  {strcpy(retType,""); beginMethodBody();}
                | COLON id {strncpy(retType,$2,StrLen);
                 STentry *st;
                 ClassDef *cd;

                 if (!(cd=findClass($2))) {
                  fprintf(stderr,
            "Il tipo \"%s\" non esiste??  Errore nella definizione\n",$2);
                  fail("");
                 }
                 st=addSTentry();
                 st->type=cd;
                 strcpy(st->name,methName);
                 st->entryType=st_methodname;
                 beginMethodBody();
                 printf("_x_%s *%s_%d;\n",$2,methName,uniq[1]);};
                | COLON SUPER {strcpy(retType,"super");
                 STentry *st;
                 ClassDef *cd;

                 if (!(cd=findClass(thisClass))) {
                  fprintf(stderr,
          "Il tipo \"%s\" non esiste??  Errore nella definizione\n",thisClass);
                  fail("");
                 }
                 st=addSTentry();
                 st->type=cd;
                 strcpy(st->name,methName);
                 st->entryType=st_methodname;
                 beginSuperMethodBody();
            } id PAR_OPEN exprList PAR_CLOSE {
                    short i;
                    Method *meth;
                    findStaticCallResult result;
                    result=findStaticCall($4,$6,&meth);
                    if (result==findStaticCallOK)
```

```
                    {
                     // DEVE essere super (e non ci sono ambiguita', sappiamo,
                     // per via delle regole imposte sulle combinazioni di
                     // metodi super/normali
                     if (!strcmp(meth->ret,"super")) {
                      string ss;
                      printf("\t\t\t\t\t\t\t\t\t//  Super call");
                      if (strcmp(meth->Class->name,thisSuper)) {
                       fprintf(stderr,
      "Il valore di ritorno di questa invocazione e' \"%s\",\n",meth->Class->name);
                       fprintf(stderr,
      "mentre la sopraclasse corrente e' \"%s\".\n",thisSuper);
                       fail("");
                      }
                      sprintf(ss,"%s(%s_%d",$4,methName,uniq[1]);
                      for (i=0;i<$6->numParams;i++) {
                       strcat(ss,",");
                       strcat(ss,$6->paramNames[i]);
                      }
                      strcat(ss,")");
                      printf("\t\t\t\t\t\t\t\t\//  %s\n",ss);
                      printDispatchMTCmessage(ss,"",dispSuper);
                      printf(";\n");
                     } else {
                        fprintf(stderr,
                  "Il metodo invocato non sembra essere un costruttore.\n");
                        fail("Verifica la correttezza dell'invocazione.");
                     }
                    }
                    else
                    {
                      findStaticCallHandleError($4,$6,result);
                    }
                  };

methVarDeclList :  | methVarDeclList methVarDecl SEMICOLON ;
methVarDecl     :  methVarNameListOne COLON type ;
methVarNameListOne  :  id | methVarNameListOne COMMA id ;

methTail        :  methVarDeclList cmdList CURLY_CLOSE
                   {closeSTcontext(); endMethodBody();}
                 | methVarDeclList CURLY_CLOSE
                   {closeSTcontext(); endMethodBody();};
```

```
optionalAmpersand :  | AMPERSAND
          { fprintf(stderr,"WARNING! ampersand operator unsupported.\n"); };

cmdList         :  cmd | cmdList cmd ;

// manca ancora il caso CONST id ASSIGNMENT expr SEMICOLON
cmd             :  id ASSIGNMENT expr SEMICOLON {
                STentry *st;
                ClassDef *cd;
                short lev;

                if (!(cd=findClass($3.type))) {
                 fprintf(stderr,
          "Il tipo \"%s\" non esiste??  Errore nella definizione\n",$3.type);
                 fprintf(stderr," della variabile \"%s\"\n",$1);
                 fail("");
                }
                if (st=findSTentry($1,&lev)) {
                 cfrCresult res=cfrC(cd,st->type);
                 if ((res==scorrelate)||(res==supercl)) {
                  fprintf(stderr,
                   "La variabile \"%s\" è già stata definita di tipo \"%s\"\n",
                   st->name,st->type->name);
                  fprintf(stderr,
                   " mentre questa espressione è di tipo \"%s\"\n",$3.type);
                  fprintf(stderr," che non è sua sottoclasse oppure uguale.\n");
                  fail("");
                 } else {
                  if (strcmp($1,methName)||strcmp("super",retType)) {
                    printf("\t\t\t\t\t\t\t\t//  %s_%d:=%s  // %s\n",
                    $1,lev,$3.str,$3.type);
                   printf("%s_%d=",$1,lev);
                   printDispatchMTCmessage($3.str,st->type->name,dispNormal);
                   printf(";\n");
                  } else {
                   fprintf(stderr,
          "Non e' possibile assegnare il valore di ritorno di un costruttore!\n");
                   fprintf(stderr,
          "Solo le sue componenti possono venire alterate.\n");
                   fail("");
                  }
                  if (!strcmp($1,methName)) {
```

```
                  methRetValInited=1;
                 }
                }
               } else {
                st=addSTentry();
                st->type=cd;
                strcpy(st->name,$1);
                st->entryType=st_loc_var;
                   printf("\t\t\t\t\t\t\t\t\t//  %s_%d:=%s  // %s\n",
                          $1,thisUniq,$3.str,$3.type);
                   printf("_x_%s *%s_%d=",$3.type,$1,thisUniq);
                   printDispatchMTCmessage($3.str,$3.type,dispNormal);
                   printf(";\n");
               }
              }

//
// assegnamento ad un campo.
//
               | id PAR_OPEN exprList PAR_CLOSE
                 ASSIGNMENT expr optionalAmpersand SEMICOLON {
                   short i;
                   if ($3->numParams!=1) {
     fail ("Sembra un accesso ad una componente, ma la sintassi e‘ errata.");
                   }
                   if (!strcmp("",$3->paramTypes[0])) {
                    fail ("L’espressione non ha alcun tipo di ritorno");
                   }
                   if (strcmp($3->paramTypes[0],thisClass)) {// se diversi
     fprintf(stderr,"Impossibile accedere alle componenti di un oggetto\n");
     fprintf(stderr,"  di tipo \"%s\" dentro questo metodo!\n",$3->paramTypes[0]);
                    fail ("");
                   }
// verificare il nome del campo fra i campi!!!
// NB: volendo fare le cose giuste, il chiamare l’accessore al campo od un metodo
// andrebbe scelto dinamicamente.  In via transitoria, cosi‘ puo‘ andare.
                   VarDef *field;
                   field=findField(thisClass,$1);
                   if (!field) {
          fprintf(stderr,"Non trovo una componente \"%s\" dentro un \n",$1);
          fprintf(stderr,"  oggetto di tipo \"%s\"!\n",$3->paramTypes[0]);
                    fail ("");
                   }
                   if (field->type->dimNum) {
```

```
fprintf(stderr,"La componente \"%s\" della classe %s e' un vettore.\n",
                          field->name,thisClass);
          fail("");
         }
         // ultimo controllo:  compatibilita' di tipi
         ClassDef *cd1,*cd2;
         if (!(cd1=findClass($6.type))) {
             fprintf(stderr,
"Il tipo \"%s\" non esiste??  Possibile errore interno.\n",$6.type);
                          fail("");
                         }
         if (!(cd2=findClass(field->type->name))) {
             fprintf(stderr,
"Il tipo \"%s\" non esiste??  Una componente ha tipo sconosciuto.\n",
                     field->type->name);
                          fail("");
                         }
         cfrCresult res=cfrC(cd1,cd2);
         if ((res==scorrelate)||(res==supercl)) {
fprintf(stderr,"La componente \"%s\" della classe %s e' di tipo %s,\n",
         field->name,thisClass,field->type->name);
fprintf(stderr," ma l'espressione e' di tipo %s, che non e'\n",
         $6.type);
fprintf(stderr," che non è sua sottoclasse oppure uguale.\n");
         fail("");
         }
         // OK!!  Componiamo la stringa ed usciamo.
         printf("\t\t\t\t\t\t\t\t\t//  %s->%s:=%s  // %s\n",
           $3->paramNames[0],$1,$6.str,$6.type);
         printf ("%s->me._x_%s=",$3->paramNames[0],$1);
         printDispatchMTCmessage($6.str,field->type->name,dispNormal);
         printf(";\n");
         }

     | id PAR_OPEN exprList PAR_CLOSE
       indexList ASSIGNMENT expr optionalAmpersand SEMICOLON {
         short i;
         if ($3->numParams!=1) {
 fail ("Sembra un accesso ad una componente, ma la sintassi e' errata.");
         }
         if (!strcmp("",$3->paramTypes[0])) {
          fail ("L'espressione non ha alcun tipo di ritorno");
         }
         if (strcmp($3->paramTypes[0],thisClass)) {// se diversi
```

```
        fprintf(stderr,"Impossibile accedere alle componenti di un oggetto\n");
        fprintf(stderr,"  di tipo \"%s\" dentro questo metodo!\n",$3->paramTypes[0]);
                    fail ("");
                   }
// verificare il nome del campo fra i campi!!!
// NB: volendo fare le cose giuste, il chiamare l'accessore al campo od un metodo
// andrebbe scelto dinamicamente.  In via transitoria, cosi' puo' andare.
                   VarDef *field;
                   field=findField(thisClass,$1);
                   if (!field) {
      fprintf(stderr,"Non trovo una componente \"%s\" dentro un \n",$1);
      fprintf(stderr,"  oggetto di tipo \"%s\"!\n",$3->paramTypes[0]);
                    fail ("");
                   }
                   if (!field->type->dimNum) {
      fprintf(stderr,"La componente \"%s\" della classe %s non e' un vettore.\n",
                      field->name,thisClass);
                    fail("");
                   }

// Verifichiamo che sia stato specificato il giusto numero di indici.
                   if ($5->numIndexes!=field->type->dimNum) {
      fprintf(stderr,"Sono stati specificati %ld indici, ",
                     (long)($5->numIndexes));
      fprintf(stderr,
              "ma il campo \"%s\" \ndella classe %s ha dimensione %ld.\n",
              field->name,thisClass,(long)(field->type->dimNum));
                    fail("");
                   }
                   // ultimo controllo:  compatibilita' di tipi
                   if (strcmp(field->type->name,$7.type)) {
      fprintf(stderr,"La componente \"%s\" della classe %s e' di tipo %s,\n",
                                     field->name,thisClass,field->type->name);
                       fprintf(stderr,"ma l'espressione e' di tipo %s.\n",
                                     $7.type);
                       fail("");
                   }
                   // OK!!  Componiamo la stringa ed usciamo.
                   printf("\t\t\t\t\t\t\t\t//  %s->%s",$3->paramNames[0],$1);
                   for (i=0;i<$5->numIndexes;i++) {
                    printf("[%s]",$5->indexes[i]);
                   }
                   printf(":=%s  // %s\n",$7.str,$7.type);
```

```
                printf("%s->me._x_%s",$3->paramNames[0],$1);
                for (i=0;i<$5->numIndexes;i++) {
                 if (field->type->dim[i].start==0)
                  printf("[%s->me.me]",$5->indexes[i]);
                 else
                  printf("[%s->me.me-(%ld)]",
                      $5->indexes[i],field->type->dim[i].start);
                }
                printf("=");
                printDispatchMTCmessage($7.str,field->type->name,dispNormal);
                printf(";\n");

                delete $5;
                };


           | expr optionalAmpersand SEMICOLON {
           if (strcmp($1.type,"")) {
fprintf(stderr,"L'espressione restituisce un tipo non nullo!!\n",$1.type);
            fail("");
           } else {
              printf("\t\t\t\t\t\t\t\t\t//  %s\n",$1.str);
              printDispatchMTCmessage($1.str,$1.type,dispNormal);
              printf(";\n");
           }
          };


           | expr AMPERSAND id SEMICOLON {
           if (strcmp($1.type,"")) {
fprintf(stderr,"L'espressione restituisce un tipo non nullo!!\n",$1.type);
            fail("");
           } else {
fprintf(stderr,"WARNING! ampersand operator unsupported.\n",$1.type);
              printf("\t\t\t\t\t\t\t\t\t//  %s\n",$1.str);
              printDispatchMTCmessage($1.str,$1.type,dispNormal);
              printf(";\n");
           }
          };

       | whilePre CURLY_OPEN {openSTcontext();} cmdList CURLY_CLOSE {
              closeSTcontext();
              printf("goto loop%s;\n",$1);
              printf("end%s:\n",$1);
```

```
                    printf("\t\t\t\t\t\t\t\t\t//  ENDWHILE\n");
}

                | CURLY_OPEN {openSTcontext();}
                  cmdList CURLY_CLOSE {closeSTcontext();}
                | ifHead cmdList CURLY_CLOSE
                  {
                   printf("\t\t\t\t\t\t\t\t\t//  ENDIF\n");
                   printf("false%s:\n",$1);
                   printf("end%s:\n",$1);
                   closeSTcontext();
                  }
                | ifHead cmdList CURLY_CLOSE ELSE CURLY_OPEN
                  {
                   printf("\t\t\t\t\t\t\t\t\t//  ELSE\n");
                   printf("goto end%s;\n",$1);
                   printf("false%s:\n",$1);
                   closeSTcontext();
                   openSTcontext();
                  }
                  cmdList CURLY_CLOSE
                  {
                   printf("\t\t\t\t\t\t\t\t\t//  ENDIF\n");
                   printf("end%s:\n",$1);
                   closeSTcontext();
                  }
                | forPre cmd CURLY_OPEN
                  {
                   openSTcontext();
                   printf("goto top%s;\n",$1);
                   printf("for%s:\n",$1);
                  }
                                cmdList CURLY_CLOSE
                  {
                   closeSTcontext();
                   printf("goto loop%s;\n",$1);
                   printf("\t\t\t\t\t\t\t\t\t//  ENDFOR\n");
                   printf("end%s:\n",$1);
                  } ;
forHead         :  FOR cmd {
                    string ss;
                    printf("\t\t\t\t\t\t\t\t\t//  FOR LOOP :\n");
                    getTemp(ss);
                    strcpy($$,ss);
```

```
                    printf("top%s:\n",ss);
                   } ;
forPre          :  forHead expParam SEMICOLON
                   {
                    if (strcmp($2.type,"bool")) {
                     fail("L'espressione restituisce un tipo non bool!!");
                    } else {
                     printf("\t\t\t\t\t\t\t\t\t//  FOR %s\n",$2.str);
                     printf ("if (!get_bool(%s)) goto end%s;\n",$2.str,$1);
                     printf("goto for%s;\n",$1);
                     printf("loop%s:\n",$1);
                     strcpy($$,$1);
                    }
                   } ;

ifHead          :  IF expParam CURLY_OPEN
                   {
                    if (strcmp($2.type,"bool")) {
                     fail("L'espressione restituisce un tipo non bool!!");
                    } else {
                     string ss;
                     printf("\t\t\t\t\t\t\t\t\t//  IF %s\n",$2.str);
                     getTemp(ss);
                     printf ("if (!get_bool(%s)) goto false%s;\n",$2.str,ss);
                     openSTcontext();
                     strcpy($$,ss);
                    }
                   }
              | ifHeadPre ELSIF expParam CURLY_OPEN
                   {
                    if (strcmp($3.type,"bool")) {
                     fail("L'espressione elsif restituisce un tipo non bool!!");
                    } else {
                     printf("\t\t\t\t\t\t\t\t\t//  ELSIF %s\n",$3.str);
                     printf ("if (!get_bool(%s)) goto false%s;\n",$3.str,$1);
                     openSTcontext();
                     strcpy($$,$1);
                    }
                   } ;
ifHeadPre       :  ifHead cmdList CURLY_CLOSE
                   {
                    string ss;
                    getTemp(ss);
                    strcpy($$,ss);
```

```
                    printf("end%s:\n",$1);
                    printf("goto end%s;\n",ss);
                    printf("false%s:\n",$1);
                    closeSTcontext();
                   };
whileHead       :  WHILE    {
                    string ss;
                    printf("\t\t\t\t\t\t\t\t\t//  WHILE LOOP :\n");
                    getTemp(ss);
                    strcpy($$,ss);
                    printf("loop%s:\n",$$);
                   };
whilePre        :  whileHead expParam {
                   if (strcmp($2.type,"bool")) {
                    fail("L'espressione restituisce un tipo non bool!!");
                   } else {
                    printf("\t\t\t\t\t\t\t\t\t//  WHILE %s\n",$2.str);
                    printf ("if (!get_bool(%s)) goto end%s;\n",$2.str,$1);
                    strcpy($$,$1);
                   }
                  };
%%
//-----------------------------------------------
//
//  Costruzione delle strutture dati
//
//-----------------------------------------------
//
ClassDef *addClassDef(char *name,char *super,short Pub,short Lib)
{
 ClassDef *p=ClassDefList;

 if (!p) p=ClassDefList=new ClassDef;
 else {
  while (p->next) p=p->next;
  p->next=new ClassDef;
  p=p->next;
 }
 strncpy(p->name,name,StrLen);
 strncpy(p->super,super,StrLen);
 p->Library=Lib;
 p->Public=Pub;
 p->next=NULL;
 p->vars=NULL;
```

```
 p->scanned=0;

 return p;
}
//
void VarDefAppend(VarDef *&start,VarDef *it)
{
 VarDef *p=start;

 if (!p) start=it; else {
  while (p->next) p=p->next;
  p->next=it;
 }
}
//
VarDef *SetupVar(char *it)
{
 VarDef *vd=new VarDef;
 strncpy (vd->name,it,StrLen);
 vd->type=NULL;
 vd->next=NULL;
 return vd;
}
//
Type *SetupType(char *name)
{
 Type *tp=new Type;
 strncpy (tp->name,name,StrLen);
 tp->dimNum=0;
 return tp;
}
//
void BumpDim(Type *tp,Dim *dim)
{
 if (tp->dimNum==maxDimNum-1) fail("Too many dimensions for this array.");
 tp->dim[tp->dimNum++]=*dim;
}
//
void SetVarListType(VarDef *vd,Type *tp)
{
 while (vd) {
  vd->type=tp;
  vd=vd->next;
 }
```

```
}
//

void addMessage(char *name,short arity,short Pub)
{
  Message *m;

  // check if already defined

  m=MessageList;
  while (m) {
   if ((m->arity==arity) && (!strncmp(m->name,name,StrLen))) {
    if (m->Public!=Pub) fail("Should be private instead of public or viceversa.");
      thisMessage=m;
      return;
   }
   m=m->next;
  }

  m=new Message;
  strncpy(m->name,name,StrLen);
  m->arity=arity;
  m->meths=NULL;
  m->next=MessageList;
  m->Public=Pub;
  MessageList=m;
  thisMessage=m;
}

void addMethod()
{
 currentMethod=new Method;
 currentMethod->next=thisMessage->meths;
 thisMessage->meths=currentMethod;
}
//
int yyerror(const char *msg)
{
 (void) fprintf(stderr,"%s\n",msg);
 return (0);
}
//
void fail(const char *msg)
{
```

```
 fflush(stdout);
 fprintf(stderr,"\n%s\n",msg);
 exit (10);
}
//
ClassDef *findClass(char *name)
{
 ClassDef *c=ClassDefList;

 while (c) {
  if (!strcmp(name,c->name)) return c;
  c=c->next;
 }
 return NULL;
}
//
// Qui finisce la parte di supporto per il caricamento di
// classi/def.var/messaggi/metodi
//
// ---------------------------------------------
//
// Qui inizia (si spera) la parte di supporto per la
// produzione del codice
//
//

//
// dump delle classi, preparazione del sorgente
// per il supporto run-time
//

void prepareOneMTCclass(ClassDef *c);
void buildMTCstruct(ClassDef *c);
void prepareMTCclasses()
{
 ClassDef *c;

 c=ClassDefList;
 while (c) {
  printf("MTCclass *_y_%s;\n",c->name);
  c=c->next;
 }
 printf("\n");
```

```
 printf("void setupMTCclasses() {\n");
 while (c) {
  c->scanned=0;
  c=c->next;
 }
 c=ClassDefList;
 while (c) {
  prepareOneMTCclass(c);
  c=c->next;
 }
 printf("}\n");

 printf("\n#ifdef __MWERKS__\n #include \"::includes:library.c\"\n");
 printf("#else\n #include \"includes/library.c\"\n#endif\n\n");

 c=ClassDefList;
 while (c) {
  buildMTCstruct(c);
  c=c->next;
 }
 printf("\n");
}

void prepareOneMTCclass(ClassDef *c)
{
 ClassDef *s;

 if (c) {
  if (!c->scanned) {
   if (strcmp("",c->super)) {
    if (!(s=findClass(c->super))) {
     printf("\"%s\" ",c->super);
     fail("class not found");
    }
    prepareOneMTCclass(s);
   }
//
 if (strcmp(c->super,"")) {
   printf(" _y_%s=",c->name);
   printf("new MTCclass(\"%s\",_y_%s);\n",
     c->name,c->super);
 } else {
   printf(" _y_%s=new MTCclass(\"%s\",NULL);\n",
     c->name,c->name);
```

```
 }
   c->scanned=1;
  }
 }
}

void buildMTCstruct(ClassDef *c)
{
 VarDef *v;
 short i;

 if (!c->Library) {
  v=c->vars;
  printf("typedef struct _data_%s {\n",c->name);
  if (strcmp(c->super,"")) {
   printf("  struct _data_%s super;\n",c->super);
  }
  while (v) {
   printf("  void *_x_%s",v->name);
   for (i=0;i<v->type->dimNum;i++)
    printf("[%ld]",v->type->dim[i].end - v->type->dim[i].start +1);
   printf(";\n");
   v=v->next;
  }
  printf("};\n");

  v=c->vars;
  printf("typedef struct _x_%s {\n",c->name);
  printf("  short refNum;\n");
  printf("  MTCclass *mytype;\n");
  printf("  struct _data_%s me;\n",c->name);
  printf("};\n");
 }
}

//
//-------------------------------------------
//
char signBuf[StrLen*(maxParams+1)];
char *setupSignature(Message *m,Method *mt)
{
   short i;

   strcpy(signBuf,"_m_");
```

```
   strcat(signBuf,m->name);
   for (i=0;i<m->arity;i++) {
    strcat(signBuf,"_");
    strcat(signBuf,mt->par[i]);
   }
   return signBuf;
}
// setupSignature() deve essere coerente con beginMethodBody(), che parte da
// dati diversi ma deve ottenere lo stesso risultato
//
// beginMethodBody parte da numParams, paramNames[], paramTypes[],
//                          methName e retType
// beginSuperMethodBody e' simile, ma dedicato ai metodi Super
//
void beginMethodBody()
{
 int i;

 if (strcmp(retType,"")) {
  printf ("//\n// return type:  %s\n//\nvoid *",retType);
 } else {
  printf ("//\n// no return value\n//\nvoid *");
 }

  printf ("_m_%s",methName);
  for (i=0;i<numParams;i++) {
   printf("_%s",paramTypes[i]);
  }
  printf (" (");
  for (i=0;i<numParams;i++) {
   if (!i) printf("\n           ");
   // i parametri formali hanno SEMPRE livello 1
   printf("_x_%s *%s_%d",paramTypes[i],paramNames[i],uniq[1]);
   if (i!=numParams-1) printf(",\n           ");
 }
  printf (")\n{\n// ##BEGIN##\n");
}
//
void beginSuperMethodBody()
{
 int i;

 printf ("//\n// Super method, type:  %s\n//\nvoid *",thisClass);
```

```
  printf ("_m_%s",methName);
  for (i=0;i<numParams;i++) {
   printf("_%s",paramTypes[i]);
  }
  printf (" (\n           _x_%s *%s_%d",thisClass,methName,uniq[1]);
  if (numParams) {
   printf(",\n           ");
   for (i=0;i<numParams;i++) {
    printf("_x_%s *%s_%d",paramTypes[i],paramNames[i],uniq[1]);
    if (i!=numParams-1) printf(",\n           ");
   }
  }
  printf (")\n{\n// ##BEGIN##\n");
}
//
void endMethodBody()
{
 if (strcmp(retType,"")&&strcmp(retType,"super"))
 {
   if (!methRetValInited) {
 fail("Attenzione!  Il valore di ritorno del metodo non e‘ stato inizializzato!");
   } else {
     printf("\t\t\t\t\t\t\t\t\t//  return value:\n");
     printf("return (void*)(%s_%d)",methName,uniq[1]);
   }
 } else {
  printf("\t\t\t\t\t\t\t\t\t//  nothing to return:\n");
  printf("return NULL");
 }
 printf (";\n}\n\n");
}
//
void prepareMTCmethodsAndMessages()
{
 Message *m=MessageList;
 Method *mt;
 short i;

 printf("void setupMTCmessages() {\n");
 printf("  MTCmessage *m;\n\n");
 while(m) {
 printf("  m=new MTCmessage(\"%s\",%d);\n",m->name,m->arity);
  mt=m->meths;
  while (mt) {
```

```
   //
   printf("  new MTCmethod(m,(void*)%s",setupSignature(m,mt));
   for (i=0;i<m->arity;i++) {
    printf(",_y_%s",mt->par[i]);
   }
   printf(");\n");
   mt=mt->next;
  }
  m=m->next;
 }
  printf("}\n");
}

// ---------------------------------------------
//
// compilazione vera e propria.
//
// i simboli non funzionali possono essere dei seguente tipo:
// - variabili locali
// - parametri
// - variabili globali (del package)
// - il valore di ritorno del metodo
//
// durante la definizione delle var.  locali (nel corpo di
// un metodo) possono essere aperti e chiusi nuovi contesti
//
//
// i simboli funzionali possono essere:
// - accessori di campo (restituiscono il ptr all'oggetto
//   identificato dal campo)
// - metodi




/*
  Funzioni di supporto:  confronto e controllo sulla mancanza di
   ambiguità fra i metodi dei messaggi.
*/

// confronto fra 2 classi.
//  1:  prima classe sopraclasse della seconda
//  0:  scorrelate
// -1:  seconda classe sopraclasse della prima
```

```
// -2:  uguali

typedef enum { uguali=-2,subcl,scorrelate,supercl } cfrCresult;

cfrCresult cfrC(ClassDef *prima,ClassDef *seconda)
{
 ClassDef *a;

 if (prima==seconda) return uguali;
 if (prima) {
  a=findClass(prima->super);
  while (a) {
    if (a==seconda) return subcl;
    a=findClass(a->super);
  }
 }
 if (seconda) {
  a=findClass(seconda->super);
  while (a) {
    if (a==prima) return supercl;
    a=findClass(a->super);
  }
 }
 return scorrelate;
}

long messageOK(Message *thism)
{
 string toSearchP[maxParams];
 Method *one,*two,*three;
 long ar,found,fault,q,toSearch,equal;
 cfrCresult res,vr;

 ar=thism->arity;
 if (!(one=thism->meths)) {
   fprintf(stderr,
    "Il messaggio \"%s\" di arità %ld non ha metodi\n",thism->name,ar);
   return 10;
 }
 if (!ar) {
  if (one->next) {
   fprintf(stderr,
    "Il messaggio \"%s\" di arità 0 ha più di un metodo\n",thism->name);
   return 10;
```

```
 } else return 0;
}



// Allora, come scritto da qualche altra parte nella documentazione, bisogna
// effettuare il seguente controllo:  per ogni messaggio, e per ogni arità,
// bisogna verificare i metodi a coppie:  se i due metodi sono ordinari, allora
// bisogna controllare che i metodi siano coerenti:
// - se ad una funzione fun(A,B):C affianco una fun(A1,B1):C1 ed A1 sottoclasse o
// uguale ad A, e B1 sottoclasse o uguale a B, ALLORA anche C1 deve essere sottoclasse
// oppure uguale a C. Come vantaggio addizionale, diventa possibile determinare in
// fase di compilazione il tipo "più generico" di ritorno di una funzione, noti i tipi
// "più generici" di una specifica invocazione del messaggio (ossia le dichiarazioni
// statiche dei parametri passati in un certo punto, che sono sempre determinabili).
// Se invece i due metodi sono uno super ed uno normale:
// - il metodo normale non deve per nessun motivo essere più specifico (od uguale)
// del metodo super.  Ossia:  se ho un super costr(A,B):C ed un metodo fun(A1,B1):C1,
// allora non devono essere contemporaneamente A1 sottoclasse od uguale di A e B1
// sottoclasse od uguale a B.
// Se infine i due metodi sono entrambi super:
// - se ho costr(A,B):C e costr(A1,B1):C1, con A1 sottoclasse od uguale di A e B1
// sottoclasse od uguale a B, ALLORA C1 deve essere uguale a C.
// Se le ultime due condizioni sono verificate, allora posso determinare staticamente
// il tipo più specifico (ed il più generico) costruito per una data combinazione
// di parametri da un metodo super (ossia da un costruttore).
// Bello!

one=thism->meths;

fault=0;

while (one) {
 two=one->next;
 while (two) {

  vr=uguali;
  for (q=0,toSearch=1;(q<ar) && toSearch; q++) {

   res=cfrC(findClass(one->par[q]),findClass(two->par[q]));
   switch (vr) {
   case supercl:  if ((res!=supercl)&&(res!=uguali)) toSearch=0;
                 break;
   case subcl:    if ((res!=subcl)&&(res!=uguali)) toSearch=0;
```

```
               break;
 case uguali:   if (res==scorrelate)
                toSearch=0;
               else
                vr=res;
               break;
 default:  toSearch=0;
          break;

 }
}

// in questo punto vr contiene (se toSearch==1) la relazione fra l'insieme
// dei parametri di one e quelli di two (supercl,subcl,uguali o scorrelate)


if (toSearch) {

 ClassDef *oneClass,*twoClass;


 oneClass=strcmp(one->ret,"super")?findClass(one->ret):one->Class;
 twoClass=strcmp(two->ret,"super")?findClass(two->ret):two->Class;

 if (strcmp(one->ret,"super")&&strcmp(two->ret,"super"))
 { // entrambe non super

  res=cfrC(oneClass,twoClass);
  switch (vr) {
  case supercl:
   if ((res==scorrelate)||(res==subcl)) {
    fprintf(stderr,"Il metodo %s(%s",thism->name,two->par[0]);
    for (q=1;q<ar;fprintf(stderr,",%s",two->par[q++]));
    fprintf(stderr,") restituisce un valore di ritorno di tipo %s;\n",
            twoClass->name);
    fprintf(stderr,"  ma questo è incoerente con la definizione:\n");
    fprintf(stderr,"  %s(%s",thism->name,one->par[0]);
    for (q=1;q<ar;fprintf(stderr,",%s",one->par[q++]));
    fprintf(stderr,
     ") che restituisce un valore di ritorno di tipo %s:\n",oneClass->name);
    fprintf(stderr,
     "  la classe %s non è sottoclasse od uguale a %s!\n",
     twoClass->name,oneClass->name);
    return 10;
```

```
      }
      break;
     case subcl:
      if ((res==scorrelate)||(res==supercl)) {
       fprintf(stderr,"Il metodo %s(%s",thism->name,one->par[0]);
       for (q=1;q<ar;fprintf(stderr,",%s",one->par[q++]));
        fprintf(stderr,") restituisce un valore di ritorno di tipo %s;\n",
                oneClass->name);
        fprintf(stderr,"  ma questo è incoerente con la definizione:\n");
        fprintf(stderr,"  %s(%s",thism->name,two->par[0]);
        for (q=1;q<ar;fprintf(stderr,",%s",two->par[q++]));
        fprintf(stderr,
         ") che restituisce un valore di ritorno di tipo %s:\n",twoClass->name);
        fprintf(stderr,
         "  la classe %s non è sottoclasse od uguale a %s!\n",
         oneClass->name,twoClass->name);
        return 10;
       }
       break;
     case uguali:
      fprintf(stderr,"La definizione %s(%s",thism->name,one->par[0]);
      for (q=1;q<ar;fprintf(stderr,",%s",one->par[q++]));
      fprintf(stderr,
   ") è duplicata!\n(errore inaspettato!  possibile problema nel compilatore)\n");
      return 10;
     default:
      fprintf(stderr,"Errore inaspettato nel compilatore.\n");
      return 10;

     }
    } else {
// uno super e uno no, oppure entrambi super

     if ((!strcmp(one->ret,"super"))&&(!strcmp(two->ret,"super")))
     { // entrambe super

      if (cfrC(oneClass,twoClass)!=uguali) {
        fprintf(stderr,"Il costruttore %s(%s",thism->name,two->par[0]);
        for (q=1;q<ar;fprintf(stderr,",%s",two->par[q++]));
        fprintf(stderr,") restituisce un valore di ritorno di tipo %s;\n",
                twoClass->name);
        fprintf(stderr,"  ma questo è incoerente con il costruttore:\n");
        fprintf(stderr,"  %s(%s",thism->name,one->par[0]);
        for (q=1;q<ar;fprintf(stderr,",%s",one->par[q++]));
```

```
        fprintf(stderr,
         ") che restituisce un valore di ritorno di tipo %s:\n",oneClass->name);
        fprintf(stderr,
         "  la classe %s non è uguale a %s!\n",twoClass->name,oneClass->name);
        fprintf(stderr,
         "Due costruttori i cui parametri siano sottoclassi dell'altro\n");
        fprintf(stderr,"devono restituire un valore di tipo uguale.\n");
        return 10;
       }
      } else {
// uno super e uno no.  Errore in ogni caso.
        fprintf(stderr,
         "Il %s %s(%s",strcmp(one->ret,"super")?"costruttore":"metodo",
         thism->name,two->par[0]);
        for (q=1;q<ar;fprintf(stderr,",%s",two->par[q++]));
        fprintf(stderr,") restituisce un valore di ritorno di tipo %s;\n",
                twoClass->name);
        fprintf(stderr,"  ma questo è incompatibile con il %s:\n",
                       strcmp(one->ret,"super")?"metodo":"costruttore");
        fprintf(stderr,"  %s(%s",thism->name,one->par[0]);
        for (q=1;q<ar;fprintf(stderr,",%s",one->par[q++]));
        fprintf(stderr,
         ") che restituisce un valore di ritorno di tipo %s:\n",oneClass->name);
        fprintf(stderr,
         "  Un metodo ed un costruttore non possono avere i parametri\n");
        fprintf(stderr,
         "  uno sottoclassi o uguali all'altro.\n");
        return 10;
       }
      }
     }
     two=two->next;
    }
    one=one->next;
   }


   // Il più è fatto; ora bisogna cercare (a coppie)
   // quelli che danno luogo a possibili ambiguità

   fault=0;
   one=thism->meths;
   while (one) {
    two=one->next;
```

```
 while (two) {

  for (q=0,toSearch=0,equal=1; q<ar; q++) {
   res=cfrC(findClass(one->par[q]),findClass(two->par[q]));
   if (res==scorrelate) { toSearch=0; equal=0; break; }
   if (res==uguali) strncpy(toSearchP[q],one->par[q],StrLen);
   else {
    equal=0;
    if (res==supercl) {
     strncpy(toSearchP[q],two->par[q],StrLen);
    } else {
     strncpy(toSearchP[q],one->par[q],StrLen);
    } if (vr==scorrelate) {
     vr=res;
    } else if (vr!=res) toSearch=1;
  }}
  if (equal) {
    fprintf(stderr,"Uhm..  definizione duplicata:  %s(%s",thism->name,toSearchP[0]);
    for (q=1;q<ar;fprintf(stderr,",%s",toSearchP[q++]));
    fprintf(stderr,")..\n");
    fault=1;
  }
  if (toSearch) { // possibile ambiguità!  cerchiamo i toSearchP:
   for (three=thism->meths; three ; three=three->next ) {
    for (found=1,q=0;q<ar;q++)
     if (cfrC(findClass(three->par[q]),findClass(toSearchP[q]))!=uguali) {
      found=0; break;
     }
    if (found) break;
   }
   if (!found) {
    fprintf(stderr,"Uhm..  definizione mancante:  %s(%s",thism->name,toSearchP[0]);
    for (q=1;q<ar;fprintf(stderr,",%s",toSearchP[q++]));
    fprintf(stderr,")..\n");
    fault=1;
  }}
  two=two->next;
 }
 one=one->next;
}

if (fault) return fault;
```

```
 return 0;
}
//

void checkMessages()
{
 long f=0;
 Message *m=MessageList;

 while (m) {
  if (messageOK(m)) break;
  m=m->next;
 }
 if (m) {
  fprintf(stderr,
   "Ci sono stati errori nella verifica dei messaggi.  Esco disgustato.\n");
  exit(10);
}}
//
void getTemp(char *s)
{
 static long counter=0;

 sprintf(s,"_tmp%ld",counter++);
}


//-------------------------------------------
typedef enum {findStaticCallOK,findStaticCallUnknown,
              findStaticCallNoValid,findStaticCallNoSuitable} findStaticCallResult;
//-------------------------------------------
//
// cerca il tipo statico corrispondente ad una invocazione (il minimo
// garantito per una determinata combinazione di tipi di parametri)
//
findStaticCallResult findStaticCall(char *msg,paramListType *params,Method **meth)
{
 // prima di tutto bisogna trovare la combinazione nomemessaggio/arità

 Message *m=MessageList;

 while (m && !( !strcmp(m->name,msg) && (m->arity==params->numParams) )) {
  m=m->next;
```

```
}
if (!m) {
 return findStaticCallUnknown;
}
// trovato il messaggio; ora cerchiamo il metodo migliore.
Method *best,*scan;
cfrCresult res;
short maybe,better,ar,q;

if (!(scan=m->meths)) {
 return findStaticCallNoValid;
}
ar=m->arity;
if (!ar) {
 *meth=m->meths;   // se non ci sono parametri
                   // torna il primo (e si spera unico) metodo.
 return findStaticCallOK;
}
best=NULL;
while (scan) {
 for (q=0,maybe=1;q<ar;q++) {
  res=cfrC(findClass(params->paramTypes[q]),findClass(scan->par[q]));
  if (res==scorrelate) { maybe=0; break; }
  if (res==supercl) { maybe=0; break; }
 }
 if (maybe) { // va bene.  Vediamo se è il migliore.
  if (best) {
   better=1;
   for (q=0;q<ar;q++) {
    if (cfrC(findClass(best->par[q]),findClass(scan->par[q]))==subcl)
     { better=0; break; }
  }}
  if ((!best) || better) {
   best=scan;
 }}
 scan=scan->next;
}

if (!best) {
   return findStaticCallNoSuitable;
}
*meth=best;
return findStaticCallOK;
}
```

```
//-------------------------------------------
// stampa il messaggio d'errore adatto, in accordo con il valore di ritorno
// di findStaticCall(), ed esce.
findStaticCallHandleError(char *msg,paramListType *params,
                          findStaticCallResult result)
{
 short q;

 switch (result) {
  case findStaticCallOK : break;
  case findStaticCallUnknown :
    fprintf(stderr,
     "Il messaggio \"%s\" di arità %ld è sconosciuto!\n",
     msg,(long)(params->numParams));
    fail("");
    break;
  case findStaticCallNoValid :
    fprintf(stderr,
     "Oh-oh!  non ci sono proprio metodi validi per il messaggio \"%s\"!..\n",msg);
    fail("");
    break;
  case findStaticCallNoSuitable :
    fprintf(stderr,
     "Non ci sono metodi adatti per l'invocazione %s(%s",
     msg,params->paramTypes[0] );
    for (q=1;q<params->numParams;fprintf(stderr,",%s",params->paramTypes[q++]));
    fprintf(stderr,")\n");
    fail("");
    break;
 }
}



//-------------------------------------------


VarDef *findField(char *Class,char *field) {
 ClassDef *c;
 VarDef *v;

 if (!(c=findClass(Class))) return NULL;
 v=c->vars;
```

```
 while (v)
 {
  if (!strcmp(field,v->name)) return v;
  v=v->next;
 }
 return NULL;
}

typedef enum {dispNormal,dispSuper} dispType;

//-----------------------------------------------
// stampa la parte di dispatching, se necessaria,
// corrispondente ad una invocazione.  E' realizzata in modo poco elegante:
// la procedura corretta sarebbe stata quella di estendere
// opportunamente il record descritivo dell'espressione per
// contemplare i possibili casi; per semplificare, sapendo
// che le espressioni in questo punto sono della forma
// A->B oppure A(A1,A2..An), faccio un parsing in miniatura
// e stampo i comandi C necessari.
void printDispatchMTCmessage(char *str,char *type,dispType dt)
{
 char *p,*p1;
 string s_copy,s_nam;
 short i=0;
 if (!(p=strchr(str,'('))) {
  printf ("(_x_%s*)(%s)",type,str);
 } else {
  if (!strcmp(type,"")) {
   printf("(void)(dispatch%sMTCmessage",dt==dispNormal?"":"Super");
  } else {
   printf("(_x_%s*)(dispatch%sMTCmessage",type,dt==dispNormal?"":"Super");
  }
  // cerco i parametri
  strcpy(s_copy,str);
  strcpy(s_nam,"");

  strtok(s_copy,"(,)"); // viene modificato s_copy
  while (p=strtok(NULL,"(,)")) {
   if (i++) strcat(s_nam,",");
   strcat(s_nam,p);
  }
  if (strcmp(s_nam,""))
   printf("(\"%s\",%s))",s_copy,s_nam);
  else
```

```
   printf("(\"%s\"))",s_copy);
 }
}

//-------------------------------------------

// solo le componenti possono essere vettori, dunque le variabili
// sfuse (che vengono mantenute nella tabella dei simboli) possono
// essere solo oggetti; come tipo viene quindi utilizzato solo il
// puntatore alla classe di appartenenza.
//

// trattandosi di una versione preliminare, utilizzo liste semplici;
// in realtà potrebbero essere adottate tecniche più sofisticate (hash etc.)

typedef enum {st_param,st_glob_var,st_loc_var,st_methodname} STcontent;

struct STentry {
  string name;
  ClassDef *type;
  STcontent entryType;
  short parNum; // da 1 in avanti
  STentry *next;
};

STentry *levelPointers[200];
STentry *topPointers[200];
// definito all'inizio del file.
// short uniq[200]; // serve in caso due definizioni con nome e livello uguale,
            // ma tipo diverso, siano fatte successivamente
short uniqCounter,thisUniq;
short STlevel;

STentry *topST;

//
// per ogni contesto i, gli elementi usati nel contesto
// partono da levelPointers[i] fino al link NULL
// i topPointers memorizzano l'ultima posizione utilizzata in ogni livello.
//
void initSymbolTable()
{
 STlevel=0; // contesto attuale == 0 , ossia globale
 levelPointers[0]=NULL;
```

```
 topPointers[0]=NULL;
 uniqCounter=0;
 uniq[0]=++uniqCounter;
 thisUniq=uniqCounter;
}

//
// NB: findSTlastContextEntry cerca solo nell'ULTIMO CONTESTO!
//
STentry *findSTlastContextEntry(char *s)
{
 STentry *st;

 st=levelPointers[STlevel];
 while (st) {
  if (!strncmp(st->name,s,StrLen)) return st;
  st=st->next;
 }
 return NULL;
}

//
// findSTentry cerca in tutti i contesti, dal più recente al più vecchio
//
STentry *findSTentry(char *s,short *level)
{
 STentry *st;

 short scanLevel=STlevel;

 while (scanLevel>0) {

  st=levelPointers[scanLevel--];
  while (st) {
   if (!strncmp(st->name,s,StrLen)) {
        *level=uniq[scanLevel+1];
        return st;
   }
   st=st->next;
  }
 }
 return NULL;
}
```

```
//
STentry *addSTentry()
{
 STentry *st=new STentry;

 if (!levelPointers[STlevel])
  levelPointers[STlevel]=st; // registra il primo
 if (topPointers[STlevel])
  topPointers[STlevel]->next=st;
 topPointers[STlevel]=st; // registra l'ultimo
 st->next=NULL;
 return st;
}
//
void openSTcontext()
{
 STlevel++;
 topPointers[STlevel]=levelPointers[STlevel]=NULL;
 uniq[STlevel]=++uniqCounter;
 thisUniq=uniqCounter;
}
//
void closeSTcontext()
{
 STentry *p1,*p=levelPointers[STlevel--];

 while (p)
 {
  p1=p->next;
  delete p;
  p=p1;
 }
 thisUniq=uniq[STlevel];
}
//

//
// ---------------------------------------------
//
void main(int ac,char *av[])
{
 extern int yyparse();
 char buf[StrLen];
```

```
#ifdef __MWERKS__
 FILE *fi,*fo;

// ac=ccommand(&av);
 SIOUXSettings.autocloseonquit=FALSE;
 SIOUXSettings.asktosaveonclose=FALSE;
 SIOUXSettings.showstatusline=FALSE;
 SIOUXSettings.columns=100;
 SIOUXSettings.rows=56;
 SIOUXSettings.toppixel=40;
 SIOUXSettings.leftpixel=5;
 setlocale(LC_ALL,"");

 if ((fo=freopen("::test.out4","w",stdout)) == NULL) {
  fprintf (stderr,"Can't redirect stdout\n");
  fflush(stdout);
  exit (1);
 }

 if ((fi=freopen("::test.outx","r",stdin)) == NULL) {
  fprintf (stderr,"Can't redirect stdin\n");
  fflush(stdout);
  exit (1);
 }
#endif

 if (yyparse()) {
  fprintf (stderr,"Sorry, there's an error, somewhere..\n");
  exit (10);
 } else {
  fflush(stdout);
  fprintf (stderr,"Scan completed.  Checking classes...\n\n");
  fprintf (stderr,"\nPhase 4 successfully completed.\n");
  fprintf (stderr,"Class hierarchy and method headers checked.\n");
  exit (0);
//


 }
}

//
```

A.3.11. File: BOH/Code5/runtime.cpp

```
//
// runtime.cpp - Antonio Cunei
//
// Supporto runtime per l'esecuzione di un programma BOH.
//

#include <locale.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __MWERKS__
 #include <sioux.h>
 #include <console.h>
#else
#endif
const short StrLen=512;
typedef char String[StrLen+1];

class tree
{
private:
 tree *tsuper;
 tree *tfirstson;
 tree *tbrother;
 String tname;
public:
 tree(char *pname,tree *&base,tree *tsuper);
 tree *super();
 char *name();
 tree *firstson();
 tree *brother();
};

inline tree *tree::super() { return tsuper; };
inline tree *tree::firstson() { return tfirstson; };
inline tree *tree::brother() { return tbrother; };
inline char *tree::name() { return tname; };

tree::tree(char *pname,tree *&base,tree *psuper)
{
```

```
 tree *p;

 tsuper=psuper;
 tfirstson=NULL;
 tbrother=NULL;
 strncpy(tname,pname,StrLen);
 if (!psuper) {
  if (base) {
   cout << " Errore aggiungendo il nodo all'albero!  La radice è già allocata\n";
   exit(10);
  }
  base=this;
 } else {
  if (!psuper->tfirstson) {
   psuper->tfirstson=this;
  } else {
   p=psuper->tfirstson;
   while (p->tbrother) p=p->tbrother;
   p->tbrother=this;
}}}

tree *getByNameScan(tree *r,char *s)
{
 tree *i,*res;

  if (!strcmp(r->name(),s)) return r;
  i=r->firstson();
  if (!i) return NULL;

  while (i) {
   if (res=getByNameScan(i,s)) return res;
   i=i->brother();
  }
  return NULL;
}

tree *getByName(tree *base,char *name)
{
 tree *i=getByNameScan(base,name);

 if (i) return(i);

 cout << "Non riesco a trovare la definizione del nome \"" << name << "\"\n";
 exit(10);
```

```
 return NULL;
};

//---
/*
  Funzioni di supporto:  confronto e controllo sulla mancanza di
   ambiguità fra i metodi dei messaggi.
*/

// confronto fra 2 classi.
//  1:  prima classe sopraclasse della seconda
//  0:  scorrelate
// -1:  seconda classe sopraclasse della prima
// -2:  uguali

typedef enum { uguali=-2,subcl,scorrelate,supercl } cfrCresult;

cfrCresult cfrC(tree *prima,tree *seconda)
{
 tree *a;

 if (prima==seconda) return uguali;
 a=prima->super();
 while (a) {
   if (a==seconda) return subcl;
   a=a->super();
 }
 a=seconda->super();
 while (a) {
   if (a==prima) return supercl;
   a=a->super();
 }
 return scorrelate;
}
//-----

class MTCclass :  public tree {
 String tname;
public:
 MTCclass(char *pname,MTCclass *psuper);
 MTCclass *super();
 MTCclass *firstson();
 MTCclass *brother();
};
```

```
inline MTCclass *MTCclass::super()
   { return (MTCclass *)(((tree *)this)->super()); };
inline MTCclass *MTCclass::firstson()
   { return (MTCclass *)(((tree *)this)->firstson()); };
inline MTCclass *MTCclass::brother()
   { return (MTCclass *)(((tree *)this)->brother()); };

static MTCclass *MTCclasses=NULL;
MTCclass *getMTCclassByName(char *name)
   { return (MTCclass *)getByName(MTCclasses,name); };

MTCclass::MTCclass(char *pname,MTCclass *psuper)
        :  tree (pname,(tree*&)MTCclasses,psuper) {};

//-----
class list {
private:
 list *tnext;
public:
 list(list *&base);
 list *next();
};

inline list *list::next() { return tnext; };

list::list(list *&base)
{
 list *p;

 tnext=NULL;
 p=base;
 if (!p) {
  base=this;
 } else {
  while (p->tnext) p=p->tnext;
  p->tnext=this;
}}

//-----
typedef struct startBlock {
  short refNum;
  MTCclass *mytype;
};
```

```
const unsigned long MAXPARAMS=10;
#define SuperParamsList(x) void *p0##x##,void *p1##x##,void *p2##x##, \
                     void *p3##x##,void *p4##x##,void *p5##x##,void *p6##x##, \
                     void *p7##x##,void *p8##x##,void *p9##x
#define SetParam(x) this->tparams[##x##]=(MTCclass*)p##x

typedef MTCclass *MTCClassParams[MAXPARAMS];

class MTCmessage;

class MTCmethod :  public list {
private:
 void *tcode;
 MTCClassParams tparams;
public:
 MTCmethod(MTCmessage *msg,void *pcode,SuperParamsList(=NULL));
 void *code();
 MTCmethod *next();
 MTCclass *params(unsigned long q);
};
inline void *MTCmethod::code() { return tcode; };
inline MTCmethod *MTCmethod::next()
   { return (MTCmethod *)(((list *)this)->next()); };
inline MTCclass *MTCmethod::params(unsigned long q) { return tparams[q]; };

//

class MTCmessage :  public list {
 friend MTCmethod::MTCmethod(MTCmessage *msg,void *pcode,SuperParamsList( ));
 unsigned long tarity;
 String tname;
 MTCmethod *tmethods;
public:
 MTCmessage (char *pname,unsigned long parity);
 MTCmethod *methods();
 long MTCmessageOK();
 char *name();
 MTCmessage *next();
 unsigned long arity();
};

inline MTCmethod *MTCmessage::methods() { return tmethods; };
inline unsigned long MTCmessage::arity() { return tarity; };
inline MTCmessage *MTCmessage::next()
```

```
   { return (MTCmessage *)(((list *)this)->next()); };
inline char *MTCmessage::name() { return tname; };

MTCmessage *MTCmessages=NULL;
MTCmessage::MTCmessage(char *pname,unsigned long parity)
          :  list ((list *&)MTCmessages) {
 tarity=parity;
 tmethods=NULL;
 strncpy(tname,pname,StrLen);
};

MTCmessage *getMTCmessageByName(char *name,long arity)
{
 MTCmessage *base=MTCmessages;
 while (base) {
  if ((!strcmp(base->name(),name))&&(base->arity()==arity)) return base;
  base=base->next();
 }
 cout << "Non riesco a trovare la definizione del nome \"" << name
      << "\" di arità " << arity << "\n";
 exit(10);
 return NULL;
}

//
MTCmethod::MTCmethod(MTCmessage *msg,void *pcode,SuperParamsList( ))
         :  list ((list *&)(msg->tmethods))
{
 SetParam(0);
 SetParam(1);
 SetParam(2);
 SetParam(3);
 SetParam(4);
 SetParam(5);
 SetParam(6);
 SetParam(7);
 SetParam(8);
 SetParam(9);

 unsigned long q,ar=msg->arity();

 for (q=0;q<ar;q++) {
  if (!(this->tparams[q])) {
   cout << "Un metodo ha meno parametri del corrispondente messaggio\n";
```

```
   exit (10);
 }}
 for (;q<MAXPARAMS;q++) {
  if (this->tparams[q]) {
   cout << "Un metodo ha più parametri del corrispondente messaggio\n";
   exit (10);
 }}
 tcode=pcode;
};

//-----

typedef enum {dispNormal,dispSuper} dispType;

 // cerco di inviare il messaggio m agli oggetti indicati.
 // quale sarà il metodo buono?  Non ho (ancora) problemi di efficienza,
 // quindi li passo tutti.

#define SetTPar(x) params[##x##]=(MTCclass*)(p##x##?((startBlock*)p##x##)->mytype:NULL)
void *dispatchGeneralMTCmessage(char *msg,dispType dt,
                                void *superParam,SuperParamsList( ));


void *dispatchMTCmessage(char *msg,SuperParamsList(=NULL))
{
 return dispatchGeneralMTCmessage(msg,dispNormal,NULL,
                                  p0,p1,p2,p3,p4,p5,p6,p7,p8,p9);
}

void *dispatchSuperMTCmessage(char *msg,void *superParam,SuperParamsList(=NULL))
{
 return dispatchGeneralMTCmessage(msg,dispSuper,superParam,
                                  p0,p1,p2,p3,p4,p5,p6,p7,p8,p9);
}

void *dispatchGeneralMTCmessage(char *msg,dispType dt,
                                void *superParam,SuperParamsList( ))
{
 MTCmessage *m;
 MTCClassParams params;

 SetTPar(0);
 SetTPar(1);
 SetTPar(2);
```

```
SetTPar(3);
SetTPar(4);
SetTPar(5);
SetTPar(6);
SetTPar(7);
SetTPar(8);
SetTPar(9);

long q,ar=0;

while ((ar<MAXPARAMS)&&(params[ar])) ar++;

q=ar;
while (q<MAXPARAMS) {
 if (params[q]) {
  cout << "Uno dei parametri di una invocazione di \"" << msg
       << "\" è nullo!  (forse un errore nel compilatore?)\n";
  exit (10);
 }
 q++;
}

m=getMTCmessageByName(msg,ar);

MTCmethod *best,*scan;
cfrCresult res;
long maybe,better;

if (!(scan=m->methods())) {
 cout << "Oh-oh!  non ci sono proprio metodi validi per il messaggio "
      << m->name() <<" !..\n";
 exit(10);
}
ar=m->arity();
if (!ar) { // torna il primo (si spera unico) metodo.
 if (dt==dispNormal) {
  return ((void*(*)())m->methods()->code())();
 } else {
  return ((void*(*)(void*))m->methods()->code())(superParam);
 }
}

best=NULL;
while (scan) {
```

```
 for (q=0,maybe=1;q<ar;q++) {
  res=cfrC(params[q],scan->params(q));
  if (res==scorrelate) { maybe=0; break; }
  if (res==supercl) { maybe=0; break; }
 }
 if (maybe) { // va bene.  Vediamo se è il migliore.
  if (best) {
   better=1;
   for (q=0;q<ar;q++) {
    if (cfrC(best->params(q),scan->params(q))==subcl) { better=0; break; }
  }}
  if ((!best) || better) {
   best=scan;
 }}
 scan=scan->next();
}

if (!best) {
 cout << "non ci sono metodi adatti per l'invocazione "
      << m->name() << "(" << params[0]->name() ;
 for (q=1;q<ar;cout << "," << params[q++]->name());
 cout << ")\n";
 exit (10);
}

if (dt==dispNormal) {
 switch (ar) {
  case  1 :  return ((void*(*)(void*))best->code())(p0);
  case  2 :  return ((void*(*)(void*,void*))best->code())(p0,p1);
  case  3 :  return ((void*(*)(void*,void*,void*))best->code())(p0,p1,p2);
  case  4 :  return ((void*(*)(void*,void*,void*,void*))best->code())(p0,p1,p2,p3);
  case  5 :  return ((void*(*)(void*,void*,void*,void*,void*))
                               best->code())(p0,p1,p2,p3,p4);
  case  6 :  return ((void*(*)(void*,void*,void*,void*,void*,void*))
                               best->code())(p0,p1,p2,p3,p4,p5);
  case  7 :  return ((void*(*)(void*,void*,void*,void*,void*,void*,void*))
                               best->code())(p0,p1,p2,p3,p4,p5,p6);
  case  8 :  return ((void*(*)(void*,void*,void*,void*,void*,void*,void*,void*))
                               best->code())(p0,p1,p2,p3,p4,p5,p6,p7);
  case  9 :  return ((void*(*)(void*,void*,void*,void*,void*,void*,void*,void*,void*))
                               best->code())(p0,p1,p2,p3,p4,p5,p6,p7,p8);
  case  10 :  return ((void*(*)(void*,void*,void*,void*,void*,
                               void*,void*,void*,void*,void*))
                               best->code())(p0,p1,p2,p3,p4,p5,p6,p7,p8,p9);
```

```
   default :  cout <<
     "Sembra che ci siano piu' di 10 parametri!!  Mi spiace, " <<
     "ma questa versione non riesce a gestirli.\n";
           exit(10);
  }
 } else {
  switch (ar) {
   case  1 :  return ((void*(*)(void*,void*))
                               best->code())(superParam,p0);
   case  2 :  return ((void*(*)(void*,void*,void*))
                               best->code())(superParam,p0,p1);
   case  3 :  return ((void*(*)(void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2);
   case  4 :  return ((void*(*)(void*,void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2,p3);
   case  5 :  return ((void*(*)(void*,void*,void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2,p3,p4);
   case  6 :  return ((void*(*)(void*,void*,void*,void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2,p3,p4,p5);
   case  7 :  return ((void*(*)(void*,void*,void*,void*,void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2,p3,p4,p5,p6);
   case  8 :  return ((void*(*)(void*,void*,void*,void*,void*,void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2,p3,p4,p5,p6,p7);
   case  9 :  return ((void*(*)(void*,void*,void*,void*,void*,
                              void*,void*,void*,void*,void*))
                               best->code())(superParam,p0,p1,p2,p3,p4,p5,p6,p7,p8);
   case  10 :  return ((void*(*)(void*,void*,void*,void*,void*,
                               void*,void*,void*,void*,void*,void*))
                                best->code())(superParam,p0,p1,p2,p3,p4,
                                                      p5,p6,p7,p8,p9);
   default :  cout <<
     "Sembra che ci siano piu' di 10 parametri!!  Mi spiace, " <<
     "ma questa versione non riesce a gestirli.\n";
           exit(10);
  }
 }
}

//
// ----------------------------------------
//

void setupMTCclasses();
void setupMTCmessages();
```

```
void main(int ac,char *av[])
{
#ifdef __MWERKS__
// ac=ccommand(&av);
 SIOUXSettings.autocloseonquit=FALSE;
 SIOUXSettings.asktosaveonclose=FALSE;
 SIOUXSettings.showstatusline=FALSE;
 SIOUXSettings.columns=80;
 SIOUXSettings.rows=25;
 SIOUXSettings.toppixel=40;
 SIOUXSettings.leftpixel=5;
 setlocale(LC_ALL,"");
#endif
 setupMTCclasses();
 setupMTCmessages();

 cout.precision(7);

 dispatchMTCmessage("main");
}

//-----

#ifdef __MWERKS__
 #include "::test.out4"
#else
 #include "../test.out4"
#endif
```

A.3.12. File: BOH/includes/library.c

```
//----------------------------------------
//
// Common macros
//
#define basicOp(Op,type,realOp) \
void *_m_##Op##_##type##_##type (_x_##type *a_1,_x_##type *b_1) { \
 _x_##type *thisObj=new _x_##type; \
 thisObj->mytype=_y_##type; \
 thisObj->me.me = ((a_1->me.me) realOp (b_1->me.me)); \
 return (void*)thisObj; \
}

#define basicCmp(Op,type,realOp) \
void *_m_##Op##_##type##_##type (_x_##type *a_1,_x_##type *b_1) { \
 _x_bool *thisObj=new _x_bool; \
 thisObj->mytype=_y_bool; \
 thisObj->me.me = ((a_1->me.me) realOp (b_1->me.me)); \
 return (void*)thisObj; \
}

#define basicPrim(type,truetype) \
_x_##type *new_##type (truetype it) { \
 _x_##type *thisObj=new _x_##type; \
 thisObj->mytype=_y_##type; \
 thisObj->me.me = it; \
 return thisObj; \
}
//----------------------------------------
//
// Definition:  class object
//
typedef struct _data_object {
};
typedef struct _x_object {
  short refNum;
  MTCclass *mytype;
  struct _data_object me;
};
//
// Constructor
//
```

```
void _m_object (_x_object *object_1)
{
}
//
// Methods
//
void *_m_J_object_object (_x_object *a_1,_x_object *b_1); // defined below
//
//
void _m_print_object (_x_object *a_1)
{
 cout << "<" << a_1->mytype->name() << ">";
}
void _m_println_object (_x_object *a_1) {
 _m_print_object (a_1);
 cout << "\n";
}
//---------------------------------------
//
// Definition:  class bool
//
typedef struct _data_bool {
  struct _data_object super;
  short me;
};
typedef struct _x_bool {
  short refNum;
  MTCclass *mytype;
  struct _data_bool me;
};
//
// Primitives
//
_x_bool *bool_true() {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 1;
 return thisObj;
}
_x_bool *bool_false() {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0;
 return thisObj;
```

```
}
short get_bool(_x_bool *a_1) {
 return a_1->me.me;
}
//
// Constructor
//
void *_m_not_bool (_x_bool *a_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = !(a_1->me.me);
 return thisObj;
}

//
// Methods
//
//   '~|\/?><+=_-*&^%$#@!
//   ABCDEFGHIJKLMNOPQRST

basicOp(and,bool,&&)
basicOp(or,bool,||)

basicCmp(J,bool,=)
basicCmp(HG,bool,!=)

void _m_print_bool (_x_bool *a_1) {
 if (a_1->me.me)
  cout << "true";
 else
  cout << "false";
}
void _m_println_bool (_x_bool *a_1) {
 _m_print_bool (a_1);
 cout << "\n";
}
//
// Part of _x_object
//
void *_m_J_object_object (_x_object *a_1,_x_object *b_1)
{
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = (a_1 == b_1);
```

```
 return thisObj;
}
//---------------------------------------
//
// Definition:  class num
//
typedef struct _data_num {
  struct _data_object super;
};
typedef struct _x_num {
  short refNum;
  MTCclass *mytype;
  struct _data_num me;
};
//
// Constructor
//
void _m_numzero (_x_num *numzero_1)
{
}
//---------------------------------------
//
// Definition:  generic numeric class
//
#define autoNumLib(BOHtype,Ctype)   \
typedef struct _data_##BOHtype {    \
  struct _data_num super;       \
  Ctype me;             \
};                  \
typedef struct _x_##BOHtype {       \
  short refNum;             \
  MTCclass *mytype;         \
  struct _data_##BOHtype me;        \
};                  \
basicPrim(BOHtype,Ctype)        \
                    \
basicOp(I,BOHtype,+)            \
basicOp(L,BOHtype,-)            \
basicOp(M,BOHtype,*)            \
basicOp(E,BOHtype,/)            \
                    \
void *_m_L_##BOHtype (_x_##BOHtype *a_1) {  \
 _x_##BOHtype *thisObj=new _x_##BOHtype;    \
 thisObj->mytype=_y_##BOHtype;          \
```

```
 thisObj->me.me = - (a_1->me.me);       \
 return (void*)thisObj;             \
}                   \
                    \
basicCmp(H,BOHtype,<)           \
basicCmp(G,BOHtype,>)           \
basicCmp(HJ,BOHtype,<=)         \
basicCmp(GJ,BOHtype,>=)         \
basicCmp(J,BOHtype,==)          \
basicCmp(HG,BOHtype,!=)         \
                    \
void _m_print_##BOHtype (_x_##BOHtype *a_1) { \
 cout << a_1->me.me;            \
}                   \
void _m_println_##BOHtype (_x_##BOHtype *a_1) { \
 _m_print_##BOHtype (a_1);      \
 cout << "\n";              \
}

autoNumLib(byte,signed char)
autoNumLib(ubyte,unsigned char)
autoNumLib(word,short)
autoNumLib(uword,unsigned short)
autoNumLib(long,long)
autoNumLib(ulong,unsigned long)
autoNumLib(float,float)
autoNumLib(double,double)
//---------------------------------------
//
// Definition:  class text
//
#define textLen 512

typedef struct _data_text {
  struct _data_num super;
  char me[textLen+1]; // solo come implementazione iniziale!!
};
typedef struct _x_text {
  short refNum;
  MTCclass *mytype;
  struct _data_text me;
};
//
// Primitive
```

```
//
_x_text *new_text(char *it) {
 _x_text *thisObj=new _x_text;
 thisObj->mytype=_y_text;
 strncpy(thisObj->me.me,it,textLen);
 return thisObj;
}
//
// Methods
//
//   '~|\/?><+=_-*&^%$#@!
//   ABCDEFGHIJKLMNOPQRST
_x_text *_m_I_text_text (_x_text *a_1,_x_text *b_1) {
 _x_text *thisObj=new_text(a_1->me.me);
 strcat(thisObj->me.me,b_1->me.me);
 return thisObj;
}

_x_text *_m_readtext () {
 char buf[textLen+1];
 fgets(buf,textLen-1,stdin);
 char *c=buf+(strlen(buf)-1);
 if (*c=='\n') *c='\0';
 _x_text *thisObj=new_text(buf);
 return thisObj;
}

void *_m_len_text (_x_text *a_1) {
 _x_long *thisObj=new_long((long)strlen(a_1->me.me));
 return (void*)thisObj;
}

void _m_print_text (_x_text *a_1) {
 cout << a_1->me.me;
}
void _m_println_text (_x_text *a_1) {
 _m_print_text (a_1);
 cout << "\n";
}

//---------------------------------------
//
// Stand alone
//
```

```
void _m_nl () {
 cout << "\n";
}

//-----------------------------------------------------
//--------------------------------------
//
// Definition:  class quad
//
typedef struct _data_quad {
  struct _data_num super;
  long high;
  unsigned long low;
};
typedef struct _x_quad {
  short refNum;
  MTCclass *mytype;
  struct _data_quad me;
};
//
// Primitive
//
_x_quad *new_quad (long hh,unsigned long ll) {
 _x_quad *thisObj=new _x_quad;
 thisObj->mytype=_y_quad;
 thisObj->me.low = ll;
 thisObj->me.high = hh;
 return thisObj;
}
//
// Methods
//
//   '~|\/?><+=_-*&^%$#@!
//   ABCDEFGHIJKLMNOPQRST

void *_m_I_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_quad *thisObj=new _x_quad;
 thisObj->mytype=_y_quad;
 thisObj->me.low = 0; // to do:  +
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_L_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_quad *thisObj=new _x_quad;
```

```
 thisObj->mytype=_y_quad;
 thisObj->me.low = 0; // to do:  -
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_M_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_quad *thisObj=new _x_quad;
 thisObj->mytype=_y_quad;
 thisObj->me.low = 0; // to do:  *
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_E_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_quad *thisObj=new _x_quad;
 thisObj->mytype=_y_quad;
 thisObj->me.low = 0; // to do:  /
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_L_quad (_x_quad *a_1) {
 _x_quad *thisObj=new _x_quad;
 thisObj->mytype=_y_quad;
 thisObj->me.low = 0; // to do:  -
 thisObj->me.high = 0;
 return (void*)thisObj;
}
//
//
void *_m_H_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  <
 return (void*)thisObj;
}
void *_m_G_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  >
 return (void*)thisObj;
}
void *_m_HJ_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  <=
```

```
 return (void*)thisObj;
}
void *_m_GJ_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  >=
 return (void*)thisObj;
}
void *_m_J_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  =
 return (void*)thisObj;
}
void *_m_HG_quad_quad (_x_quad *a_1,_x_quad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  !=
 return (void*)thisObj;
}

void _m_print_quad (_x_quad *a_1) {
 cout << "<quad support not yet completed>";
}
void _m_println_quad (_x_quad *a_1) {
 _m_print_quad (a_1);
 cout << "\n";
}
//----------------------------------------
//
// Definition:  class uquad
//
typedef struct _data_uquad {
  struct _data_num super;
  unsigned long high;
  unsigned long low;
};
typedef struct _x_uquad {
  short refNum;
  MTCclass *mytype;
  struct _data_uquad me;
};
//
// Primitive
```

```
//
_x_uquad *new_uquad (unsigned long hh,unsigned long ll) {
 _x_uquad *thisObj=new _x_uquad;
 thisObj->mytype=_y_uquad;
 thisObj->me.low = ll;
 thisObj->me.high = hh;
 return thisObj;
}
//
// Methods
//
//   '~|\/?><+=_-*&^%$#@!
//   ABCDEFGHIJKLMNOPQRST

void *_m_I_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_uquad *thisObj=new _x_uquad;
 thisObj->mytype=_y_uquad;
 thisObj->me.low = 0; // to do:  +
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_L_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_uquad *thisObj=new _x_uquad;
 thisObj->mytype=_y_uquad;
 thisObj->me.low = 0; // to do:  -
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_M_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_uquad *thisObj=new _x_uquad;
 thisObj->mytype=_y_uquad;
 thisObj->me.low = 0; // to do:  *
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_E_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_uquad *thisObj=new _x_uquad;
 thisObj->mytype=_y_uquad;
 thisObj->me.low = 0; // to do:  /
 thisObj->me.high = 0;
 return (void*)thisObj;
}
void *_m_L_uquad (_x_uquad *a_1) {
 _x_uquad *thisObj=new _x_uquad;
```

```
 thisObj->mytype=_y_uquad;
 thisObj->me.low = 0; // to do:  -
 thisObj->me.high = 0;
 return (void*)thisObj;
}
//
//
void *_m_H_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  <
 return (void*)thisObj;
}
void *_m_G_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  >
 return (void*)thisObj;
}
void *_m_HJ_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  <=
 return (void*)thisObj;
}
void *_m_GJ_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  >=
 return (void*)thisObj;
}
void *_m_J_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  =
 return (void*)thisObj;
}
void *_m_HG_uquad_uquad (_x_uquad *a_1,_x_uquad *b_1) {
 _x_bool *thisObj=new _x_bool;
 thisObj->mytype=_y_bool;
 thisObj->me.me = 0; // to do:  !=
 return (void*)thisObj;
}
```

```
void _m_print_uquad (_x_uquad *a_1) {
 cout << "<uquad>";
}
void _m_println_uquad (_x_uquad *a_1) {
 _m_print_uquad (a_1);
 cout << "\n";
}
```

A.3.13. File: BOH/includes/library.def

```
T object T  N T
T bool T object N T
T text T object N T
T num T object N T
T byte T num N T
T ubyte T num N T
T word T num N T
T uword T num N T
T long T num N T
T ulong T num N T
T quad T num N T
T uquad T num N T
T float T num N T
T double T num N T
{ object , 0 T
:  object :  super }
{ numzero , 0 T
:  num :  super }
{ I , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
text,text :  text :  text }
{ L , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
```

```
{ M , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
{ E , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
{ L , 1 T
ubyte :  ubyte :  ubyte }
byte :  byte :  byte }
uword :  uword :  uword }
word :  word :  word }
ulong :  ulong :  ulong }
long :  long :  long }
uquad :  uquad :  uquad }
quad :  quad :  quad }
float :  float :  float }
double :  double :  double }
{ H , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
```

```
{ G , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
{ HJ , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
{ GJ , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
{ J , 2 T
object,object :  object :  bool }
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
```

```
bool,bool :  bool :  bool }
double,double :  double :  bool }
{ HG , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
bool,bool :  bool :  bool }
{ and , 2 T
bool,bool :  bool :  bool }
{ or , 2 T
bool,bool :  bool :  bool }
{ not , 1 T
bool :  bool :  bool }
{ len , 1 T
text :  text :  long }
{ print , 1 T
object :  object :  }
ubyte :  ubyte :  }
byte :  byte :  }
uword :  uword :  }
word :  word :  }
ulong :  ulong :  }
long :  long :  }
uquad :  uquad :  }
quad :  quad :  }
float :  float :  }
double :  double :  }
bool :  bool :  }
text :  text :  }
{ println , 1 T
object :  object :  }
ubyte :  ubyte :  }
byte :  byte :  }
uword :  uword :  }
word :  word :  }
ulong :  ulong :  }
long :  long :  }
```

```
uquad :  uquad :  }
quad :  quad :  }
float :  float :  }
double :  double :  }
bool :  bool :  }
text :  text :  }
{ readtext , 0 T
:  text :  text }
{ nl , 0 T
:  :  }
```

A.3.14. File: BOH/includes/std-ops

```
300 2 l I
300 2 l L
500 2 l M
500 2 l E
750 1 r L
200 2 n J
200 2 n H
200 2 n G
200 2 n HJ
200 2 n GJ
200 2 n HG
150 2 n and
130 2 n or
```

A.3.15. File: BOH/includes/boh.h

```
const short StrLen=512;
typedef char string[StrLen+1];
#define DOTpriority 700
```

A.3.16. File: BOH/helper/mix.c

```
//
//  mix.c - Antonio Cunei
//
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef __MWERKS__
 #include <sioux.h>
 #include <console.h>
 #include "::includes:boh.h"
 #include <unix.h>
#else
 #include "../includes/boh.h"
#endif
#include <string.h>


main(int ac,char **av)
{
 FILE *help;
 string buf;

 if (!(help=fopen(av[1],"r"))) {
  fprintf(stderr,"\n Cannot open \"%s\"!  \n\n",av[1]);
  exit(10);
 }
 if (!fgets(buf,StrLen,help)) {
  fprintf(stderr,"\n Help file \"%s\" is empty!  \n\n",av[1]);
  exit(10);
 }
 while (fgets(buf,StrLen,stdin)) {
  if (!strcmp(buf,"// ##BEGIN##\n")) {
   while (fgets(buf,StrLen,help)) {
    if (!strcmp(buf,"// ##BEGIN##\n")) break;
    fputs(buf,stdout);
   }
  } else fputs(buf,stdout);
 }
 fclose(help);
}
```

A.3.17. File: BOH/boh

```
#!/bin/bash

check()
{
if [ $?  != 0 ]
then
 echo
 echo "Error during compilation."
 echo "Bailing out."
 echo
 cd $STARTDIR
 exit
fi
}

STARTDIR=‘pwd‘
BOHDIR=$(echo $0| sed -e "s/$(basename $0)\$//")
cd $BOHDIR
echo
echo " BOH - Implementazione Preliminare - Antonio Cunei 1998"
echo
echo " ------------------------------------------------------"
./prep
echo
if [ $# = 0 ]
then
 echo " Usage:    boh <filename>"
 echo
 cd $STARTDIR
 exit
fi

echo "Phase0:"
cd Code0
echo "  processing..."
./code0 <${STARTDIR}/${1} >../test.out0
check
cd ..

cd Code1
echo
```

```
echo "Phase1:"
echo "  processing..."
./code1 <../test.out0 >../test.out1
check
cd ..
rm test.out0

cd Code2
echo
echo "Phase2:"
echo "  building..."
flex -lI8 -Caem lex2.l
check
yacc -dr yacc2.y 2>/dev/null
check
g++ y.code.c y.tab.c lex.yy.c -o code2
check
echo "  processing..."
./code2 <../test.out1 >../test.out2
check
cd ..
rm test.out1

cd Code3
echo
echo "Phase3:"
echo "  processing..."
./code3 <../test.out2 >../test.out3
check
cd ..
rm test.out2

cat includes/library.def test.out3.def test.out3 >test.outx
check
rm test.out3.def
rm test.out3

cd Code4
echo
echo "Phase4:"
echo "  processing..."
./code4 <../test.outx >../test.out4
check
cd ..
```

```
rm test.outx

grep -v "^                                    //" <test.out4 | \
grep -v "^// ##BEGIN##" >result.source
grep "^                                  //\|^void \*_m_" <test.out4 | \
sed -e 's@^                                   //@@' \
    -e 's@//.*$@@' -e 's@^void _m_@function @' >result.symbolic
check

echo
echo "Source tuning (only needed for gcc):"
echo "  tweaking..."
mv test.out4 test.out4.1
grep '^_x_.* \*.*=\|// ##BEGIN##' test.out4.1 | \
     sed -e '/^_x_[0-9a-zA-T_]\+ \*.*=/ s/=.*;$/;/' >test.out4.2
check
sed -e '/^_x_.* \*.*=/ s/^_x_[0-9a-zA-T_]\+ \*//' test.out4.1 >test.out4.3
check
echo "  mixing..."
helper/mix test.out4.2 <test.out4.3 >test.out4
check
rm test.out4.1
rm test.out4.2
rm test.out4.3

echo
echo "Phase5:"
echo "  building..."
g++ Code5/runtime.cpp -o ${STARTDIR}/result
check
rm test.out4

echo
echo "Done!"
echo
echo "  Now run \"./result\"."
echo
cd $STARTDIR
```

A.3.18. File: BOH/prep

```
#!/bin/bash

check()
{
if [ $?  != 0 ]
then
 echo
 echo "Cannot initialize compiler!!"
 echo "Distribution kit corrupted."
 exit
else
 echo -n "."
fi
}

if [ -f helper/mix ]; then exit; fi

echo
echo -n " Initializing compiler..."
echo -n "4."
cd Code0
flex -lI8 -Caem lex0.l
check
g++ lex.yy.c -o code0
check
cd ..

cd Code1
echo -n "3"
flex -wlI8 -Caem lex1.l
check
yacc -dr yacc1.y
check
g++ y.code.c y.tab.c lex.yy.c -o code1
check
cd ..

cd Code3
echo -n "2"
flex -lI8 -Caem lex3.l
check
```

```
yacc -dr yacc3.y
check
g++ y.code.c y.tab.c lex.yy.c -o code3
check
cd ..

cd Code4
echo -n "1"
flex -lI8 -Caem lex4.l
check
yacc -dr yacc4.y
check
g++ y.code.c y.tab.c lex.yy.c -o code4
check
cd ..

echo -n "0"
g++ helper/mix.c -o helper/mix
check
echo
```

A.3.19. File: BOH/clean

```
#!/bin/bash

rm test.out0 2>/dev/null
rm test.out1 2>/dev/null
rm test.out2 2>/dev/null
rm test.out3 2>/dev/null
rm test.out3.def 2>/dev/null
rm test.outx 2>/dev/null
rm test.out4 2>/dev/null
rm test.out4.1 2>/dev/null
rm test.out4.2 2>/dev/null
rm test.out4.3 2>/dev/null
rm result.source 2>/dev/null
rm result.symbolic 2>/dev/null
rm result 2>/dev/null
rm helper/mix 2>/dev/null
rm ‘find .  -name core‘ 2>/dev/null

rm ‘ls -1 Code0/* | grep -v ’/lex0.l$’‘ 2>/dev/null
rm ‘ls -1 Code1/* | grep -v ’/lex1.l$’ | grep -v ’/yacc1.y$’‘ 2>/dev/null
rm Code2/* 2>/dev/null 2>/dev/null
rm ‘ls -1 Code3/* | grep -v ’/lex3.l$’ | grep -v ’/yacc3.y$’‘ 2>/dev/null
rm ‘ls -1 Code4/* | grep -v ’/lex4.l$’ | grep -v ’/yacc4.y$’‘ 2>/dev/null
```

A.3.20. File: BOH/dog

```
#!/bin/bash

if [ $# = 0 ]
then
 tr '[ABCDEFGHIJKLMNOPQRST]' '[‘~|\\/?><+=_\-*&^%$#@!]'
else
 tr '[ABCDEFGHIJKLMNOPQRST]' '[‘~|\\/?><+=_\-*&^%$#@!]'  <${1}
fi
```

A.3.21. File: BOH/autopack

```
#!/bin/bash

./clean
cd ..
tar cfvz ‘date ’+%m%d%H%M’‘.tgz BOH
cd BOH
```

A.3.22. File: BOH/de_iso_8859_1

```
#!/bin/bash
cd ..
mkdir converted
cp -r BOH converted
for ii in ‘find BOH -type f -print‘
do
echo "CONVERT <${ii} >converted/${ii}"
sed -e ’s/à/a‘/g’ -e ’s/è/e‘/g’ -e ’s/ì/i‘/g’ -e ’s/ò/o‘/g’ -e ’s/ù/u‘/g’ \
    -e ’s/á/a‘/g’ -e ’s/é/e‘/g’ -e ’s/í/i‘/g’ \
    -e ’s/ó/o‘/g’ -e ’s/ú/u‘/g’ <${ii} >converted/${ii}
done
```

A.4. Programmi di esempio in BOH

A.4.1. File: BOH/test/test.primo

```
first_example :  uses system
{
 !main()
 {
  println("Hello, world!");
 }
}
```

Output:

```
Hello, world!
```

```
File:  BOH/test/test.secondo
```

```
second_package:  uses system
{
//
// first class definition:  glass
//

 !glass :  super object
 {
  !glass():  super object()
  {
  }

  !break(w:glass)
  {
   println("Crash!");
  }
 }
//----------
//
// second class definition:  mattress
//

 !mattress :  super object
 {
  springs:long;
```

```
  !mattress(n:long):  super object()
  {
   mattress.springs:=n;
  }

  !break(m:mattress)
  {
   for a:=0; a<m.springs; a:=a+1;
   {
    println("Sproingg!!");
   }
  }
 }
 !main()
 {
  a:=glass();
  b:=mattress(3);
  break(a);
  break(b);
 }
}
```

Output:

```
Crash!
Sproingg!!
Sproingg!!
Sproingg!!
```

A.4.2. File: BOH/test/test.UNO

```
basic_system :  uses system {

/*
Un commento /* annidato?  */
// anche sulla linea va bene.
*/
// fuori dal blocco.

!  mytestKclass :  super object {

 !  zipp()
 {
  "zipp!".println;
 }

 !  main()
 {
  nl();
  print("  La lunghezza della stringa ""Machiavelli"" e':  ");
  println(len("Machiavelli"));
  print("  Mentre 4+4*(5+3)-7/2 fa ");
  println(4+4*(5+3)-7/2);
  println("  Non sono bravo?");
  println(2(4));
  /*
  println(5e3(4));
  println(51f6(4));
  println(51f-(6)(4));
  */
  nl();
  println("  Proviamo qualcosa di piu' difficile:");
  "  Cerchiamo di valutare (2*3+7.-(4)).*(4+3.*(5)) :  ".print;
  ((2*3+7.-(4)).*(4+3.*(5))).println;
  .nl;
  "-------------".println;
  .nl;
  "Prova di interazione!".println;
  .nl;
  " Batti qualche schifezza!  :  ".print;
  " Hai battuto :  """.+(.readText).+(""""").println;
  .nl;
```

```
"-------------".println;
"  Vediamo degli esempi di sintassi alternative:".println;
"    5.+(6).+(7).+(8).*(2).println  :  ".print;
     5.+(6).+(7).+(8).*(2).println;
"    println(*(+(+(+(5,6),7),8),2)) :  ".print;
     println(*(+(+(+(5,6),7),8),2));
"    println((5+6+7+8)*2)           :  ".print;
     println((5+6+7+8)*2);
.nl;
"-------------".println;
" Vediamo se riesco a stampare anche i caratteri speciali!  :  ""%\""".println;
.nl;
" Ed ora un po' di numeri assortiti :  ".print;
0.print; " ".print;
1.print; " ".print;
1.-.print; " ".print;
2000000000ul.print; " ".print;
4000000000ul.print; " ".print;
6000000000000000uq.print; " ".print;
2147483647.print; " ".print;
2147483648.-.println;

true.and(false).println;
false.and(false).println;
.nl;
true.or(false).println;
false.or(false).println;
.nl;

.zipp;
"Ecco fatto!".println;
.nl;

/*
{
 b:=34+8*4-2;
 c:=4;
 a:=b+c;
 a.println;
 .nl;
}
*/

if (4<5) {
```

```
  .nl;
  puf:=3;
  } elsif (6>3) {
  .nl;
  puf:=65;
  puf.println;
  } elsif (7<>9) {
  .nl;
  } else {
  .nl;
  }

  {
   c:=0;

   while c<10 {
    c.print;
    " ".print;
    c:=c+1;
   }
   .nl; .nl;
  }

  for j:=0; j<10; j:=j+1; {
   j.print;
   " :  ".print;
   for i:=0; i<10; i:=i+1; {
    i.print; " ".print;
   }
   .nl;
  }

  c:="La dimostrazione, ora, e‘ proprio terminata.";
  c.println;
 }
}
}
```

Output:

```
  La lunghezza della stringa "Machiavelli" e‘:  11
  Mentre 4+4*(5+3)-7/2 fa 33
  Non sono bravo?
```

```
4.2

  Proviamo qualcosa di piu‘ difficile:
  Cerchiamo di valutare (2*3+7.-(4)).*(4+3.*(5)) :  171

-------------

Prova di interazione!

 Batti qualche schifezza!  :  hgàè234983gl
 Hai battuto :  "hgàè234983gl"

-------------
  Vediamo degli esempi di sintassi alternative:
    5.+(6).+(7).+(8).*(2).println  :  52
    println(*(+(+(+(5,6),7),8),2)) :  52
    println((5+6+7+8)*2)           :  52

-------------
 Vediamo se riesco a stampare anche i caratteri speciali!  :  "%\"

 Ed ora un po’ di numeri assortiti :  0 1 -1 2000000000 4000000000 <uquad> 2147483647
-2147483648
false
false

true
false

zipp!
Ecco fatto!


0 1 2 3 4 5 6 7 8 9

0 :  0 1 2 3 4 5 6 7 8 9
1 :  0 1 2 3 4 5 6 7 8 9
2 :  0 1 2 3 4 5 6 7 8 9
3 :  0 1 2 3 4 5 6 7 8 9
4 :  0 1 2 3 4 5 6 7 8 9
5 :  0 1 2 3 4 5 6 7 8 9
6 :  0 1 2 3 4 5 6 7 8 9
7 :  0 1 2 3 4 5 6 7 8 9
8 :  0 1 2 3 4 5 6 7 8 9
```

```
9 :  0 1 2 3 4 5 6 7 8 9
La dimostrazione, ora, e‘ proprio terminata.
```

A.4.3. File: BOH/test/test.DUE

```
basic_system :  uses {

!  mytestKclass :  super object {
 !  main()
 {
println(2(4));
println(5e3(4));
println(51f6(4));
51f-(6)(4).println;

println(4.2);
println(4.5e3);
println(4.51f6);
println(4.51f-(6));

println(4.2);
println(4.5e3);
println(4.51f6);
println(4.51f-6);

println(18);
println(-18);
println(18w);
println(-18w);


999e.println;
999e99.println;
999.9.println;
999.9e.println;
999e-99.println;
999.9e99.println;
999.9e-99.println;
999.9e-(99).println;
9e-99(999).println;
9(999).println;
9e(999).println;
999e-(99).println;
9e99(999).println;
9e-(99)(999).println;
```

```
println(-999e);
println(-999e99);
println(-999.9);
println(-999.9e);
println(-999e-99);
println(-999.9e99);
println(-999.9e-99);
println(-999.9e-(99));
println(-9e-99(999));
println(-9(999));
println(-9e(999));
println(-999e-(99));
println(-9e99(999));
println(-9e-(99)(999));

-999e.println;
-999e99.println;
-999.9.println;
-999.9e.println;
-999e-99.println;
-999.9e99.println;
-999.9e-99.println;
-999.9e-(99).println;
-9e-99(999).println;
-9(999).println;
-9e(999).println;
-999e-(99).println;
-9e99(999).println;
-9e-(99)(999).println;

999e.-.println;
999e99.-.println;
999.9.-.println;
999.9e.-.println;
999e-99.-.println;
999.9e99.-.println;
999.9e-99.-.println;
999.9e-(99).-.println;
9e-99(999).-.println;
9(999).-.println;
9e(999).-.println;
999e-(99).-.println;
9e99(999).-.println;
9e-(99)(999).-.println;
```

```
12873e.println;
3e11.println;
971.1.println;
119.1878237683463768732e.println;
7623f-2.println;
567.4e12.println;
0.98765f-99.println;
712.9e-(18).println;
0e-001(91872133).println;
563(98675).println;
652e(1672518).println;
98675e-(3).println;
788e000013(2).println;
11112e-(5)(8761).println;

a:=2e;

for c:=0; c<10; c:=c+1; {
 a.println;
 a:=a*a;
}

  }
 }
}
```

Output:

```
4.2
4500
4510000
4.51e-06
4.2
4500
4510000
4.51e-06
4.2
4500
4510000
4.51e-06
18
-18
```

```
18
-18
999
9.99e+101
999.9
999.9
9.99e-97
9.999e+101
9.999e-97
9.999e-97
9.999e-97
999.9
999.9
9.99e-97
9.999e+101
9.999e-97
-999
-9.99e+101
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-9.999e-97
-9.999e-97
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-999
-9.99e+101
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-9.999e-97
-9.999e-97
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
```

```
-999
-9.99e+101
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-9.999e-97
-9.999e-97
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
12873
3e+11
971.1
119.1878
76.23
5.674e+14
0
7.129e-16
9187213
98675.56
1672519
98.675
2.788e+13
0.08761111
2
4
16
256
65536
4.294967e+09
1.844674e+19
3.402824e+38
1.157921e+77
1.340781e+154
```

A.4.4. File: BOH/test/test.complex

```
basic_system :  uses system {

operator x +j x  710;
operator x -j x  710;

//---------

!  complex :  super num
{
 re,im:double;

 !  -j(r,i:double):  super numZero()
 {
  -j.re := r;
  -j.im := -i;
 }

 !  +j(r,i:double):  super numZero()
 {
  +j.re:= r;
  +j.im:= i;
 }

 !  +(a,b:complex):  super numZero()
 {
  +.re:=a.re+b.re;
  +.im:=a.im+b.im;
 }

 !  -(a:complex):  complex {
  -:=(-(a.re))-j(a.im);
 }

 !  *(a,b:complex):  complex {
  *:=(a.re*b.re-a.im*b.im)+j(a.im*b.re+a.re*b.im);
 }

 !  complex(d:double):  complex {
  complex:=d+j0e;
 }
```

```
 !  /(a,b:complex):  complex {
  d:=b.re*b.re+b.im*b.im;
  if d=0 {
   "Division by zero".println;
   /:=0.0+j0.0;
  } else {
   /:=((a.re*b.re+a.im*b.im)/d)+j((a.im*b.re-a.re*b.im)/d);
  }
 }

 !  =(a,b:complex):  bool {
  if (a.re=b.re and a.im=b.im) {
   =:=true;
  } else {
   =:=false;
  }
 }

 !  println(c:complex)
 {
  c.re.print;
  if c.im<0.0 {
   "-j".print; -(c.im).println;
  } else {
   "+j".print;   c.im.println;
  }
 }

 !complexTest()
 {
  "La parte reale       di 5.0+j3.4 è :  ".print;  5.0+j3.4.re.println;
  "La parte immaginaria di 5.0+j3.4 è :  ".print;  5.0+j3.4.im.println;
 }
}

//---------

!  mytest_class :  super object
{
 !  main()
 {
  a:=4.0+j5.0;
  " a = ".print;
    a   .println; .nl;
```

```
  b:=3.91-j4.2;
  " b = ".print;
    b   .println; .nl;
  c:=a*b;
  " c = a*b = ".print;
    c         .println;
  .nl;
  "c+b/a + 1.0-j9.3 = ".print;
  (c+b/a + 1.0-j9.3)   .println;
  .nl;
  "(b/c).+(a) + 3.2-j4.3 = ".print;
  ((b/c).+(a) + 3.2-j4.3)  .println;
  .nl;
  "b.+(a/c) = ".print;
  (b.+(a/c))   .println;
  .nl;
  "b.+(a/c) + 1.0-j9.3 = ".print;
  (b.+(a/c) + 1.0-j9.3)   .println;
  .nl;
  "12.3+j4.7 * 3.2-j8.25 = ".print;
  (12.3+j4.7 * 3.2-j8.25)   .println;

  .nl;
  .complexTest;
  .nl;
 }
}


}
```

Output:

```
 a = 4+j5

 b = 3.91-j4.2

 c = a*b = 36.64+j2.75

c+b/a + 1.0-j9.3 = 37.50927-j7.436585

(b/c).+(a) + 3.2-j4.3 = 7.297561+j0.5780488
```

```
b.+(a/c) = 4.028744-j4.072449

b.+(a/c) + 1.0-j9.3 = 5.028744-j13.37245

12.3+j4.7 * 3.2-j8.25 = 78.135-j86.435

La parte reale       di 5.0+j3.4 è :  5
La parte immaginaria di 5.0+j3.4 è :  3.4
```

A.4.5. File: BOH/test/test.TRE

```
basic_system :  uses {

operator x +j x 710;
operator x -j x 710;

//---------

!  complex :  super num
{
 re,im:double;

 !  +j(r,i:double):  super numZero()
 {
  +j.re:= r;
  +j.im:= i;
 }

 !  println(c:complex)
 {
  c.re.print;
  if c.im<0.0 {
   "-j".print;   c.im.-.println;
  } else {
   "+j".print;   c.im.println;
  }
 }
}

//---------

!  list :  super object
{
 it:object;
 next:list;

 !  add(l:list,x:object):  super object()
 {
  add.next:=l;
  add.it:=x;
 }
```

```
 !  emptyList():  super object()
 {
  emptyList.next:=emptyList;
  emptyList.it:=object();
 }

 !  scanPrint(l:list)
 {
  while not(l=l.next) {
   println(l.it);
   l:=l.next;
  }
 }
}


//---------
// solo a fini di test:  sottoclasse di complex

!  plaf :  super complex
{
 vec:  long[-8..-3];

 !  printme(x:plaf)
 {
  for a:=-8; a<-3; a:=a+1; {
   x.vec[a].println;
  }
 }

 !  fillme() :  super 2.4+j5.1
 {
  for a:=-8; a<-3; a:=a+1; {
   fillme.vec[a]:=(10-a)*8;
  }
 }
}

//---------

!  main()
{
 a:=2e;
```

```
 for c:=0; c<10; {a:=a*a;c:=c+1;} {
  a.println;
 }

 .nl;

 5.0+j3.4.println;

 .nl;
//

 "Ora vediamo una lista di elementi assortiti:".println;
 .nl;
 .emptyList.add(4).add(1.2+j4.7).add("ciao").add(912.45*7e91).scanPrint;
 .nl;
//

 "Ed infine un bel vettore, (sottoclasse di complex!)  riempito e subito stampato:"
  .println;
 .nl;
 fillme().printme;
 fillme().println;
}
}
```

Output:

```
2
4
16
256
65536
4.294967e+09
1.844674e+19
3.402824e+38
1.157921e+77
1.340781e+154

5+j3.4

Ora vediamo una lista di elementi assortiti:

6.38715e+94
```

```
ciao
1.2+j4.7
4

Ed infine un bel vettore, (sottoclasse di complex!)  riempito e subito stampato:

144
136
128
120
112
2.4+j5.1
```

A.4.6. File: BOH/test/test.op_test

```
op_test:  uses system
{
operator x lessico x 150;
operator a+ww x 150;
operator x *rrww 55;
operator x +j x 800;
operator x -j x 800;

col:=picchio(gino,camomilla);
pos:=lessicolessicolessico;
tok:=4+5*3--6/2.5e-6+-cino*rrwwlessicologo*rrww;
sup:=a+wwa+ww;
uu:=*rrww*rrww;
tik:=a+ww*rrww;
cip:=3+j4+5.2-j7.8e3;
+j:=+j+j+j;
}
```

Dopo la fase 2 (con lo script “dog”):

```
op_test :  uses system {

col:=picchio(gino,camomilla);
pos:=lessico(lessico,lessico);
tok:=*rrww(lessico(*rrww(+(-(+(4,*(5,3)),/(-(6),5e-(6)(2))),-(cino))),logo));
sup:=a+ww(a+ww);
uu:=*rrww(*rrww);
tik:=a+ww(*rrww);
cip:=+(+j(3,4),-j(2(5),8e3(7)));
+j:=+j(+j,+j);
}
```

A.5. Un esempio di compilazione

A.5.1. File di partenza: BOH/test/test.comp

```
basic_system :  uses {

//---------

!  list :  super object
{
 it:object;
 next:list;

 !  add(l:list,x:object):  super object()
 {
  add.next:=l;
  add.it:=x;
 }

 !  emptyList():  super object()
 {
  emptyList.next:=emptyList;
  emptyList.it:=object();
 }

 !  scanPrint(l:list)
 {
  while not(l=l.next) {
   println(l.it);
   l:=l.next;
  }
 }
}

//---------

!  main()
{

 for c:=2; c<8; c:=c+1; {
  if c<6 {c.println;} else {println("troppo!");}
 }
 .nl;
```

```
 "Ora vediamo una lista di elementi assortiti:".println;
 .nl;
 .emptyList.add(4).add(4.7).add("ciao").add(912.45*7e91).scanPrint;
 .nl;
}
}
```

A.5.2. Dopo la fase zero: test.out0

```
basicKsystem :  uses {

EELLLLLLLLL

T list :  super object
{
 it:object;
 next:list;

 T add(l:list,x:object):  super object()
 {
  add.next:Jl;
  add.it:Jx;
 }

 T emptylist():  super object()
 {
  emptylist.next:Jemptylist;
  emptylist.it:Jobject();
 }

 T scanprint(l:list)
 {
  while not(lJl.next) {
   println(l.it);
   l:Jl.next;
  }
 }
}

EELLLLLLLLL

T main()
{

 for c:J2; cH8; c:JcI1; {
  if cH6 {c.println;} else {println("troppo!");}
 }
 .nl;

 "Ora vediamo una lista di elementi assortiti:".println;
```

```
 .nl;
 .emptylist.add(4).add(4.7).add("ciao").add(912.45M7e91).scanprint;
 .nl;
}
}
```

A.5.3. Dopo la fase uno: test.out1

```
basicKsystem :  uses {



T list :  super object
{
 it:object;
 next:list;

 T add(l:list,x:object):  super object()
 {
  add.next:Jl;
  add.it:Jx;
 }

 T emptylist():  super object()
 {
  emptylist.next:Jemptylist;
  emptylist.it:Jobject();
 }

 T scanprint(l:list)
 {
  while not(lJl.next) {
   println(l.it);
   l:Jl.next;
  }
 }
}



T main()
{

 for c:J2; cH8; c:JcI1; {
  if cH6 {c.println;} else {println("troppo!");}
 }
 .nl;

 "Ora vediamo una lista di elementi assortiti:".println;
```

```
 .nl;
 .emptylist.add(4).add(4.7).add("ciao").add(912.45M7e91).scanprint;
 .nl;
}
}
```

A.5.4. Dopo la fase due: test.out2

```
      basicKsystem :  uses  {



T list :  super object {
 it:object  ;
 next:list  ;


 T add (l:list ,x:object ):super object() {
  next(add):Jl;
  it(add):Jx;
 }


 T emptylist ():super object() {
  next(emptylist):Jemptylist;
  it(emptylist):Jobject();
 }


 T scanprint (l:list ) {
    while not(J(l,next(l))) {
   println(it(l));
   l:Jnext(l);
  }
 }
}




T main () {

 for c:J2; H(c,8); c:JI(c,1); {
    if H(c,6) {println(c);  } else {println("troppo!");}
 }
 nl();

 println("Ora vediamo una lista di elementi assortiti:");
```

```
 nl();
 scanprint(add(add(add(add(emptylist(),4),7(4)),"ciao"),M(45(912),7e91)));
 nl();
}
}
```

A.5.5. Dopo la fase tre: test.out3

```
 basicKsystem :  uses {
T list :  super object {
it:object ;
next:list ;
T add (l:list ,x:object ):super object() {
next(add):Jl;
it(add):Jx;
}
T emptylist ():super object() {
next(emptylist):Jemptylist;
it(emptylist):Jobject();
}
T scanprint (l:list ) {
while not(J(l,next(l))) {
println(it(l));
l:Jnext(l);
}
}
}
T main () {
for c:J2; H(c,8); c:JI(c,1); {
if H(c,6) {println(c); } else {println("troppo!");}
}
nl();
println("Ora vediamo una lista di elementi assortiti:");
nl();
scanprint(add(add(add(add(emptylist(),4),7(4)),"ciao"),M(45(912),7e91)));
nl();
}
}
```

A.5.6. Dopo la fase tre: test.out3.def

```
T list T object T
it :  object;
next :  list;
{ main , 0 T
:  :  }
{ scanprint , 1 T
list :  list :  }
{ emptylist , 0 T
:  list :  super }
{ add , 2 T
list,object :  list :  super }
N
```

A.5.7. Input per la fase quattro: test.outx

```
T object T  N T
T bool T object N T
T text T object N T
T num T object N T
T byte T num N T
T ubyte T num N T
T word T num N T
T uword T num N T
T long T num N T
T ulong T num N T
T quad T num N T
T uquad T num N T
T float T num N T
T double T num N T
{ object , 0 T
:  object :  super }
{ numzero , 0 T
:  num :  super }
{ I , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
text,text :  text :  text }
{ L , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
```

```
{ M , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
{ E , 2 T
ubyte,ubyte :  ubyte :  ubyte }
byte,byte :  byte :  byte }
uword,uword :  uword :  uword }
word,word :  word :  word }
ulong,ulong :  ulong :  ulong }
long,long :  long :  long }
uquad,uquad :  uquad :  uquad }
quad,quad :  quad :  quad }
float,float :  float :  float }
double,double :  double :  double }
{ L , 1 T
ubyte :  ubyte :  ubyte }
byte :  byte :  byte }
uword :  uword :  uword }
word :  word :  word }
ulong :  ulong :  ulong }
long :  long :  long }
uquad :  uquad :  uquad }
quad :  quad :  quad }
float :  float :  float }
double :  double :  double }
{ H , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
```

```
{ G , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
{ HJ , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
{ GJ , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
{ J , 2 T
object,object :  object :  bool }
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
```

```
bool,bool :  bool :  bool }
double,double :  double :  bool }
{ HG , 2 T
ubyte,ubyte :  ubyte :  bool }
byte,byte :  byte :  bool }
uword,uword :  uword :  bool }
word,word :  word :  bool }
ulong,ulong :  ulong :  bool }
long,long :  long :  bool }
uquad,uquad :  uquad :  bool }
quad,quad :  quad :  bool }
float,float :  float :  bool }
double,double :  double :  bool }
bool,bool :  bool :  bool }
{ and , 2 T
bool,bool :  bool :  bool }
{ or , 2 T
bool,bool :  bool :  bool }
{ not , 1 T
bool :  bool :  bool }
{ len , 1 T
text :  text :  long }
{ print , 1 T
object :  object :  }
ubyte :  ubyte :  }
byte :  byte :  }
uword :  uword :  }
word :  word :  }
ulong :  ulong :  }
long :  long :  }
uquad :  uquad :  }
quad :  quad :  }
float :  float :  }
double :  double :  }
bool :  bool :  }
text :  text :  }
{ println , 1 T
object :  object :  }
ubyte :  ubyte :  }
byte :  byte :  }
uword :  uword :  }
word :  word :  }
ulong :  ulong :  }
long :  long :  }
```

```
uquad :  uquad :  }
quad :  quad :  }
float :  float :  }
double :  double :  }
bool :  bool :  }
text :  text :  }
{ readtext , 0 T
:  text :  text }
{ nl , 0 T
:  :  }
T list T object T
it :  object;
next :  list;
{ main , 0 T
:  :  }
{ scanprint , 1 T
list :  list :  }
{ emptylist , 0 T
:  list :  super }
{ add , 2 T
list,object :  list :  super }
N
 basicKsystem :  uses {
T list :  super object {
it:object ;
next:list ;
T add (l:list ,x:object ):super object() {
next(add):Jl;
it(add):Jx;
}
T emptylist ():super object() {
next(emptylist):Jemptylist;
it(emptylist):Jobject();
}
T scanprint (l:list ) {
while not(J(l,next(l))) {
println(it(l));
l:Jnext(l);
}
}
}
T main () {
for c:J2; H(c,8); c:JI(c,1); {
if H(c,6) {println(c); } else {println("troppo!");}
```

```
}
nl();
println("Ora vediamo una lista di elementi assortiti:");
nl();
scanprint(add(add(add(add(emptylist(),4),7(4)),"ciao"),M(45(912),7e91)));
nl();
}
}
```

A.5.8. Dopo la fase quattro: result.symbolic

```
void *_m_add_list_object (
  Super call
  add_2->next:=l_2
  add_2->it:=x_2
  nothing to return:
void *_m_emptylist (
  Super call
  emptylist_3->next:=emptylist_3
  _tmp0:=NewObject "object"
  object(_tmp0)
  emptylist_3->it:=_tmp0
  nothing to return:
void *_m_scanprint_list (
  WHILE LOOP :
  _tmp2:=l_4->me._x_next
  _tmp3:=J(l_4,_tmp2)
  _tmp4:=not(_tmp3)
  WHILE _tmp4
  _tmp5:=l_4->me._x_it
  println(_tmp5)
  l_4:=l_4->me._x_next
  ENDWHILE
  nothing to return:
void *_m_main ()
  _tmp6:=2
  c_6:=_tmp6
  FOR LOOP :
  _tmp8:=8
  _tmp9:=H(c_6,_tmp8)
  FOR _tmp9
  _tmp10:=1
  c_6:=I(c_6,_tmp10)
  _tmp11:=6
  _tmp12:=H(c_6,_tmp11)
  IF _tmp12
  println(c_6)
  ELSE
  _tmp14:="troppo!"
  println(_tmp14)
  ENDIF
  ENDFOR
```

```
nl()
_tmp15:="Ora vediamo una lista di elementi assortiti:"
println(_tmp15)
nl()
_tmp16:=NewObject "list"
emptylist(_tmp16)
_tmp17:=4
_tmp18:=NewObject "list"
add(_tmp18,_tmp16,_tmp17)
_tmp19:=4.7
_tmp20:=NewObject "list"
add(_tmp20,_tmp18,_tmp19)
_tmp21:="ciao"
_tmp22:=NewObject "list"
add(_tmp22,_tmp20,_tmp21)
_tmp23:=912.45
_tmp24:=7e91
_tmp25:=M(_tmp23,_tmp24)
_tmp26:=NewObject "list"
add(_tmp26,_tmp22,_tmp25)
scanprint(_tmp26)
nl()
nothing to return:
```

A.5.9. Dopo la fase quattro: result.source

```
MTCclass *_y_object;
MTCclass *_y_bool;
MTCclass *_y_text;
MTCclass *_y_num;
MTCclass *_y_byte;
MTCclass *_y_ubyte;
MTCclass *_y_word;
MTCclass *_y_uword;
MTCclass *_y_long;
MTCclass *_y_ulong;
MTCclass *_y_quad;
MTCclass *_y_uquad;
MTCclass *_y_float;
MTCclass *_y_double;
MTCclass *_y_list;

void setupMTCclasses() {
 _y_object=new MTCclass("object",NULL);
 _y_bool=new MTCclass("bool",_y_object);
 _y_text=new MTCclass("text",_y_object);
 _y_num=new MTCclass("num",_y_object);
 _y_byte=new MTCclass("byte",_y_num);
 _y_ubyte=new MTCclass("ubyte",_y_num);
 _y_word=new MTCclass("word",_y_num);
 _y_uword=new MTCclass("uword",_y_num);
 _y_long=new MTCclass("long",_y_num);
 _y_ulong=new MTCclass("ulong",_y_num);
 _y_quad=new MTCclass("quad",_y_num);
 _y_uquad=new MTCclass("uquad",_y_num);
 _y_float=new MTCclass("float",_y_num);
 _y_double=new MTCclass("double",_y_num);
 _y_list=new MTCclass("list",_y_object);
}

#ifdef __MWERKS__
 #include "::includes:library.c"
#else
 #include "includes/library.c"
#endif

typedef struct _data_list {
```

```
  struct _data_object super;
  void *_x_it;
  void *_x_next;
};
typedef struct _x_list {
  short refNum;
  MTCclass *mytype;
  struct _data_list me;
};

//
// Super method, type:  list
//
void *_m_add_list_object (
           _x_list *add_2,
           _x_list *l_2,
           _x_object *x_2)
{
(void)(dispatchSuperMTCmessage("object",add_2));
add_2->me._x_next=(_x_list*)(l_2);
add_2->me._x_it=(_x_object*)(x_2);
return NULL;
}

//
// Super method, type:  list
//
void *_m_emptylist (
           _x_list *emptylist_3)
{
(void)(dispatchSuperMTCmessage("object",emptylist_3));
emptylist_3->me._x_next=(_x_list*)(emptylist_3);
_x_object *_tmp0=new _x_object;
_tmp0->mytype=_y_object;
(void)(dispatchSuperMTCmessage("object",_tmp0));
emptylist_3->me._x_it=(_x_object*)(_tmp0);
return NULL;
}

//
// no return value
//
void *_m_scanprint_list (
           _x_list *l_4)
```

```
{
loop_tmp1:
_x_list *_tmp2=(_x_list*)(l_4->me._x_next);
_x_bool *_tmp3=(_x_bool*)(dispatchMTCmessage("J",l_4,_tmp2));
_x_bool *_tmp4=(_x_bool*)(dispatchMTCmessage("not",_tmp3));
if (!get_bool(_tmp4)) goto end_tmp1;
_x_object *_tmp5=(_x_object*)(l_4->me._x_it);
(void)(dispatchMTCmessage("println",_tmp5));
l_4=(_x_list*)(l_4->me._x_next);
goto loop_tmp1;
end_tmp1:
return NULL;
}

//
// no return value
//
void *_m_main ()
{
_x_long *_tmp6=new_long(2);
_x_long *c_6=(_x_long*)(_tmp6);
top_tmp7:
_x_long *_tmp8=new_long(8);
_x_bool *_tmp9=(_x_bool*)(dispatchMTCmessage("H",c_6,_tmp8));
if (!get_bool(_tmp9)) goto end_tmp7;
goto for_tmp7;
loop_tmp7:
_x_long *_tmp10=new_long(1);
c_6=(_x_long*)(dispatchMTCmessage("I",c_6,_tmp10));
goto top_tmp7;
for_tmp7:
_x_long *_tmp11=new_long(6);
_x_bool *_tmp12=(_x_bool*)(dispatchMTCmessage("H",c_6,_tmp11));
if (!get_bool(_tmp12)) goto false_tmp13;
(void)(dispatchMTCmessage("println",c_6));
goto end_tmp13;
false_tmp13:
_x_text *_tmp14=new_text("troppo!");
(void)(dispatchMTCmessage("println",_tmp14));
end_tmp13:
goto loop_tmp7;
end_tmp7:
(void)(dispatchMTCmessage("nl"));
_x_text *_tmp15=new_text("Ora vediamo una lista di elementi assortiti:");
```

```
(void)(dispatchMTCmessage("println",_tmp15));
(void)(dispatchMTCmessage("nl"));
_x_list *_tmp16=new _x_list;
_tmp16->mytype=_y_list;
(void)(dispatchSuperMTCmessage("emptylist",_tmp16));
_x_long *_tmp17=new_long(4);
_x_list *_tmp18=new _x_list;
_tmp18->mytype=_y_list;
(void)(dispatchSuperMTCmessage("add",_tmp18,_tmp16,_tmp17));
_x_double *_tmp19=new_double(4.7);
_x_list *_tmp20=new _x_list;
_tmp20->mytype=_y_list;
(void)(dispatchSuperMTCmessage("add",_tmp20,_tmp18,_tmp19));
_x_text *_tmp21=new_text("ciao");
_x_list *_tmp22=new _x_list;
_tmp22->mytype=_y_list;
(void)(dispatchSuperMTCmessage("add",_tmp22,_tmp20,_tmp21));
_x_double *_tmp23=new_double(912.45);
_x_double *_tmp24=new_double(7e91);
_x_double *_tmp25=(_x_double*)(dispatchMTCmessage("M",_tmp23,_tmp24));
_x_list *_tmp26=new _x_list;
_tmp26->mytype=_y_list;
(void)(dispatchSuperMTCmessage("add",_tmp26,_tmp22,_tmp25));
(void)(dispatchMTCmessage("scanprint",_tmp26));
(void)(dispatchMTCmessage("nl"));
return NULL;
}

void setupMTCmessages() {
  MTCmessage *m;

  m=new MTCmessage("add",2);
  new MTCmethod(m,(void*)_m_add_list_object,_y_list,_y_object);
  m=new MTCmessage("emptylist",0);
  new MTCmethod(m,(void*)_m_emptylist);
  m=new MTCmessage("scanprint",1);
  new MTCmethod(m,(void*)_m_scanprint_list,_y_list);
  m=new MTCmessage("main",0);
  new MTCmethod(m,(void*)_m_main);
  m=new MTCmessage("nl",0);
  new MTCmethod(m,(void*)_m_nl);
  m=new MTCmessage("readtext",0);
  new MTCmethod(m,(void*)_m_readtext);
  m=new MTCmessage("println",1);
```

```
  new MTCmethod(m,(void*)_m_println_text,_y_text);
  new MTCmethod(m,(void*)_m_println_bool,_y_bool);

...omissis...

  m=new MTCmessage("object",0);
  new MTCmethod(m,(void*)_m_object);
}
```

A.5.10. Dopo la fase quattro: test.out4.1

```
MTCclass *_y_object;
MTCclass *_y_bool;
MTCclass *_y_text;
MTCclass *_y_num;
MTCclass *_y_byte;
MTCclass *_y_ubyte;
MTCclass *_y_word;
MTCclass *_y_uword;
MTCclass *_y_long;
MTCclass *_y_ulong;
MTCclass *_y_quad;
MTCclass *_y_uquad;
MTCclass *_y_float;
MTCclass *_y_double;
MTCclass *_y_list;

void setupMTCclasses() {
 _y_object=new MTCclass("object",NULL);
 _y_bool=new MTCclass("bool",_y_object);
 _y_text=new MTCclass("text",_y_object);
 _y_num=new MTCclass("num",_y_object);
 _y_byte=new MTCclass("byte",_y_num);
 _y_ubyte=new MTCclass("ubyte",_y_num);
 _y_word=new MTCclass("word",_y_num);
 _y_uword=new MTCclass("uword",_y_num);
 _y_long=new MTCclass("long",_y_num);
 _y_ulong=new MTCclass("ulong",_y_num);
 _y_quad=new MTCclass("quad",_y_num);
 _y_uquad=new MTCclass("uquad",_y_num);
 _y_float=new MTCclass("float",_y_num);
 _y_double=new MTCclass("double",_y_num);
 _y_list=new MTCclass("list",_y_object);
}

#ifdef __MWERKS__
 #include "::includes:library.c"
#else
 #include "includes/library.c"
#endif

typedef struct _data_list {
```

```
  struct _data_object super;
  void *_x_it;
  void *_x_next;
};
typedef struct _x_list {
  short refNum;
  MTCclass *mytype;
  struct _data_list me;
};

//
// Super method, type:  list
//
void *_m_add_list_object (
           _x_list *add_2,
           _x_list *l_2,
           _x_object *x_2)
{
                                    //  Super call
                                    //  object(add_2)
(void)(dispatchSuperMTCmessage("object",add_2));
                                    //  add_2->next:=l_2  // list
add_2->me._x_next=(_x_list*)(l_2);
                                    //  add_2->it:=x_2  // object
add_2->me._x_it=(_x_object*)(x_2);
                                    //  nothing to return:
return NULL;
}

//
// Super method, type:  list
//
void *_m_emptylist (
           _x_list *emptylist_3)
{
_x_object *_tmp0;
                                    //  Super call
                                    //  object(emptylist_3)
(void)(dispatchSuperMTCmessage("object",emptylist_3));
                                    //  emptylist_3->next:=emptylist_3  // list
emptylist_3->me._x_next=(_x_list*)(emptylist_3);
                                    //  _tmp0:=NewObject "object"
_tmp0=new _x_object;
_tmp0->mytype=_y_object;
```

```
                                    //  object(_tmp0)
(void)(dispatchSuperMTCmessage("object",_tmp0));
                                    //  emptylist_3->it:=_tmp0  // object
emptylist_3->me._x_it=(_x_object*)(_tmp0);
                                    //  nothing to return:
return NULL;
}

//
// no return value
//
void *_m_scanprint_list (
           _x_list *l_4)
{
_x_list *_tmp2;
_x_bool *_tmp3;
_x_bool *_tmp4;
_x_object *_tmp5;
                                    //  WHILE LOOP :
loop_tmp1:
                                    //  _tmp2:=l_4->me._x_next  // list
_tmp2=(_x_list*)(l_4->me._x_next);
                                    //  _tmp3:=J(l_4,_tmp2)  // bool
_tmp3=(_x_bool*)(dispatchMTCmessage("J",l_4,_tmp2));
                                    //  _tmp4:=not(_tmp3)  // bool
_tmp4=(_x_bool*)(dispatchMTCmessage("not",_tmp3));
                                    //  WHILE _tmp4
if (!get_bool(_tmp4)) goto end_tmp1;
                                    //  _tmp5:=l_4->me._x_it  // object
_tmp5=(_x_object*)(l_4->me._x_it);
                                    //  println(_tmp5)
(void)(dispatchMTCmessage("println",_tmp5));
                                    //  l_4:=l_4->me._x_next  // list
l_4=(_x_list*)(l_4->me._x_next);
goto loop_tmp1;
end_tmp1:
                                    //  ENDWHILE
                                    //  nothing to return:
return NULL;
}

//
// no return value
//
```

```
void *_m_main ()
{
_x_long *_tmp6;
_x_long *c_6;
_x_long *_tmp8;
_x_bool *_tmp9;
_x_long *_tmp10;
_x_long *_tmp11;
_x_bool *_tmp12;
_x_text *_tmp14;
_x_text *_tmp15;
_x_list *_tmp16;
_x_long *_tmp17;
_x_list *_tmp18;
_x_double *_tmp19;
_x_list *_tmp20;
_x_text *_tmp21;
_x_list *_tmp22;
_x_double *_tmp23;
_x_double *_tmp24;
_x_double *_tmp25;
_x_list *_tmp26;
                                    //  _tmp6:=2  // long
_tmp6=new_long(2);
                                    //  c_6:=_tmp6  // long
c_6=(_x_long*)(_tmp6);
                                    //  FOR LOOP :
top_tmp7:
                                    //  _tmp8:=8  // long
_tmp8=new_long(8);
                                    //  _tmp9:=H(c_6,_tmp8)  // bool
_tmp9=(_x_bool*)(dispatchMTCmessage("H",c_6,_tmp8));
                                    //  FOR _tmp9
if (!get_bool(_tmp9)) goto end_tmp7;
goto for_tmp7;
loop_tmp7:
                                    //  _tmp10:=1  // long
_tmp10=new_long(1);
                                    //  c_6:=I(c_6,_tmp10)  // long
c_6=(_x_long*)(dispatchMTCmessage("I",c_6,_tmp10));
goto top_tmp7;
for_tmp7:
                                    //  _tmp11:=6  // long
_tmp11=new_long(6);
```

```
                                        //  _tmp12:=H(c_6,_tmp11)  // bool
    _tmp12=(_x_bool*)(dispatchMTCmessage("H",c_6,_tmp11));
                                        //  IF _tmp12
    if (!get_bool(_tmp12)) goto false_tmp13;
                                        //  println(c_6)
    (void)(dispatchMTCmessage("println",c_6));
                                        //  ELSE
    goto end_tmp13;
    false_tmp13:
                                        //  _tmp14:="troppo!"  // text
    _tmp14=new_text("troppo!");
                                        //  println(_tmp14)
    (void)(dispatchMTCmessage("println",_tmp14));
                                        //  ENDIF
    end_tmp13:
    goto loop_tmp7;
                                        //  ENDFOR
    end_tmp7:
                                        //  nl()
    (void)(dispatchMTCmessage("nl"));
                                        //  _tmp15:="Ora vediamo una lista di elementi assortiti:"  //
text
    _tmp15=new_text("Ora vediamo una lista di elementi assortiti:");
                                        //  println(_tmp15)
    (void)(dispatchMTCmessage("println",_tmp15));
                                        //  nl()
    (void)(dispatchMTCmessage("nl"));
                                        //  _tmp16:=NewObject "list"
    _tmp16=new _x_list;
    _tmp16->mytype=_y_list;
                                        //  emptylist(_tmp16)
    (void)(dispatchSuperMTCmessage("emptylist",_tmp16));
                                        //  _tmp17:=4  // long
    _tmp17=new_long(4);
                                        //  _tmp18:=NewObject "list"
    _tmp18=new _x_list;
    _tmp18->mytype=_y_list;
                                        //  add(_tmp18,_tmp16,_tmp17)
    (void)(dispatchSuperMTCmessage("add",_tmp18,_tmp16,_tmp17));
                                        //  _tmp19:=4.7 // double
    _tmp19=new_double(4.7);
                                        //  _tmp20:=NewObject "list"
    _tmp20=new _x_list;
    _tmp20->mytype=_y_list;
```

```
                                    //  add(_tmp20,_tmp18,_tmp19)
(void)(dispatchSuperMTCmessage("add",_tmp20,_tmp18,_tmp19));
                                    //  _tmp21:="ciao"  // text
_tmp21=new_text("ciao");
                                    //  _tmp22:=NewObject "list"
_tmp22=new _x_list;
_tmp22->mytype=_y_list;
                                    //  add(_tmp22,_tmp20,_tmp21)
(void)(dispatchSuperMTCmessage("add",_tmp22,_tmp20,_tmp21));
                                    //  _tmp23:=912.45 // double
_tmp23=new_double(912.45);
                                    //  _tmp24:=7e91 // double
_tmp24=new_double(7e91);
                                    //  _tmp25:=M(_tmp23,_tmp24)  // double
_tmp25=(_x_double*)(dispatchMTCmessage("M",_tmp23,_tmp24));
                                    //  _tmp26:=NewObject "list"
_tmp26=new _x_list;
_tmp26->mytype=_y_list;
                                    //  add(_tmp26,_tmp22,_tmp25)
(void)(dispatchSuperMTCmessage("add",_tmp26,_tmp22,_tmp25));
                                    //  scanprint(_tmp26)
(void)(dispatchMTCmessage("scanprint",_tmp26));
                                    //  nl()
(void)(dispatchMTCmessage("nl"));
                                    //  nothing to return:
return NULL;
}

void setupMTCmessages() {
  MTCmessage *m;

  m=new MTCmessage("add",2);
  new MTCmethod(m,(void*)_m_add_list_object,_y_list,_y_object);
  m=new MTCmessage("emptylist",0);
  new MTCmethod(m,(void*)_m_emptylist);
  m=new MTCmessage("scanprint",1);
  new MTCmethod(m,(void*)_m_scanprint_list,_y_list);
  m=new MTCmessage("main",0);
  new MTCmethod(m,(void*)_m_main);
  m=new MTCmessage("nl",0);
  new MTCmethod(m,(void*)_m_nl);
  m=new MTCmessage("readtext",0);
  new MTCmethod(m,(void*)_m_readtext);
  m=new MTCmessage("println",1);
```

```
  new MTCmethod(m,(void*)_m_println_text,_y_text);
  new MTCmethod(m,(void*)_m_println_bool,_y_bool);

...omissis...

  m=new MTCmessage("object",0);
  new MTCmethod(m,(void*)_m_object);
}
```

A.5.11. In esecuzione: test.comp

```
2
3
4
5
troppo!
troppo!

Ora vediamo una lista di elementi assortiti:

6.38715e+94
ciao
4.7
4
```

Ringraziamenti

Desidero ringraziare innanzitutto i miei docenti, che, specialmente nell'ultimo periodo, mi hanno incoraggiato e spronato a concludere questo corso di studi; fra tutti vorrei esprimere in particolar modo la mia riconoscenza alla relatrice di questa tesi, Prof.ssa Maria Staniszkis, per la grande cortesia e per l'impagabile aiuto, nonchè per l'importante incoraggiamento ad approfondire i contenuti del corso di laurea con ulteriori studi. Vorrei ringraziare inoltre il Prof. Franco Parlamento ed il Prof. Carlo Tasso per la loro grande disponibilità.

Ringrazio qui tutti i miei amici, con cui ho trascorso anni importanti e indimenticabili, per essermi sempre stati vicini ed essere stati un po' la mia seconda famiglia, sopportando con pazienza la mia introversione nei periodi in cui dovevo studiare. Se dovessi elencarvi tutti finirei l'inchiostro, ma sappiate che siete tutti nel mio cuore!

Non può mancare inoltre un grazie speciale alla mia famiglia, che ha sempre rispettato le mie scelte nello studio e nel lavoro, il che ha rappresentato per me un importante segno di fiducia e di affetto. Questa tesi vuole essere per loro il mio ringraziamento.

Bibliografia

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[Ano94] Anonymous. Inheritance or delegation? *Byte Magazine*, 19(5):60, May 1994.

[App95] Apple Computer. *Dylan Reference Manual*, October 1995. (Draft).

[App96] Apple Computer. *Newton Programmer's Guide (For Newton 2.0)*. Addison-Wesley, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bal95] Stoney Ballard. *Dylan Competitive Analysis*. Apple Computer, February 1995.

[Bea94] Michel Beaudouin. *Object-Oriented Languages : Basic Principles and Programming Techniques*. Chapman & Hall, 1994.

[bis] *GNU Bison*. Documentazione in linea.

[BMP92] Doug Bell, Ian Morrey, and John Pugh. *Software Engineering, a programming approach*. Prentice-Hall, 2nd edition, 1992.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[Cha] Jane Chandler. *Introduction to Object-Oriented Programming Languages*. Department of Information Systems, University of Portsmouth, `http://www.sis.port.ac.uk/~chandler/OOLectures/oopl/oopl.htm`. (basato su [Bea94]).

[Clo] *The Common Lisp Object System*, `http://iahost.dis.ulpgc.es/clos.html`. (basato su [Gra96]).

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, Berlin, 1981.

[CM91] Antonio Cunei and Marino Miculan. *OOPLog, progetto per l'Esame di Linguaggi di Programmazione (Prof. Carlo Tasso)*. Università degli Studi di Udine, 1991.

[Dal97] Jeff Dalton. *Brief Guide to CLOS*. University of Edinburgh, August 1997, `http://www.tnt.uni-hannover.de/data/www/soft/case/lang/lisp/clos.html`.

[Deb95] Clive Debenham. *An Introduction To TAOS*. Tantric Technologies, March 1995, `tantric@cix.compulink.co.uk`.

[Eck] Bruce Eckel. *Thinking in C++*, 2.0 edition, `http://www.bruceeckel.com/`. (to be published by Prentice-Hall; downloadable in RTF format).

[Edi86] Edia Borland s.r.l., V. Cirene, 11 - 20135 Milano. *Turbo Pascal - Versione 3.0*, December 1986.

[Fla97] D. Flanagan. *Java In A Nutshell*. A Nutshell Handbook. O'Reilly, 2nd edition, 1997.

[Gar] *Garbage Collection FAQ*, `http://www.iecc.com/gclist/GC-faq.html`.

[gcc] *GNU C Compiler*. Documentazione in linea.

[Geh84] Narain Gehani. *Ada, an advanced introduction (including reference manual for the Ada programming language*. Prentice-Hall software series. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Gol84] Adele Goldberg. *Smalltalk-80: the interactive programming environment*. Addison-Wesley computer science. Addison-Wesley, Reading (Mass.), 1984.

[Gra96] Paul Graham. *ANSI Common Lisp*. Prentice Hall series in artificial intelligence. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[Gro92] Peter Grognono. *Programmare in Pascal e Turbo Pascal*. Franco Muzzio Editore, 1992.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Trans. Softw. Eng.*, SE-16(12):1344–1351, October 1990.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, October 1992.

[HRB+87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The emerald programing lanuage report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987.

[Hut87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, University of Washington, Computer Science Department, January 1987.

[IBM] IBM Corporation, Object Technology Products Group, Austin, Texas. *The System Object Model (SOM) and the Component Object Model (COM): A comparison of technologies from a developer's perspective*.

[Ins] Institut für Computersysteme - ETH Zürich. *The Official Oberon Home Page*, `http://www.oberon.ethz.ch`.

[Jon] Richard Jones. *The Garbage Collection Page*, `http://stork.ukc.ac.uk/computer_science/Html/Jones/gc.html`.

[JRH88] Eric Jul, Rajendra K. Raj, and Norman C. Hutchinson. The emerald system user's guide. Technical Report Ver. 1.3, Department of Computer Science, University of Washington, Seattle, November 1988.

[Jul88] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, University of Washington, Computer Science Department, December 1988.

[KCC+97] G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, R. Morrison, D.S. Munro, and S. Scheuerl. Flask: An architecture supporting concurrent distributed persistent applications. Technical Report CS/97/4, University of St Andrews, Scotland, 1997.

[KM] G.N.C. Kirby and R. Morrison. *A Persistent View of Encapsulation*. St Andrews, Fife KY16 9SS, Scotland.

[KM97] G.N.C. Kirby and R. Morrison. Orthogonal persistence as an implementation platform for software development environments. Technical Report CS/97/6, University of St Andrews, Scotland, 1997.

[KR89] Brian W. Kernigan and Dennis M. Ritchie. *Il Linguaggio C*. Jackson, 1989.

[Kur96] Kurt Nørmark. *Hooks and Open Points*, 1996, `http://www.cs.auc.dk/~normark/hooks/hypertext/hooks.html`.

[Mar93] Dave Mark. *Learn C++ on the Macintosh*. Addison-Wesley, 1993.

[Mau96] Rainer Mauth. A better foundation: Development frameworks let you build an application with reusable objects. *Byte Magazine*, 21(9), September 1996. (International Features Section).

[Met] Metrowerks Inc. *CodeWarrior Pascal: Language Reference*, `http://www.jstream.com/java11jpds/pascal/pascalbook.html`.

[Mod] *Modula-3 Home Page*, `http://www.research.digital.com/SRC/modula-3/html/home.html`.

[Mot92] Motorola Inc. *M68000 Family Programmer's Reference Manual*, 1992. (Codice: M68000PM/AD).

[Mot93] Motorola Inc. *PowerPC^{TM} 601 RISC Microprocessor User's Manual*, 1993. (Codice: MPC601UM/AD).

[Mot94] Motorola Inc. *DSP96002 32-bit Digital Signal Processor User's Manual*, 1994. (Codice: DSP96002UM/AD).

[MS97] Leonid Mikhajlov and Emil Sekerinski. The fragile base class problem and its solution. Technical Report TUCS Technical Report No 117, Turku Centre for Computer Science, June 1997.

[New] NewMonics Inc. *Discussions on Real-time Garbage Collection*, `http://www.newmonics.com/webroot/technologies/gc/`.

[Obea] *Oberon V3 Pages*, `http://caesar.ics.uci.edu/oberon`.

[Obeb] *Oberon V4 Pages*, `http://www.ssw.uni-linz.ac.at/Oberon.html`.

[Obec] Oberon microsystems. *Component Software: A Case Study Using BlackBox Components*. Preliminary version.

[Orn96] David Ornstein. *Garbage Collection in Smalltalk/V*, June 1996, `http://www.parcplace.com/support/vsesupp/TIPS/note2481.htm`.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1999.

[PC93] Jill Nicola Peter Coad. *Object-oriented programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[Pes95a] Carlo Pescio. *C++, Manuale di Stile*. Infomedia, Pisa, 1995.

[Pes95b] Carlo Pescio. Il problema della "fragile base class" in c++. *Computer Programming*, 41, September 1995.

[Proa] *IC Prolog][, manuale in linea*, `http://laotzu.doc.ic.ac.uk/Localinfo/icprolog.html`.

[Prob] *SICStus Prolog User's Manual*, `http://www.sics.se/isl/sicstus`.

[PS94] Dick Pountain and Clemens Szyperski. Extensible software systems: New programming tools are needed to develop software systems tha can be easily extended with new modules. *Byte Magazine*, 19(5):57, May 1994.

[PvE96] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean, Language Report (Version 1.1)*, March 1996.

[Pyl81] I.C. Pyle. *The ADA programming language: a guide for programmers*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

[RTL+] Rajindra K. Raj, Evan Tampero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. *Emerald: A General-Purpose Programming Language*. Seattle.

[SB86] M. Stefik and D. Bobrow. Object oriented programming: Themes and variations. *AI Magazine*, 6(4), 1986.

[Sch96] Herbert Schildt. *Guida al Linguaggio C++*. McGraw-Hill, 1996.

[Sha97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[SMR89] A. Straw, F. Mellender, and S. Riegel. Object management in a persistent smalltalk system. *Software–Practice and Experience*, 19(8):719–737, August 1989.

[Str94] Bjarne Stroustrup. *The design and evolution of C++*. Addison-Wesley, Reading, 1994.

[Sun] Sun Microsystem. *Java OS*, `http://java.sun.com/products/javaos/`.

[Sun95] Sun Microsystems. *The Java Virtual Machine Specification*, 1.0 beta edition, August 1995, `http://java.sun.com/doc/vmspec/VMSpec.ps`.

[Uni] University of California at Riverside. *Home Page for DYLAN Language*, `http://cuda.ucr.edu/Page_lang/inet_links/dylan.html`.

[Val] Andrew Valencia. *A Tutorial for GNU Smalltalk*. Valencia Consulting. (documento distribuito insieme a GNU Smalltalk 1.1.5).

[Way94] Peter Wayner. Objects on the march: The trend is toward an object-oriented approach to the design of operating systems. *Byte Magazine*, 19(1):139, January 1994.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 September 1992. Springer-Verlag, `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`.

[Wir] Niklaus Wirth. *A Brief History of Modula and Lilith*, `http://www.modulaware.com/mdlt52.htm`.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Texts and monographs in computer science. Springer, Berlin, 3rd edition, 1985.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[Wya94] Geoff Wyant. Introducing modula-3. *Linux Journal*, 8, December 1994, `ftp://ftp.gte.com/pub/m3/linux-journal.html`.

[Zan91] Fausto Zanasi. *Teoria e pratica del linguaggio Prolog*. Calderini, 1991.